# Chaotic Routing – Design and Implementation of an Adaptive Multicomputer Network Router

by

Kevin Bolding

A dissertation submitted in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy

University of Washington

1993

Approved by_____
(Chairperson of Supervisory Committee)

Program Authorized
to Offer Degree_____

Date_____

In presenting this dissertation in partial fulfillment of the requirements for the Doctoral degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that extensive copying of this dissertation is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for copying or reproduction of this dissertation may be referred to Univerity Microfilms, 1490 Eisenhower Place, P.O. Box 975, Ann Arbor, MI, 48106, to whom the author has granted "the right to reproduce and sell (a) copies of the manuscript in microform and/or (b) printed copies of the manuscript made from microform."

Signature_____

Date_____

University of Washington

Abstract

# Chaotic Routing – Design and Implementation of an Adaptive Multicomputer Network Router

by Kevin Bolding

Chairperson of Supervisory Committee: Professor Lawrence Snyder
Department of
Computer Science and Engineering

A crucial component of a massively parallel multicomputer is the interconnection network which links all of the nodes of the computer together. This network provides the primary method of communication between the hundreds or thousands of processing nodes and is, thus, critical to the successful operation of the multicomputer.

Current state-of-the-art interconnection networks use simple, oblivious routing techniques which achieve very good performance when loading is light, but do not perform well in the presence of non-uniform congestion or faults. Chaotic routing, a non-minimal adaptive routing technique, provides a mechanism which takes into account the presence of congestion and faults when choosing a path for a message and can, thus, achieve better performance.

Chaotic routing is shown to be applicable to all finite-sized networks of bounded degree with bi-directionally connected links. A design for a chaotic router is presented which includes both the features of virtual cut-through routing and the advantages of internal non-blocking buffering. This design allows messages to follow minimum-latency paths when network loading is light, and provides sufficient buffering to ensure high throughput when the network is heavily loaded.

The resulting design is compared with other minimal and non-minimal adaptive designs, as well as with oblivious routing. Chaotic routing is shown to be superior to most other adaptive designs due to its lower design complexity, and simulations are used to compare it with oblivious and deflection routing. The simulations show

that, for mesh-connected networks, chaotic routing performs only slightly better than oblivious routing. However, for torus- and hypercube-connected networks, chaotic routing is superior to oblivious and deflection routing, achieving much higher throughput and lower latency.

To demonstrate the feasibility of chaotic routing, a prototype chaos router chip is presented. This chip, fabricated in 1.2$\mu$m CMOS, implements a two-dimensional chaos router. The chip, in simulations, operates at 66MHz, with the speed being limited only by the speed at which the 5V CMOS pads can be switched. Thus, the chip should operate at the same clock speed as oblivious routers using the same technology, giving an equivalent bandwidth capability.

# Table of Contents

**List of Figures** v

**List of Tables** viii

**Chapter 1: Introduction** 1

1.1 Multicomputer Networks . . . . . 1
1.2 Network Topologies . . . . . 2
1.2.1 Network Characterization . . . . . 4
1.3 Flow Control Techniques . . . . . 8
1.4 Network Routing . . . . . 10
1.4.1 Oblivious Routing . . . . . 11
1.4.2 Minimal Adaptive Routing . . . . . 14
1.4.3 Non-minimal Adaptive Routing . . . . . 16
1.5 Summary . . . . . 20

**Chapter 2: Generalized Chaotic routing** 21

2.1 Previous Work . . . . . 21
2.1.1 Conceptual Chaotic Routing . . . . . 21
2.1.2 Chaos Router Architecture . . . . . 23
2.2 Chaotic Routing in Non-hypercubic Networks . . . . . 24
2.2.1 Deadlock Freedom . . . . . 24
2.2.2 Livelock Freedom . . . . . 25
2.3 The Cut-through Chaos Router . . . . . 27
2.4 Comparisons with Other Routers . . . . . 33
2.4.1 The Oblivious Decision. . . . . 33
2.4.2 The Adaptive Decision. . . . . 34
2.5 Oblivious Routers – The State-of-the-art . . . . . 34
2.6 Minimal Adaptive Routers . . . . . 35

2.7 Deflection Non-minimal Adaptive Routers . . . . . 36
2.8 Queueing Non-minimal Adaptive Routers . . . . . 37
2.8.1 Livelock prevention . . . . . 37
2.8.2 Queue management . . . . . 38
2.9 Fault-tolerance . . . . . 39
2.10 Out-of-order Packet Delivery . . . . . 39
2.11 Summary . . . . . 40

**Chapter 3: Detailed Chaos Router Design 42**
3.1 Design Goals . . . . . 42
3.1.1 Uncongested Network Performance . . . . . 42
3.1.2 Congested Network Performance . . . . . 49
3.2 Chaos Router Design . . . . . 50
3.2.1 Packets and Data . . . . . 50
3.3 Core Router Design . . . . . 52
3.4 Multiqueue Design . . . . . 55
3.4.1 Multiqueue Scoreboarding . . . . . 58
3.5 Channel Design . . . . . 62
3.6 Performance . . . . . 65
3.6.1 Multiqueue Performance . . . . . 65
3.6.2 Core router performance . . . . . 74
3.7 Summary . . . . . 75

**Chapter 4: Simulation Studies 76**
4.1 Routers Compared . . . . . 76
4.1.1 Chaotic Routing . . . . . 77
4.1.2 Oblivious Routing . . . . . 77
4.1.3 Deflection Routing . . . . . 79
4.2 Traffic Models . . . . . 79
4.3 Simulation Methodology . . . . . 80
4.4 MultiQueue Size Experiments . . . . . 81
4.5 Shared vs. Non-shared Channels Experiments . . . . . 86

4.6 Chaotic vs. Oblivious Routing Experiments . . . . . 92
4.6.1 Uniform Random Traffic . . . . . 92
4.6.2 Hot Spot Traffic . . . . . 99
4.7 Chaotic vs. Deflection Routing Experiments . . . . . 104
4.8 Summary . . . . . 104

**Chapter 5: Implementation – Mesh/Torus Router Chip 107**
5.1 Methodology . . . . . 107
5.2 Data paths . . . . . 109
5.2.1 Channel . . . . . 109
5.2.2 FIFO . . . . . 110
5.2.3 Crossbar . . . . . 112
5.2.4 Incrementer/Decrementer . . . . . 115
5.3 Control logic . . . . . 117
5.3.1 Routing pipeline . . . . . 117
5.3.2 Header decode . . . . . 119
5.3.3 Output Frame . . . . . 120
5.4 MultiQueue . . . . . 121
5.4.1 Multiqueue Input Logic . . . . . 123
5.4.2 Multiqueue Scoreboard . . . . . 124
5.5 Channel Controller . . . . . 126
5.6 Results . . . . . 129
5.6.1 Simulations . . . . . 129
5.7 Summary . . . . . 132

**Chapter 6: Conclusions and Remaining Work 135**
6.1 Conclusions . . . . . 135
6.2 Remaining Work . . . . . 138
6.2.1 Chaos Chip Characterization . . . . . 139
6.2.2 Broadened Simulation Analysis . . . . . 140
6.2.3 Fault Detection . . . . . 140
6.2.4 Fault Recovery . . . . . 141

6.3 Summary . . . . . 141

**Bibliography** **143**

**Appendix A: Extended Simulation Results – 64-node Networks** **151**

**Appendix B: Extended Simulation Results – 1024-node Networks** **164**

# List of Figures


1.1 A typical computational node . . . . . 2
1.2 A portion of a mesh-connected interconnection network . . . . . 3
1.3 Three 16-node networks. (a) 2-d mesh. (b) 2-d torus. (c) 4-d hypercube. 5
1.4 Basic flow-control methods. (a) Circuit-switched. (b) Packet-switched. 9
1.5 Dimension order (X, then Y) oblivious routing. . . . . 12
1.6 Minimal adaptive routing. . . . . 15
1.7 Non-minimal adaptive routing. (a) Routing around congestion. (b) Detouring around a faulty node. . . . . 17

2.1 Conceptual hypercube chaos router. . . . . 22
2.2 Probability of message delivery, $P(i)$, with $Diameter_{net} = 25$. . . . . 28
2.3 Number of games to provide 95% confidence of delivery. . . . . 29
2.4 A conceptual two-dimensional minimal cut-through router. . . . . 30
2.5 A conceptual two-dimensional cut-through chaos router. . . . . 31
2.6 Violation of the packet-exchange protocol when using cut-through. . . 32

3.1 Ratio of minimum average latency with chaotic routing to minimum average latency with oblivious routing: 2-D mesh network. . . . . 46
3.2 Ratio of minimum average latency with chaotic routing to minimum average latency with oblivious routing: 2-D torus network. . . . . 47
3.3 Ratio of minimum average latency with chaotic routing to minimum average latency with oblivious routing: Hypercube network. . . . . 48
3.4 The six interesting FIFO states. . . . . 51
3.5 *ActionDim* selection for a two-dimensional router. . . . . 54
3.6 Logical MultiQueue diagram. $d = 5$ and $Q = 5$. . . . . 56
3.7 Multiqueue destination matrix. . . . . 59
3.8 Multiqueue storage matrix. . . . . 61
3.9 Uni-directional vs. Bi-directional Channels. . . . . 63

3.10 Worst case number of non-routable packets in the multiqueue with $Q = 5$ and $L = 20$. . . . . 68
3.11 Probability of routing within $n$ cycles for various queue sizes, $L = 20$. 73

4.1 Virtual channel deadlock prevention scheme . . . . . 78
4.2 256-node chaotic mesh: shared vs nonshared (Uniform traffic) . . . . 88
4.3 256-node chaotic mesh: shared vs nonshared (4X hot spot traffic) . . 89
4.4 256-node chaotic torus: shared vs nonshared (Uniform traffic) . . . . 90
4.5 256-node chaotic torus: shared vs nonshared (4X hot spot traffic) . . 91
4.6 256-node shared channel hypercube: chaotic vs oblivious (Uniform traffic) . . . . . 93
4.7 256-node shared channel mesh: chaotic vs oblivious (Uniform traffic) 94
4.8 256-node shared channel torus: chaotic vs oblivious (Uniform traffic) 95
4.9 Updated results: 256-node shared channel torus: chaotic vs oblivious (Uniform traffic) . . . . . 96
4.10 Average injection delay for a 256-node mesh with chaotic routing . . 98
4.11 256-node shared channel hypercube: chaotic vs oblivious (4X hot spot traffic) . . . . . 100
4.12 256-node shared channel mesh: chaotic vs oblivious (4X hot spot traffic)101
4.13 256-node shared channel torus: chaotic vs oblivious (4X hot spot traffic)102
4.14 256-node nonshared channel mesh: chaotic vs deflection (Uniform traffic)105
4.15 256-node nonshared channel torus: chaotic vs deflection (Uniform traffic)106

5.1 Two phase non-overlapping clock. . . . . 108
5.2 Pipeline constructed with two-phase level-sensitive dynamic latches. . 109
5.3 FIFO diagram. . . . . 111
5.4 FIFO layout. . . . . 113
5.5 Crossbar unit layout. . . . . 114
5.6 Crossbar Connect cell layout. . . . . 116
5.7 Control Logic routing Pipeline. . . . . 118
5.8 Output frame match logic. . . . . 122
5.9 Multiqueue Scoreboard diagram. . . . . 125
5.10 Chaos Router Floorplan. . . . . 130

5.11 Chaos Router Chip Layout. . . . . 131
5.12 Simulator Environment. . . . . 133
5.13 Simulator Packet Format. . . . . 134

A.1 64-node shared channel hypercube: chaos vs oblivious (Uniform traffic) 152
A.2 64-node shared channel hypercube: chaos vs oblivious (4X hot spot traffic) . . . . . 153
A.3 64-node shared channel mesh: chaos vs oblivious (Uniform traffic) . . 154
A.4 64-node shared channel mesh: chaos vs oblivious (4X hot spot traffic) 155
A.5 64-node shared channel torus: chaos vs oblivious (Uniform traffic) . . 156
A.6 64-node shared channel torus: chaos vs oblivious (4X hot spot traffic) 157
A.7 64-node nonshared channel mesh: chaos vs deflection (Uniform traffic) 158
A.8 64-node nonshared channel torus: chaos vs deflection (Uniform traffic) 159
A.9 64-node chaos mesh: shared vs nonshared (Uniform traffic) . . . . . 160
A.10 64-node chaos mesh: shared vs nonshared (4X hot spot traffic) . . . . 161
A.11 64-node chaos torus: shared vs nonshared (Uniform traffic) . . . . . 162
A.12 64-node chaos torus: shared vs nonshared (4X hot spot traffic) . . . . 163

B.1 1024-node shared channel hypercube: chaos vs oblivious (Uniform Traffic) . . . . . 165
B.2 1024-node shared channel hypercube: chaos vs oblivious (4X hot spot traffic) . . . . . 166
B.3 1024-node shared channel mesh: chaos vs oblivious (Uniform traffic) . 167
B.4 1024-node shared channel mesh: chaos vs oblivious (4X hot spot traffic)168
B.5 1024-node shared channel torus: chaos vs oblivious (Uniform traffic) . 169
B.6 1024-node shared channel torus: chaos vs oblivious (4X hot spot traffic)170
B.7 1024-node nonshared channel mesh: chaos vs deflection (Uniform traffic)171
B.8 1024-node nonshared channel torus: chaos vs deflection (Uniform traffic)172
B.9 1024-node chaos mesh: shared vs nonshared (Uniform traffic) . . . . . 173
B.10 1024-node chaos mesh:shared vs nonshared (4X hot spot traffic) . . . 174
B.11 1024-node chaos torus: shared vs nonshared (Uniform traffic) . . . . . 175
B.12 1024-node chaos torus: shared vs nonshared (4X hot spot traffic) . . . 176

# List of Tables


3.1 Delay in waiting for packets to become routable for various packet lengths ($L$) and queue sizes ($Q$). . . . . 70
3.2 Queue delay characterization for a variety of packet and queue sizes. . 72

4.1 256-node mesh throughput as a function of queue size (Uniform traffic) 82
4.2 256-node mesh latency as a function of queue size (Uniform traffic) . 82
4.3 256-node mesh throughput as a function of queue size (Hot spot traffic) 83
4.4 256-node mesh latency as a function of queue size (Hot spot traffic) . 83
4.5 256-node torus throughput as a function of queue size (Uniform traffic) 84
4.6 256-node torus latency as a function of queue size (Uniform traffic) . 84
4.7 256-node torus throughput as a function of queue size (Hot spot traffic) 85
4.8 256-node torus latency as a function of queue size (Hot spot traffic) . 85

5.1 Channel Arbitration Protocol Signals. . . . . 128

# Acknowledgments

Many people have helped to make my five years spent at the University of Washington some of the most significant of my life. Larry Snyder, my advisor, deserves special thanks not only due to his invaluable support, teaching, and guidance in his role as my advisor, but also for the caring and supportive way he has helped me through other events affecting my life during this time.

I am indebted to Smaragda Konstantindou for the original concept of chaotic routing, as presented in her PhD dissertation. Also, Carl Ebeling has provided an enormous amount of insight into the intricacies of VLSI design.

Sen-Ching Cheung and Melanie Lewis Fulgham have helped tremendously in running and analyzing the hundreds of simulations necessary for this work.

The chaos router chip is the result of the combined effort of many individuals. Special thanks go to Sen-Ching Cheung, Soha Hassoun, Ton Ngo, and Robert Wille for extra effort on this project. Shyng Chen, Sung-Eun Choi, Pai Chou, David Hubbell, Donald Lobo, Miles Ohlrich, Wayne Ohlrich, Ross Ortega, and Peter Sullivan also contributed to the design. Finally, Larry McMurchie has proved himself invaluable through his help in coordinating the myriad of design tools necessary for this project.

I would like to thank my pastor, Ron Jacobson, and my friend, Miriam Wetzel, for their kindness, friendship, and support during my time here at graduate school.

My parents, Darrell and Patricia Bolding, have been an endless source of support, love, and encouragement since day one. Without their guidance, teaching, and motivation, I would never have come this far.

My final thanks go to God, revealed to us by His son, Jesus Christ, without whom I would be, literally, nothing.

# Dedication

In the service of Jesus Christ, my Lord.

Whatever you do, work at it with all your heart, as working for the Lord, not for men.

– *Colossians 3:23*

# Chapter 1

# INTRODUCTION

In the never-ending quest for higher-performance computers, the *multicomputer* has shown promise. Multicomputers are networks of independent computers which share data through the use of a very high bandwidth local network. Typically, a multicomputer is composed of from 16 to 1024 or more computational *nodes* connected in a regularly structured network. By connecting together hundreds or thousands of moderately-priced nodes into one large multicomputer network, a cost-effective method of achieving high-performance computing is realized.

## *1.1 Multicomputer Networks*

A multicomputer consists of many *nodes* connected together via an interconnection network. Strictly speaking, a network node may contain a computer, an I/O server, or any other device which communicates with the network. However, a typical node is a *computational* node with a high-performance CPU and some amount of local memory. The computational node is usually connected to the interconnection network through a communication co-processor and a network router. Figure 1.1 shows a typical computational node, with a CPU and memory connected via a local bus to a communication co-processor. In this example, the local bus conforms to the processor's protocol, and the communication co-processor converts the local bus protocol to the network protocol. The communication co-processor may be quite powerful, performing direct memory access to the local memory and automatically performing packetization, if needed, to convert messages to and from the network protocol.

Communication with other networked components is accomplished by sending and receiving messages through the communication co-processor to the processor port of the associated network router. The interconnection network consists of routers and channels. The channels are nothing more than wires which connect adjacent routers,

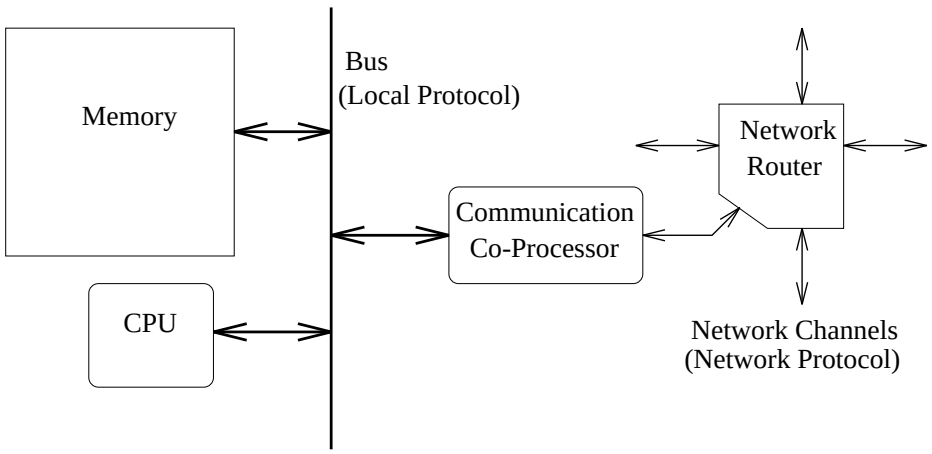



Figure 1.1: A typical computational node

and the routers simply pass messages along from their sources to their destinations, delivering messages to their destination nodes when appropriate. Figure 1.2 shows a portion of a typical interconnection network, in this case a two-dimensional mesh-connected network.

There are many factors which determine how messages flow through multicomputer interconnection networks. These can be broken down into three basic areas [Dally 90]: network topology, routing, and flow control. This dissertation concerns primarily routing techniques, though many topological and flow control issues are relevant. Therefore, topology and flow control will be discussed briefly first, followed by extensive discussion of routing techniques.

## *1.2 Network Topologies*

Interconnection networks may vary significantly in size and topology. Networks may contain anywhere from 2 to 16384 or more nodes. Because multicomputers rely on fast point-to-point communication between many closely associated nodes, most multicomputer networks are very regular in nature, belonging to the class of $k$-ary $d$-cubes. In this class of networks, the network contains $N = k^d$ nodes. Commonly referred to as "radix $k$, dimension $d$" networks, members of this class have $d$ dimensions, with each being $k$ nodes on a side. Assuming a $d$ dimensional address of the form $(a_{d-1}, a_{d-2}, \ldots, a_1, a_0)$, where each $a_n$ is in the range $(0, \ldots, k-1)$, a node is

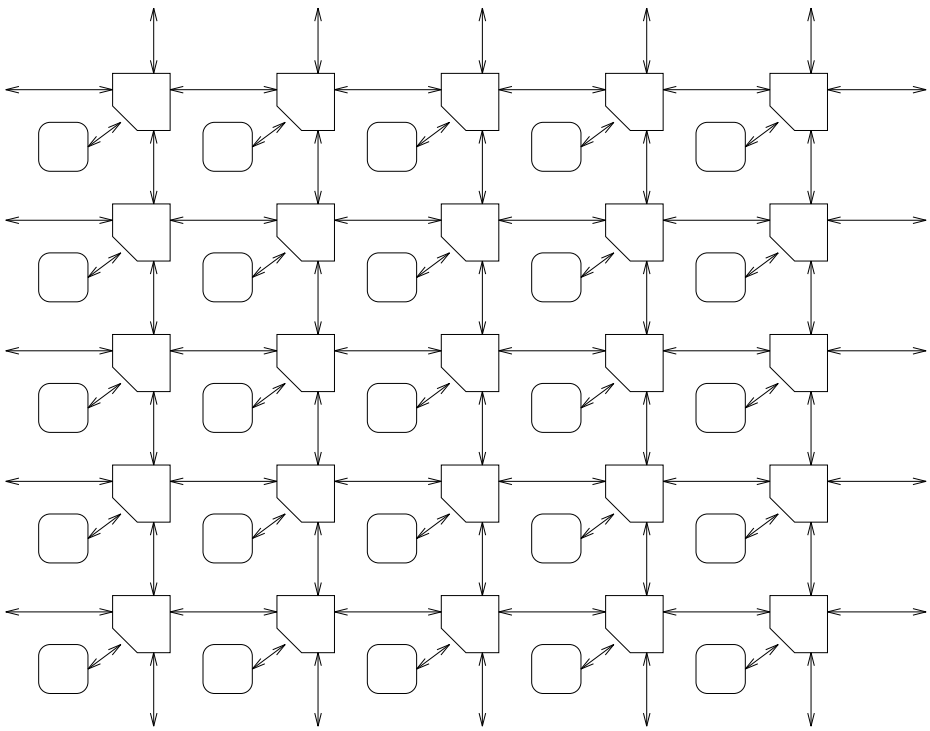


Figure 1.2: A portion of a mesh-connected interconnection network

connected to $2d$ neighbors,[1] those which have addresses which differ by $\pm 1$ in exactly one of the $d$ dimensions. When a neighbor would have a non-existent address, i.e., one of the dimensions is greater than $k-1$ or less than zero, there are two ways of connecting the link. The first is to ignore the non-existent neighbor and not include the link at all. Thus, this creates "open-ended" networks which have terminal edges. The second method of dealing with the "edges" is to add a *wraparound* link from the first element in a dimension to the last element in that dimension, i.e. use "$(a_i \pm 1) \bmod k$" when computing the neighboring nodes instead of just "$a_i \pm 1$".

In general, a $k$-ary $d$-cube with open edges is known as a $d$-dimensional mesh, and as a $d$-dimensional torus if the edges are wrapped around. The special case where $k = 2$ produces a $d$-dimensional binary hypercube. A binary hypercube with $N$ nodes has $d = \log_2 N$ dimensions. It is interesting to note that a binary hypercube of dimension $d = \log_2 N$, where $d$ is even, is the same network as a torus with $k = 4$ and $d = \frac{1}{2}\log_2 N$.

---

[1] When $k = 2$, as in the binary hypercube, each node has only $d$ distinguishable neighbors.

Some common network topologies are shown in Figure 1.3. Each of the networks contains 16 nodes. In (a), a two-dimensional mesh, with $k = 4$ and $d = 2$ is shown. The edges are not "wrapped around" in this example. Figure 1.3(b) shows a two-dimensional torus, with $k = 4$ and $d = 2$ and edges "wrapped around." Finally, in (c), a four-dimensional binary hypercube is shown, which has $d = 4$ and $k = 2$. Notice that, in this case, the binary hypercube with $d = 4$ is the same network as the torus with $d = 2$, as each of the nodes has connections to the same four neighbors in each illustration.

In the remainder of this dissertation, the focus will be on mesh, torus, and hypercube networks. Comparisons between different network topologies will be limited, as the concentration of this work is on the routing of packets within a given network. Comparative studies and analysis on network topologies can be found in [Abraham 90, Agarwal 91, Abraham & Padmanabhan 91, Bolding & Konstantinidou 92, Chittor & Enbody 90, Dally 87].

### *1.2.1 Network Characterization*

Other than the topology itself, there are several characteristics of networks that help in analyzing the networks and their performance. Of special interesting when dealing with routing performance are the *diameter* of the network, the *bisection bandwidth* of the network, and, for traffic with randomly chosen sources and destinations, the average distance traveled and the maximum throughput possible.

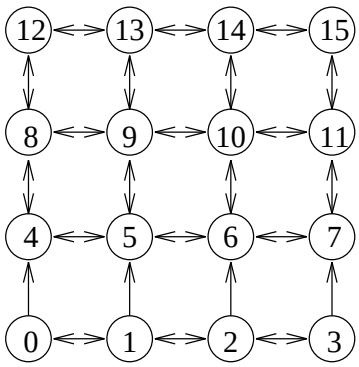
A network's *diameter* describes the maximum distance between any two nodes in the network. This can be computed by summing up the maximum distances that can be traveled in each of the $d$ dimensions. For open-ended networks, the maximum displacement is $k - 1$ in each dimension, so:

$$Diameter_{open} = d \cdot (k - 1) \tag{1.1}$$

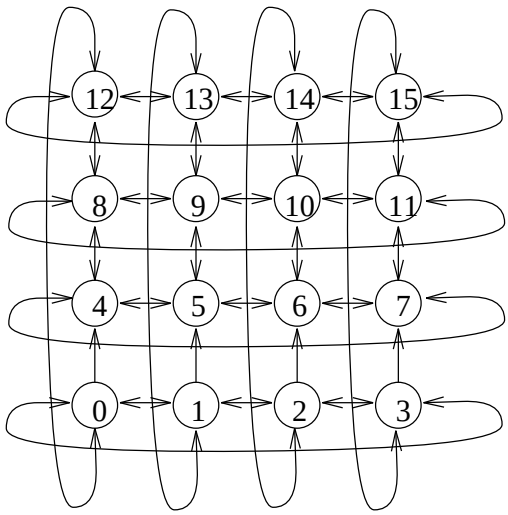
Relating this to the number of nodes in a network, $N = k^d$:

$$Diameter_{mesh} = 2\left(\sqrt{N} - 1\right)$$

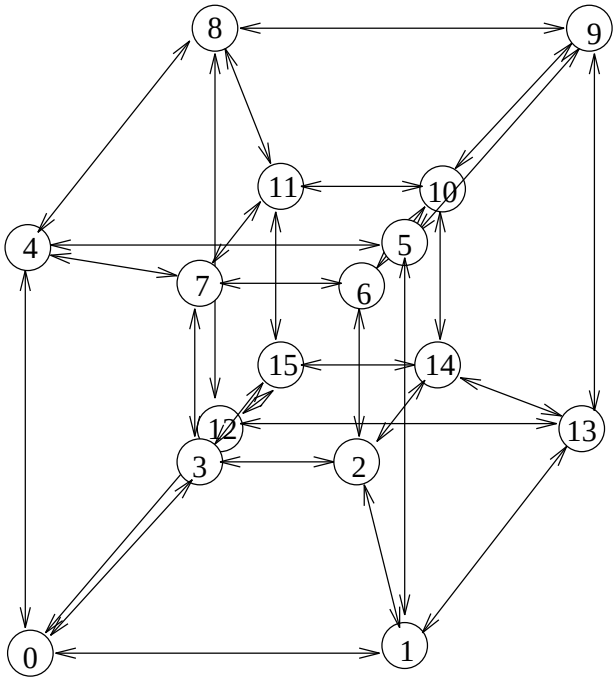
$$Diameter_{hyp} = \log_2(N)$$




(a) 2-D Mesh, k=4




(b) 2-D Torus, k=4




(c) Binary Hypercube, d=4

Figure 1.3: Three 16-node networks. (a) 2-d mesh. (b) 2-d torus. (c) 4-d hypercube.

When the edge links are wrapped around, the maximum displacement is halved, so:

$$Diameter_{wrapped} = d \cdot \left\lfloor \frac{k}{2} \right\rfloor \tag{1.2}$$

and, for a torus with $N$ nodes:

$$Diameter_{torus} = 2 \left\lfloor \frac{\sqrt{N}}{2} \right\rfloor \approx \sqrt{N}$$

Because the worst-case distances that messages must travel are determined by the diameter of the network, it is an important factor to consider. Notice that the hypercube's $\log_2 N$ diameter is, for $N > 16$, less than the mesh and torus networks' $2(\sqrt{N} - 1)$ and $\sqrt{N}$ diameters.

The *bisection bandwidth* of a network gives an idea of the network's ability to move data from one side of the network to the other. The bisection bandwidth is defined as the amount of data carried by a minimum number of wires crossing a bisection of the network. Since each wire carries one bit of data per unit time, this can be thought of as finding the minimum number of wires which must be cut in order to divide the network into two halves with equal numbers of nodes. In order to compute the bisection bandwidth, the width of the network channels must be known. Assuming that the channels are implemented as bi-directional busses of $w$ wires,[2] the bisection bandwidth is:

$$B_{open} = w \cdot k^{d-1} \text{ bits/unit time}$$

$$B_{wrapped} = 2w \cdot k^{d-1} \text{ bits/unit time}$$

In terms of $N$:

$$B_{mesh} = w\sqrt{N} \text{ bits/unit time}$$

$$B_{torus} = 2w\sqrt{N} \text{ bits/unit time}$$

$$B_{hyp} = w\frac{N}{2} \text{ bits/unit time}$$

Notice that the hypercube has, for the same width data channels, a much higher bisection bandwidth than the mesh and torus networks.

---

[2] If the channels are implemented as uni-directional busses of $w$ wires, each of the bisection bandwidth figures is multiplied by a factor of two.

The next two metrics, average distance and maximum injection rate, depend on the nature of the traffic in the network. *Random* traffic, where sources and destinations are selected from uniform random distributions over the entire network, is a commonly analyzed pattern, so the metrics are given in terms of this traffic.

The average distance, $D$, from source to destination, simply gives the number of network hops an average message must take in its path from source to destination. When traffic is random, the average delays for messages can be computed based on the average distance traveled in the network. In general, $D$ is given by [Ngai 89]:

$$\begin{aligned} D_{open} &= \frac{d}{3}\left(k - \frac{1}{k}\right) \\ &\approx \frac{d}{3}k \end{aligned}$$

$$D_{wrapped} = \frac{d}{4}k$$

In terms of $N$:

$$D_{mesh} \approx \frac{2}{3}\sqrt{N} \tag{1.3}$$

$$D_{torus} = \frac{1}{2}\sqrt{N} \tag{1.4}$$

$$D_{hyp} = \frac{1}{2}\log_2 N \tag{1.5}$$

Once again, the hypercube network has a significantly smaller average distance than the mesh or torus networks.

Because the bisection bandwidth constrains the amount of data which can be moved across a bisection of the network, it also constrains how much data can be injected into the network without overloading the bisection channels. This constraint is also dependent on the network traffic, so it will be formulated in terms of random traffic.

Under random traffic, each message has a probability of one-half that it will cross the bisection of the network. To avoid overloading the bisection channels, nodes cannot inject more data than the bisection channels can handle. Thus, the total amount of data injected into the network per unit time can not be greater than twice the bisection bandwidth. Dividing the total equally among all of the nodes gives:

$$MaxInj = 2\frac{B}{N} \text{ bits/unit time} \tag{1.6}$$

For the three networks of interest, with bi-directional channels:

$$MaxInj_{mesh} = \frac{2w}{\sqrt{N}} \text{ bits/unit time} \tag{1.7}$$

$$MaxInj_{torus} = \frac{4w}{\sqrt{N}} \text{ bits/unit time} \tag{1.8}$$

$$MaxInj_{hyp} = w \text{ bits/unit time} \tag{1.9}$$

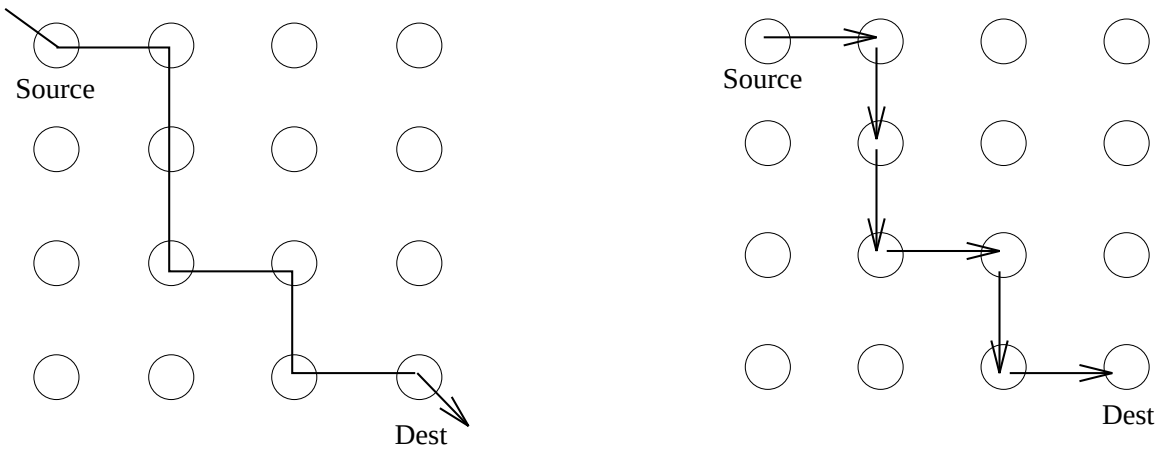
Notice that the hypercube network's injection rate is constrained only by the channel width, $w$, regardless of the network size. Also, the torus, having twice the bisection bandwidth, has twice the maximum injection rate. If the channels are uni-directional instead of bi-directional, but still have width $w$ bits per channel, the injection rates are doubled since the bisection bandwidth is doubled as well.

### *1.3 Flow Control Techniques*

Flow control concerns the method in which messages are transmitted and buffered within a network. Specifically, the method of flow control in a network describes how messages are broken up into individual bits to be transmitted and how these bits are assigned to buffers and channels in the network. There are many differing forms of flow control present in network routers, so only the most basic forms are described here.

In its simplest form, network communication can be accomplished in two basic manners: *circuit-switched* and *packet-switched* communication. In circuit-switched routing, the entire path from source to destination is acquired first, then communication proceeds. The path is relinquished only after the message has been completely transmitted. The advantages of circuit-switching include simplicity of transmission once the path is set up, high utilization of the bandwidth during transmission, and low overhead for routing information. The drawbacks include wasted bandwidth during the acquiring/relinquishing phases, the need to acquire the *entire* path before transmission, and the inability for paths to change during a message transmission. An example of circuit-switched communication is shown in Figure 1.4a.

On the other hand, in packet-switched store-and-forward communication, messages are broken up into fixed size *packets*, each independently routed towards its




(a) Circuit-switched

(b) Packet-switched

Figure 1.4: Basic flow-control methods. (a) Circuit-switched. (b) Packet-switched.

destination. Each link in the path (a *hop*) is acquired only when needed and relinquished when the packet has traversed that link. The packet is stored in any intermediate nodes and then re-transmitted to the next node in its path from source to destination. Store-and-forward communication has the advantage of only requiring link resources when they are actually in use and allowing changes in paths during transmission if necessary. The primary disadvantage is the need to receive and retransmit packets at each hop. Other disadvantages are the overhead for packetization and reassembly, the additional overhead needed to replicate routing information in each packet, and the buffer space needed to buffer the packets at each intermediate node. Figure 1.4b shows an example of packet-switched communication.

Both circuit-switched and store-and-forward communication have variants which incorporate the advantages of the other. Circuit-switched communication has evolved into *wormhole* routing with *virtual channels*. Wormhole routing is an extension of circuit-switching which allows incremental claiming and relinquishing of links and buffers in a network [Dally 87]. Messages are transmitted immediately behind the header of the message as the path in the network is established, blocking if necessary. The path is destroyed immediately behind the tail of the message. Thus, only the resources of the network presently either transmitting or buffering messages are reserved at any given time. Virtual channels add buffering to wormhole routing so that messages may temporarily relinquish ownership of links due to blocking, priorities, or

other reasons and allow non-blocked or higher priority messages to pass them [Dally 92].

Store-and-forward communication has been augmented by *virtual cut-through*, which relaxes the requirement that an entire packet be stored in a node before it is forwarded towards its destination [Kermani & Kleinrock 79]. In a virtual cut-through router, arriving packets which have a desired outgoing link available may proceed as soon as the header arrives without having to wait for the entire packet to arrive. If no desired outgoing links are immediately available, the packet is stored in the node until a link becomes available.

The distinctions between virtual cut-through and wormhole routing with virtual channels are less clear than between store-and-forward and circuit switched communication. Aside from implementation details, the primary differences are that cut-through requires packetization and the accompanying overhead, as well as enough storage at each node to hold the entire packet when it becomes blocked. Virtual cut-through, however, retains the advantages of using fixed-size packets which include a wider variety of deadlock-avoidance mechanisms and independent routing of individual packets. In the following sections, most of the path-selection algorithms work equally well with either virtual cut-through or wormhole routing. The primary exception to this is that the packet-exchange method of deadlock avoidance requires some form of packet-switched communication.

### *1.4 Network Routing*

Routing in an interconnection network is concerned with determining a path for a message to travel in getting from a source node to a destination node. In general, the route a message takes can be described by specifying an ordered list of nodes, $(n_{source}, n_2, n_3, \ldots, n_{dest})$, which describe a path starting at the source node and ending at the destination node. This path may be selected "off-line" by a mechanism which has information on the network, or "on-line" by a distributed mechanism within the network. This dissertation will focus on distributed techniques.

Routing techniques differ in how they respond to congestion and faults within a network, as well as the degree of flexibility in choosing a path that is allowed within "normal" operation. However, all routing algorithms must prevent *deadlock, livelock* and *starvation.*

Deadlock occurs when all of the members of a set of two or more messages require some resource held by another member of the deadlocked set. No message in the set can advance until at least one message relinquishes resources it has already been granted. Rather than detecting deadlock and "fixing" it, most routing algorithms constrain the routing of messages so that deadlock cannot occur. Livelock is a situation where messages are continually moving within a network, but never arriving at their destinations. Once again, most routing algorithms prevent livelock rather than fixing it once it has been detected. Finally, starvation is the condition where some node in the network is continually denied access to a resource it requires. Starvation is very dependent on network traffic and is rarely directly dealt with in routing algorithms.

### *1.4.1 Oblivious Routing*

In an *oblivious* router, the path taken by a message is determined solely by its source and destination addresses, without regard to the current state of the network. Thus, all messages which have the same source and destination will follow the same predetermined path through the network. This path is usually chosen to be a topologically shortest path in the network. Some oblivious routers may allow reconfiguration of the routing algorithm at startup time to account for faulty links in the network, but this is not allowed during normal operation.

#### *Oblivious Path Selection*

Oblivious routers have gained wide acceptance because they provide reasonable service with great simplicity. For common networks, such as those in the $k$-ary $n$-cube family, topologically shortest paths are found easily. A shortest path for a message in such a network is a path in which each hop takes the message closer to its destination in exactly one dimension. Several shortest paths may exist because the ordering of the hops with respect to dimension does not effect the outcome; the path remains a topologically shortest path regardless of the order in which the dimensions are traversed.

Since an oblivious router must choose only one of these shortest paths, it is necessary to further restrict the path-selection algorithm so that only one path is chosen. Dimension order routing is a shortest-path algorithm which imposes an ordering on

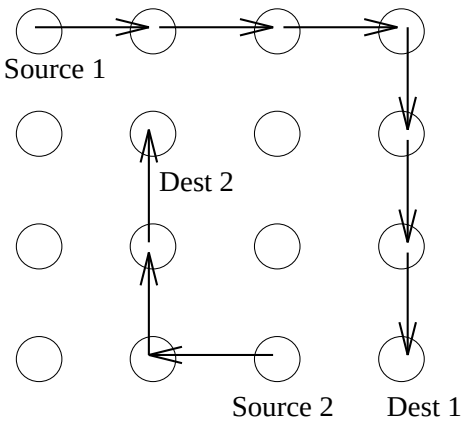



Figure 1.5: Dimension order (X, then Y) oblivious routing.

the dimensions of the network and mandates that a message traverse the network in dimension order. A message will not traverse a link in dimension $d$ until all dimensions earlier than $d$ in the order have displacements of zero. This provides a simple distributed incremental path selection algorithm which can be implemented by finding the lowest-order dimension with a non-zero displacement and routing the message along the link which reduces this displacement. When all displacements are zero, the message has arrived at its destination. Figure 1.5 shows an example of a path taken in a two dimensional mesh using dimension order routing.

*Oblivious Deadlock Prevention*

Dimension order routing can also be shown to be deadlock-free for open-ended $k$-ary $n$-cubes (those without "wrap-around" edges) using simple resource-ordering arguments [Dally 87]. Deadlock cannot occur within a dimension because the open-ended network does not allow cycles within a dimension. Since messages only change the dimension being traversed in a pre-determined order, no cycles can exist between dimensions, either. Assuming that all messages which arrive at their destination are eventually removed from the network, the lack of cyclic dependencies guarantees freedom from deadlock. In order to provide deadlock-freedom for dimension order routing in networks with wrap-around links (such as torus networks), additional mechanisms must be provided to alleviate deadlock within a single dimension. Dally and Seitz provide a solution to this using virtual channels which employs extra buffering to remove the possibility of cyclic dependencies within a single dimension

[Dally & Seitz 87]. This mechanism perturbs the network's balance, though, resulting in non-uniform traffic flow through the network and degraded performance [Bolding 92, Adve & Vernon 93].

*Congestion Avoidance*

Since oblivious routers by nature ignore congestion in the network, it might be expected that they do not perform well under highly congested loads. In fact, Borodin and Hopcroft have shown that, for any $N$ node, $d$ degree network, oblivious routing requires $\sqrt{N}/d^{3/2}$ time steps to route some permutations [Borodin & Hopcroft 85].

Valiant and Brebner [Valiant & Brebner 81] worked around this bottleneck by introducing a randomized oblivious scheme for the hypercube – messages are first sent to a randomly chosen intermediate node and then forwarded to their destination. Although this removes the theoretical worst-case bottleneck, it does so at the cost of doubling the average path lengths of messages in the network.

For certain traffic patterns, though, oblivious routing performs very well. Nearest-neighbor communication patterns produce very little congestion and are served very well by oblivious routing. Uniform random traffic in networks with no "center" nodes (*i.e.* tori and hypercubes) produces uniform random congestion for which oblivious routing works well. On the other hand, networks with "center" nodes, such as open-ended meshes, have more congestion in the center. Here, one might expect oblivious routing to perform poorly. For dimension-order oblivious routing, though, the scheme spreads the traffic out and helps reduce the effects of the central hot spot, producing very good results [Pertel 92a].

For non-uniform traffic, though, oblivious routers do not perform as well. In the presence of hot spots, some nodes in the network will become much more congested than others. However, since oblivious routers are unaware of such hot spots, they will be chosen for routing paths even when equal-length paths with no congestion exist as alternatives. Since many software applications create hot spots or other non-uniform traffic [Pfister & Norton 85], this is a serious drawback to oblivious routing.

The performance of oblivious routing is analyzed in more detail in Chapter 3 and explored through simulations in Chapter 4.

*Buffering*

Buffering can be applied to oblivious routing in many ways. Two basic types of buffering are available: blocking and non-blocking. Blocking buffering is simply in-place buffering of messages. In blocking buffering, messages do not give up the resources they hold when they cannot move in the network. The most common way to add simple blocking buffering is to add FIFO's at any desired point in any path in the network. Non-blocking buffering adds storage to the router that is off of the main data paths. In non-blocking buffering, messages that are buffered do not consume critical resources and can be bypassed by other messages which have resources available to them. Non-blocking buffering can be added by a shared buffer pool for virtual cut-through routing or by virtual channels with FIFO's for wormhole routing [Dally 92].

Adding blocking buffering to a network by inserting FIFO queues between links can help performance by allowing packets to relinquish control of links to other messages earlier. Dias and Jump showed that, for delta multistage networks, adding just a few buffers between stages could double the throughput of the network [Dias & Jump 81a]. However, adding buffers also increases latency by adding additional complexity to the routing path. Dally examined non-blocking buffering by adding virtual channels to wormhole routing and found significant performance advantages for a range of networks, even when compared to networks with equal amounts of blocking buffering [Dally 92]. In general, adding a moderate amount of buffering to an oblivious router can significantly expand its throughput capacity with only a small amount of additional latency per message [Konstantinidou 91].

### *1.4.2 Minimal Adaptive Routing*

Since most networks provide multiple shortest paths between many, if not most, source-destination pairs, it is intuitive to design a router that allows flexibility in choosing among these paths. Routers which allow some choice among minimal-length paths based on local or temporal conditions are known as *minimal adaptive* routers.

In the "basic" minimal adaptive router, the router computes the set of all outgoing links which lie on some minimal path for an incoming packet. These are known as *profitable* links. Of the profitable links, some may be currently unavailable due to traffic or faults in the network; these are removed from the set. The packet is then

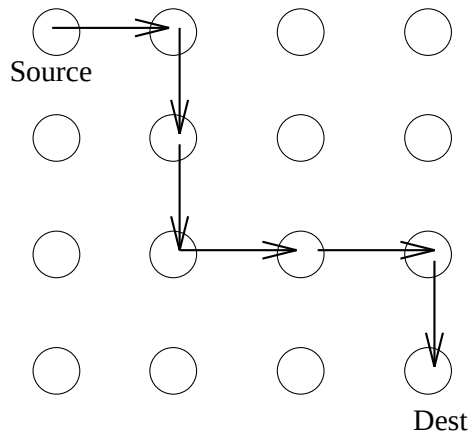



Figure 1.6: Minimal adaptive routing.

sent out on one of the remaining profitable links. If none is available, the packet waits for one to become available, in either a blocking or non-blocking manner depending on the router. Figure 1.6 shows an example of a path a message in a minimal adaptive router might take.

When more than one profitable link is available, the router must somehow choose from the alternatives. There are several alternative methods: randomly, dimension-order, and choice-preserving. The first two methods are obvious in implementation. Choice-preserving attempts to maximize the number of minimal paths still available at any time by choosing the dimension in which the packet has the largest displacement from its destination. Although this helps to preserve the largest set of profitable hops as the packet progresses towards its destination, its usefulness is marginal at best [Ngai 89].

A problem which arises in minimal adaptive routing is deadlock. The constraints which prevented deadlock in dimension-order routing are no longer present in minimal adaptive routing, so other techniques must be added to ensure freedom from deadlock. The Zenith router [Konstantinidou 90] and planar-adaptive routing [Chien & Kim 92] require extra buffering as well as extra constraints on possible paths for packets in order to prevent deadlock. Other routers [Pifarré et al. 91, Cypher & Gravano 92] are *fully minimally* adaptive, in that all minimal paths are allowed, although some amount of extra buffering is required. However, since these algorithms require differing buffer capacities for different nodes in the network, non-uniformities which disturb the network's performance are likely to result.

A problem with minimal adaptive routers is that, in general, the number of paths available decreases as the distance to the destination decreases. Packets which have a non-zero displacement in only one dimension have only one path to choose from and can no longer avoid congestion or faults. This problem becomes worse as packets near their destinations and lose their ability to maneuver around congestion.

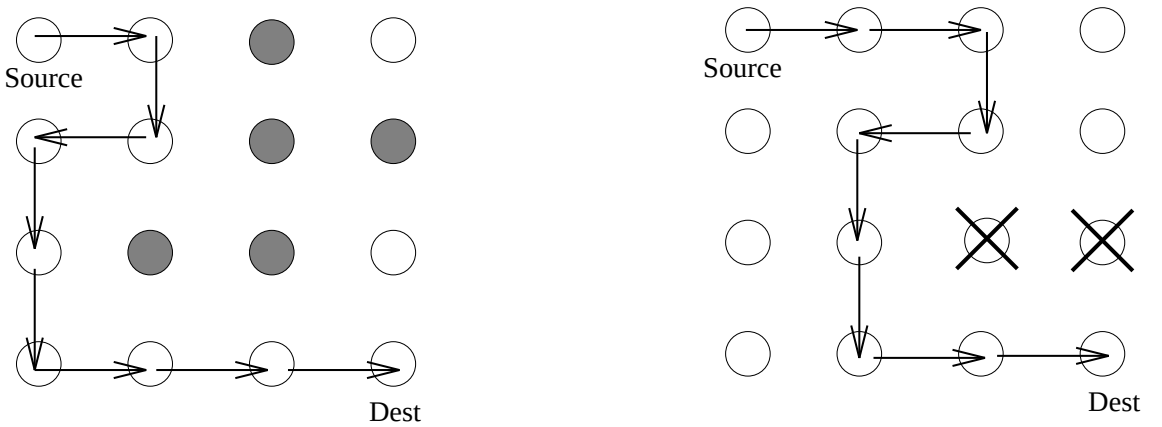
### *1.4.3 Non-minimal Adaptive Routing*

While oblivious and minimal adaptive routers require a packet always be routed on some path of minimal length, non-minimal adaptive routers allow packets to be routed on *any* path from source to destination, of any length. There are two primary motivating factors for allowing packets to follow longer than necessary paths: congestion avoidance and fault-tolerance.

If all minimal paths for a packet are heavily congested, but longer paths are uncongested, it may be possible for a packet to travel out of its way on a longer path but still arrive more quickly. Consider a traffic analogy: although the shortest path between two ends of a city may be a straight line using busy streets, it may be faster to travel a longer distance by detouring onto less congested side streets and arrive much sooner. Also, when packets detour around a congested area, the packet avoids adding to the congestion and making a bad situation worse.

If hardware faults result in a link becoming permanently unavailable, there will be some pairs of nodes in networks using minimal routers which can no longer communicate. By allowing non-minimal paths, these nodes can be avoided and communication is not lost. Figure 1.7 shows the path taken by a message in a non-minimal adaptive network. In (a), the darkened nodes are especially congested, and the packet is routed around the congestion. In (b), faulty routers (indicated by the crossed-out nodes) present a barrier which the routing algorithm is able to detour around.

Non-minimal routers must face different problems in deadlock prevention and deal with prevention of *livelock*. Solutions to these are presented below. Two performance issues also arise: the time lost in taking potentially longer paths and delay in making routing decisions due to the greater complexity of the decision. These issues are discussed in Section 2.4.




(a) Avoiding congested nodes

(b) Detouring around faulty nodes

Figure 1.7: Non-minimal adaptive routing. (a) Routing around congestion. (b) Detouring around a faulty node.

*Derouting*

The decision to route a packet on an outgoing link *away* from its destination is known as *derouting* or *misrouting*.[3] Of the routers presented in the literature, there are many differing methods of deciding when to deroute packets and choosing which packet to deroute. However, two basic classes of non-minimal adaptive routers can be distinguished based on their buffering characteristics: *deflection* routers and *queueing* routers.

Deflection routers [Smith 81, Maxemchuk 89, Fang & Szymanski 91, Smitley 89], also known as "hot-potato" or desperation routers, use a synchronous approach and a time step long enough to transmit an entire packet. At each step the incoming packets are paired with outgoing channels and are transmitted in the next step. Since deflection routers use a synchronous approach, it is guaranteed that all outgoing links will be available for use on the next routing cycle – the routing logic must attempt to find the best mapping of packets to outgoing links. In some cases, though, two or more packets may desire the same outgoing link. One packet will be given its desired link, while the other packets are assigned one of the remaining links. Thus, the "losing" packets are derouted immediately, without being buffered in the node.

[3] In some networks, there may be links available which take a packet neither closer nor further away from its destination. These "no-progress" hops will be labelled deroutes as well.

Queueing non-minimal routers [Ngai & Seitz 89, Konstantinidou & Snyder 90, Coates et al. 93] differ from deflection routers by the presence of a central buffer which holds packets awaiting free outgoing links. In general, packets move into a central buffer from incoming links and wait there until a preferred outgoing link (a link on a minimal path) becomes available. If the central buffer becomes full, some packet is selected to be derouted on the next free outgoing link. Router designs may allow for packets to bypass the queue altogether if an outgoing link is immediately available (cut-through), and there may be FIFO buffering on the input and output channels. The selection of which packet to deroute also depends on the implementation.

*Deadlock Prevention*

In fully adaptive non-minimal routers, any packet has the possibility of going out any free channel from its current location.[4] As a result, it is simple to show that deadlock cannot occur through path dependencies because there are no explicit paths. However, mechanisms to guarantee that the links themselves do not deadlock are still necessary.

Deflection routers achieve this trivially: there are no buffers and all network links are always free at the beginning of each routing cycle. All packets which enter a node during a routing cycle will exit during the next routing cycle, so all packets are always moving about.

Queueing routers usually rely on the packet-exchange protocol [Ngai & Seitz 89] which ensures that each bi-directional link in the network will not deadlock. Essentially, the protocol mandates that two nodes connected by a bi-directional link and having packets to send to each other must accept the other node's packet. In other words, node $a$ cannot send a packet to node $b$ while at the same time denying $b$'s request to send a packet to $a$.

Other methods of deadlock prevention based on restricted resource claiming have also been presented [Coates et al. 93].

---

[4] In some livelock protection schemes, there may be some packets which cannot be derouted, but, as a general rule, this principle applies.

*Livelock Prevention*

A distinct problem which plagues non-minimal adaptive routers is the possibility of packets having infinite-length paths. If there is no bound to the number of times a packet may be derouted it is conceptually possible that a packet never makes progress towards its destination, even though it is continually moving. This is known as *livelock*. There are three basic methods of dealing with livelock: "doing nothing," priorities, and randomness.

One method of dealing with livelock is to ignore it. The argument is that livelock is rare and pathological, so its likelihood is not great enough to worry about. Although this seems naive, many routers have been proposed without protection from livelock. Since livelock-causing traffic patterns exist, no matter how pathological, this is an unattractive solution.

There are two basic variations of the priority method, timestamps and battle scars. Timestamp protocols [Ngai 89] require that each packet be stamped with the time it was injected into the network. Whenever a router must choose a packet to deroute, it ensures that the oldest packet is not chosen. Thus, the oldest packet in the network is guaranteed to be routed on a minimal path to its destination; when it is delivered, there is a new "oldest" packet which is not derouted. Thus, infinite livelock is prevented since all packets will eventually be delivered or become the oldest packet and be guaranteed delivery.

Battle scar methods [Smith 81] place information in each packet which indicates how many times it has been derouted. When choosing a packet to deroute, the one with the smallest battle scar is selected – thus, packets which have been previously derouted are less likely to be continually derouted. Once a packet reaches some pre-determined maximum battle scar, it is routed on an Eulerian path through the network toward its destination which is guaranteed to be livelock-free.

Priority methods suffer from two problems: complexity and overhead. The routing decision at each node becomes more complicated due to the necessity of comparing priorities when derouting. Also, the timestamp or battle scar must be transmitted in the header of the packet consuming bandwidth that could provide other crucial routing information.

The third solution is randomization. Chaotic routers [Konstantinidou & Snyder 90] choose randomly among queued packets when selecting a packet to deroute. By

introducing randomness into the network in this manner, the very regular cycles typical of livelock decay with time. Livelock-freedom is provided only in a probabilistic sense: the probability a packet is in the network for a time greater than $t$ goes to zero as $t$ approaches infinity. However, as shown in Section 2.2.2, the probability of livelock goes to nearly zero within a reasonable time for common networks. Since the likelihood of livelock is low to start with, this provides a practical, though not deterministic, solution.

## *1.5 Summary*

In order to build high-performance multicomputers, a high-performance interconnection network is needed. Simple point-to-point networks suffice for this purpose. In order to provide uniform access to the network, most interconnection networks are built out of regularly-structured components. The most common networks are the the mesh, torus, and hypercube networks, all in the $k$-ary $d$-cube family.

Several methods of flow control exist, from circuit-switched communication to store-and-forward communication. The most promising flow control solutions involve hybrids such as virtual cut-through and wormhole routing.

The selection of a path for a message to take from its source to destination, routing, has been the subject of much study. There are many schemes, ranging from simple oblivious schemes to complicated adaptive schemes. Each of these methods has its relative merits and drawbacks. The remainder of this dissertation will concentrate on comparing routing schemes and building the case for the use of chaotic adaptive routing.

# Chapter 2

# GENERALIZED CHAOTIC ROUTING

Chaotic routing is a non-minimal adaptive routing technique that achieves high performance at a reasonable cost. By exploiting randomization in the prevention of livelock, chaotic routing can be implemented by a relatively simple design. Since the original chaotic routing algorithm is presented only for hypercubic networks, the theoretical justification for the algorithm must be extended to include other networks. Also, the original algorithm does not allow virtual cut-through techniques at a reasonable cost, causing a large performance penalty. This problem is resolved by the presentation of a new cut-through design. Finally, the new cut-through chaos router is compared with other competing router designs and shown to be competitive.

## *2.1 Previous Work*

Chaotic routing was developed by Konstantinidou and Snyder as an efficient alternative to priority non-minimal adaptive routers [Konstantinidou & Snyder 90, Konstantinidou 91]. The chaos router presented there belongs to the queueing non-minimal adaptive router class and is specific to hypercubic networks. Routing is accomplished in a store-and-forward manner and, thus, requires packetization of messages.

### *2.1.1 Conceptual Chaotic Routing*

The basic structure of a conceptual $d$-dimensional hypercube chaos router is shown in Figure 2.1. In this figure packets arrive on one of the $d$ input channels and are stored in one of the $d$ *input frames*, each capable of holding one packet. Packets are then moved into the multiqueue when space is available unless they are at their destination, in which case they are moved into the ejection frame for delivery. Packets are moved out of the multiqueue by a protocol described below into output frames, which buffer one packet each, and then finally to physical output channels.

The central component of the chaos router is the multiqueue. This is a fixed-

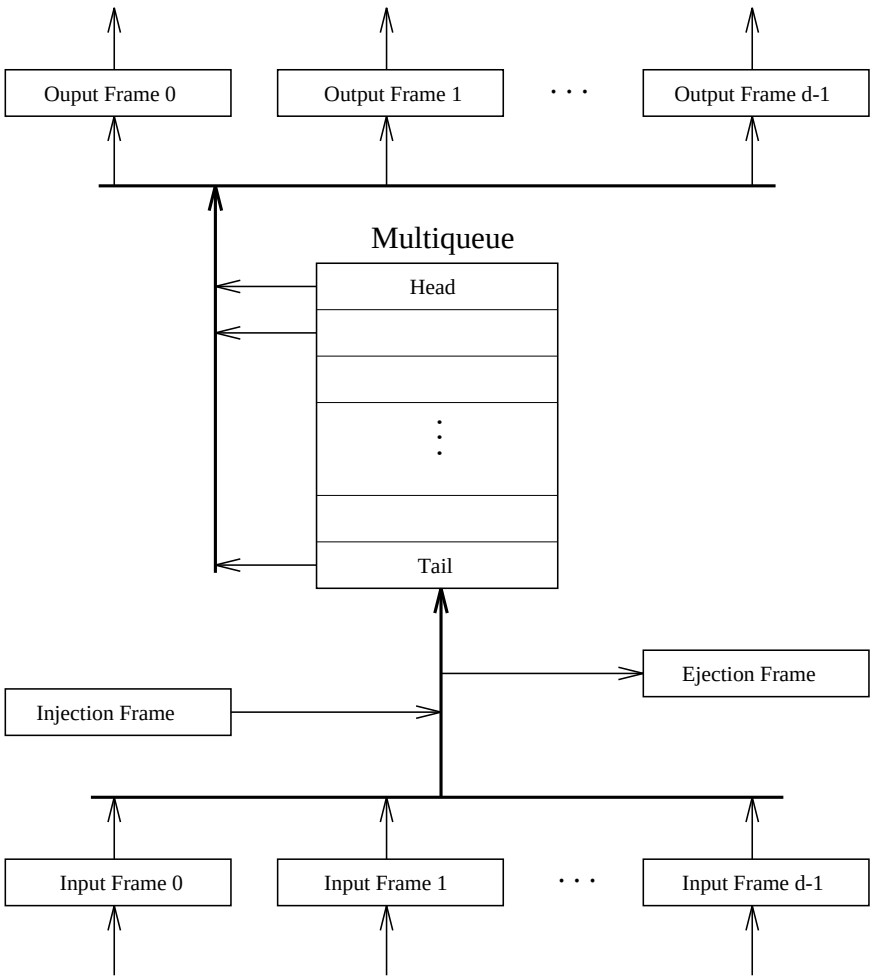



Figure 2.1: Conceptual hypercube chaos router.

size buffer which holds some small number of packets. When an output frame that is profitable to some packet in the multiqueue becomes available, the oldest[1] such packet in the multiqueue is sent to that output frame. The only exception to this occurs when the multiqueue is entirely full. In this case, when an output frame becomes available, a packet is *randomly* selected from among those in the multiqueue to be sent to this output frame, regardless of the packet's intended destination. This is the mechanism by which packets are *derouted* and sent on non-minimal paths.

To prevent deadlock, a version of the packet-exchange protocol is implemented [Ngai 89]. This protocol is enforced by ensuring that any time a packet is *sent* from the multiqueue to an output frame for dimension $i$, if a packet is in the input frame for dimension $i$, it must be read into the multiqueue. Since the outgoing packet has just vacated a spot in the multiqueue, it is guaranteed that there will be room in the multiqueue for the incoming packet.

Livelock, the condition of continually derouting packets so that they always move, but never reach their destinations, is prevented by the random nature of the derouting decision. Because the selection of a packet to deroute is non-deterministic, the probability that a single packet will be continually derouted diminishes to zero with time [Konstantinidou & Snyder 90]. Konstantinidou presents a theoretical analysis which demonstrates that probabilistic livelock freedom is guaranteed on hypercube networks [Konstantinidou 91].

### *2.1.2 Chaos Router Architecture*

To facilitate a practical design for a chaos router, two architectural changes were added to the conceptual design [Konstantinidou & Snyder 91]. The first is the addition of virtual cut-through to the router's design. This was accomplished by the addition of a crossbar which connects each of the input frames to all of the output frames directly, so that a packet which has an available profitable output frame may be immediately sent to that output frame, bypassing the multiqueue. In order to ensure deadlock freedom in the cut-through model, *d deroute frames*, similar to input and output frames, were added to the design [Konstantinidou 91].

The second major design change was the addition of a constraint that only one

[1] Oldest in this case is relative only to the order of arrival into the multiqueue, not to the order of injection into the network.

new path through the router could be established at a time. This allows a large reduction in the anticipated complexity of the router, while not affecting performance significantly [Konstantinidou & Snyder 91].

The resulting hypercube chaos router was then simulated and compared with oblivious routers and priority non-minimal routers. The results show that chaotic routing was a competitive alternative to both of the other simulated schemes [Konstantinidou 91].

## *2.2 Chaotic Routing in Non-hypercubic Networks*

Since the chaos router designed by Konstantinidou and Snyder is limited to hypercubic networks, but many other network topologies are desirable for many reasons, it is necessary to extend the work to include other topologies as well. Because the proofs of freedom from deadlock and livelock both rely on specific features of the hypercube network [Konstantinidou 91], it is necessary to extend these proofs to include other networks as well. The following sections extend the deadlock- and livelock-freedom proofs to include all finite networks with bi-directional links.

### *2.2.1 Deadlock Freedom*

The hypercube chaos router has been shown to be deadlock free based on the packet-exchange protocol [Konstantinidou 91]. The packet-exchange protocol guarantees that each bi-directional channel of the chaos router will not deadlock; that is, there will always be at least one free buffer per bi-directional channel. Since each channel will always be able to transmit messages, only node-to-node dependencies can cause deadlock. However, for chaotic routing, due to the ability to deroute packets when needed, there are no node-to-node dependencies and, thus, deadlock is avoided.

In the proofs presented in [Konstantinidou 91], there are only two elements which depend on certain topological characteristics. This first is that each channel of a node be serviced within finite time. This requires that the degree of the node be finite. The second is that the channels are bi-directional. This can be satisfied either with a physical bi-directional channel or two associated uni-directional channels which carry data in opposite directions. If all channels are bi-directional, deadlock will not occur and any packets which are derouted along a channel may always follow the channel

back in the opposite direction if needed.

Thus, for any finite network with bi-directional channels, chaotic routing will be deadlock-free.

### *2.2.2 Livelock Freedom*

Deterministic livelock freedom, ensuring that every message is delivered after a given period of time, is not true for chaotic routers. However, probabilistic livelock freedom – the probability that a packet remains undelivered after $t$ seconds goes to zero as $t$ increases – is true. The following proof generalizes the livelock-freedom proof of the hypercube chaos router [Konstantinidou 91] to include all finite-sized networks.

The packet's path through the torus network is described by a sequence of *moves*. The distance of a message from its destination is at most the diameter of the network, $Diameter_{net}$. For $k$-ary $d$-cubes, the diameter is given by Equations 1.1 and 1.2. In most networks, each hop moves a message either closer to or further from its destination. However, some networks may include "no-progress" hops after which a message is still the same distance from its destination. For this proof we will group the "no-progress" hops together with hops which move packets further from their destination as "deroutes." The probability of moving closer is the probability of being routed, $p \geq \epsilon$, which is established in a theorem arguing that a message remains in the multiqueue a bounded amount of time and is thus subjected to only a bounded number of random derouting decisions (see [Konstantinidou & Snyder 90]). The probability of moving further from the destination is $q = 1 - p$.

Define a *game* as a sequence of $Diameter_{net}$ moves. Message $M$ starts game $i$ at distance $a_i$ and finishes at distance $a_{i+1}$. Let $l_i$ denote the event that $M$ was not delivered during game $i$ and $w_i$ the event that $M$ was delivered during game $i$.

Let $Q(i)$ be the probability that message $M$ has not been delivered after $i$ games. Then

$$Q(i) = P(l_i l_{i-1}..l_1) = P(l_i \mid l_{i-1}..l_1) \cdot P(l_{i-1}..l_1)$$

For simplicity, let us substitute $F_k$ for $l_k...l_2 l_1$, $1 \leq k < i$ and let us define $P(l_1 \mid F_0) \equiv P(l_1)$ and $P(w_1 \mid F_0) \equiv P(w_1)$. Then

$$Q(i) = P(l_i \mid F_{i-1}) \cdot P(l_{i-1} \mid F_{i-2}) \cdots P(l_1) \tag{2.1}$$

Clearly, $P(l_j \mid F_{j-1}) = 1 - P(w_j \mid F_{j-1})$, $1 \leq j \leq i$. In the following we will estimate $P(w_j \mid F_{j-1})$. Let $S_{j,k}$ denote the event that message $M$ starts game $j$ at a distance of $k$ hops from its destination. Events $S_{j,k}$ are mutually exclusive and one of them necessarily happens. Thus

$$\begin{aligned} P(w_j \mid F_{j-1}) &= P(w_j S_{j,1} \cup \cdots \cup w_j S_{j,Diameter_{net}} \mid F_{j-1}) \\ &= \sum_{k=1}^{Diameter_{net}} P(w_j S_{j,k} \mid F_{j-1}) \\ &= \sum_{k=1}^{Diameter_{net}} P(w_j \mid S_{j,k} F_{j-1}) \cdot P(S_{j,k} \mid F_{j-1}). \end{aligned}$$

But $P(w_j \mid S_{j,k} F_{j-1}) \geq \epsilon^{Diameter_{net}}$, thus

$$P(w_j \mid F_{j-1}) \geq \epsilon^{Diameter_{net}} \sum_{k=1}^{Diameter_{net}} P(S_{j,k} \mid F_{j-1}).$$

Since $\sum_{k=1}^{Diameter_{net}} P(S_{j,k} \mid F_{j-1}) = 1 \Rightarrow$

$$P(w_j \mid F_{j-1}) \geq \epsilon^{Diameter_{net}} \Rightarrow$$

$$P(l_j \mid F_{j-1}) \leq 1 - \epsilon^{Diameter_{net}} \tag{2.2}$$

Finally, Equations (2.1) and (2.2) $\Rightarrow Q(i) \leq \left(1 - \epsilon^{Diameter_{net}}\right)^i$.

Since $0 \leq \epsilon < 1$, $\left(1 - \epsilon^{Diameter_{net}}\right) < 1$ as long as $Diameter_{net}$ is finite. For any finite-sized network, this is trivially true.

Thus, for any finite-sized network, the probability $Q(i)$ that $M$ will not have been delivered after $i$ games, where $i \to \infty$ is:

$$\lim_{i \to \infty} Q(i) = \lim_{i \to \infty} \left(1 - \epsilon^{Diameter_{net}}\right)^i = 0$$

The probability $P(i)$ that $M$ will be delivered after $i$ games, where $i \to \infty$ is:

$$\lim_{i \to \infty} P(i) = 1.$$

This proof essentially shows that for any network with a finite diameter, the chance of a packet not ever being delivered due to livelock goes to zero as time goes forward. The important features of the proof are that there is always some chance of a packet being routed closer to its destination and that the network's degree is finite. Thus, since a packet is always moving around in the network, and cannot get infinitely far away from its destination, it will, eventually, in the worst case, "bump into" its destination.

Although only probabilistic livelock freedom has been established, a practical routing algorithm should have a reasonable limit on the time a message may spend in the network without being delivered. Timestamp-based protocols [Ngai & Seitz 89] provide a bound based on the guaranteed shortest-path delivery for the single oldest packet in the network. This bound, however, can be very large since all other packets in the network at the time of injection must first be delivered.

For chaotic routing, although there is no definite bound on the value of $i$ for which $P(i) = 1$, the value at which $P(i)$ becomes arbitrarily close to 1 can be determined. Figure 2.2 shows the probability $P(i)$ of a message reaching its destination within $i$ games for a network with $Diameter_{net} = 25$. In this figure, the value of $\epsilon$, the probability of a profitable routing decision at each hop, is varied from 0.80 to 0.95. As can be seen from the graph, the rate at which $P(i) \rightarrow 1$ increases substantially as $\epsilon \rightarrow 1$. Practically, as observed in simulations, the probability of being profitably routed from a chaos router is always above 0.90. Figure 2.3 shows how the probability of delivery changes with the diameter of the network. As the network's diameter increases, the number of hops before the chance of being delivered is more than 95% also increases. Fortunately, is remains reasonable for practical network sizes.

## *2.3 The Cut-through Chaos Router*

Having established the applicability of chaotic routing to almost all finite networks, it is necessary to extend the functionality of the router to enhance its performance. While Konstantinidou and Snyder proposed a deadlock-free cut-through version of the chaos router [Konstantinidou & Snyder 91], it suffered from two design difficulties. First of all, it requires the use of *deroute* frames in order to ensure deadlock-freedom. Since there is a requirement of one deroute frame and the associated data paths per channel, this design requires significantly more area to implement than the "basic"

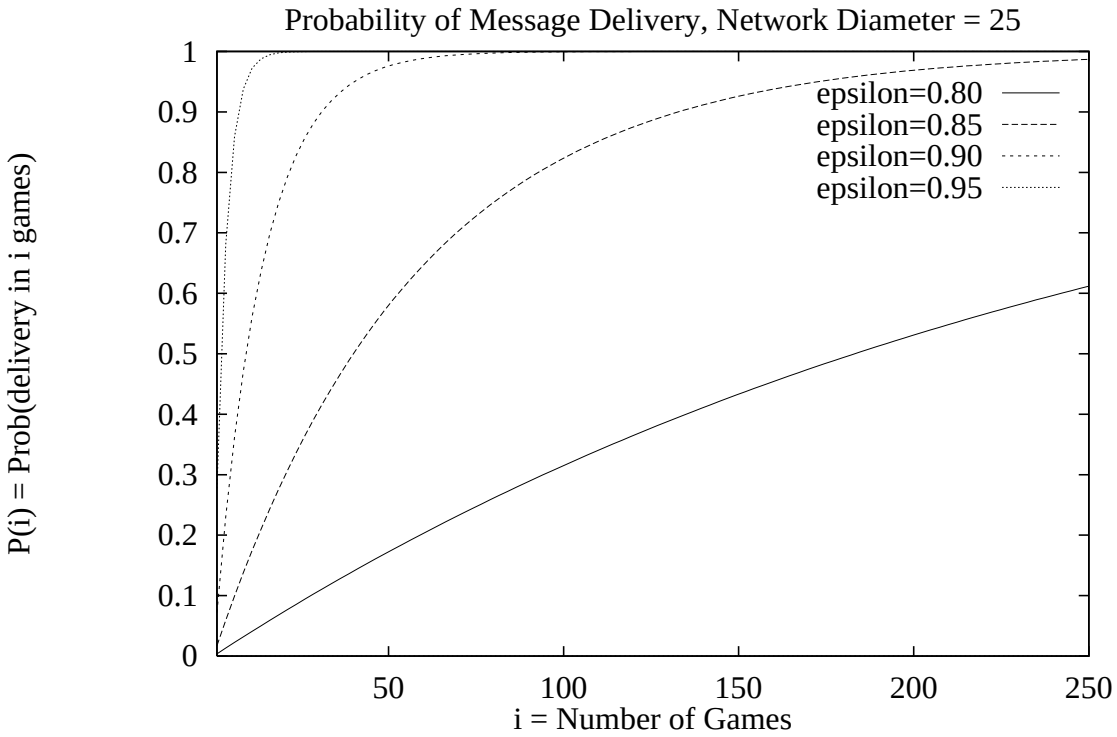



Figure 2.2: Probability of message delivery, $P(i)$, with $Diameter_{net} = 25$.

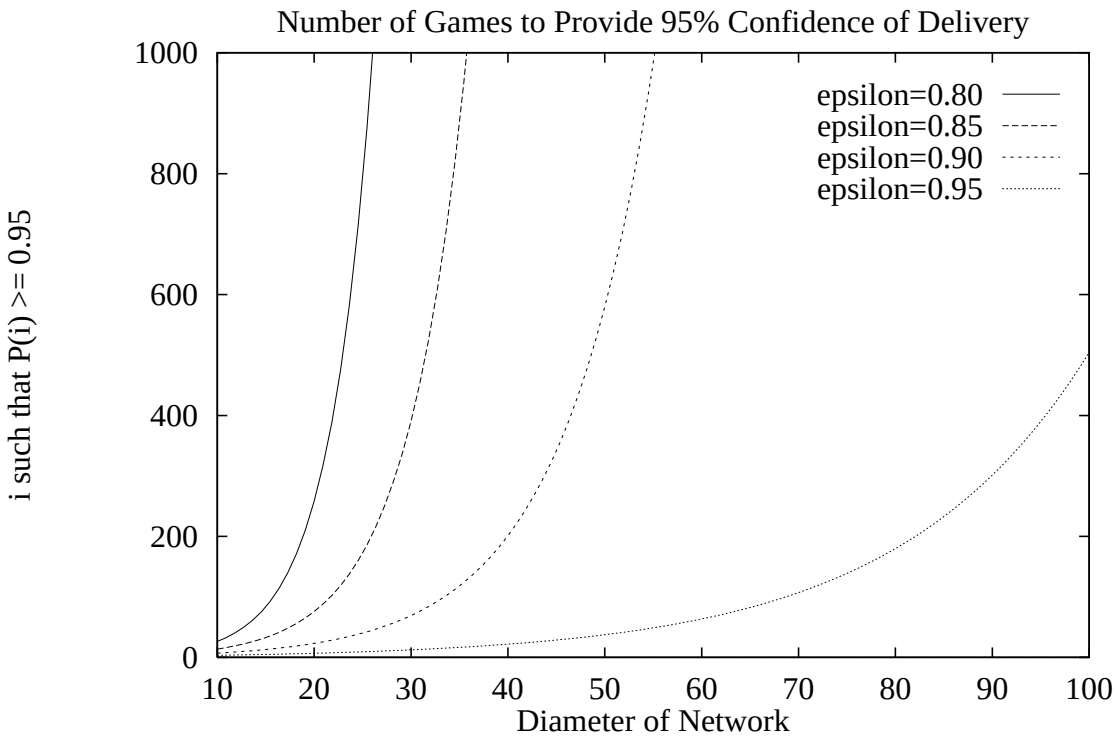



Figure 2.3: Number of games to provide 95% confidence of delivery.

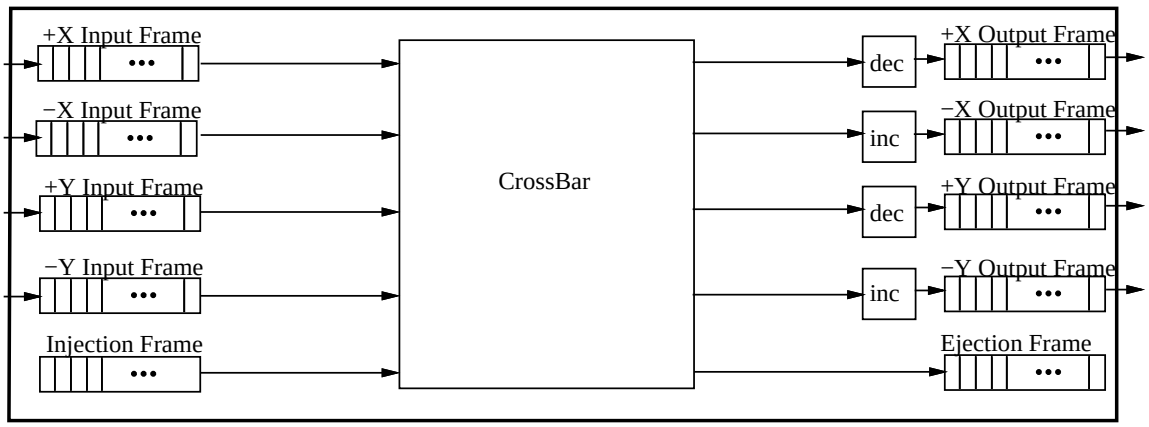



Figure 2.4: A conceptual two-dimensional minimal cut-through router.

router design. The second issue is performance. While packets which cut-through the node proceed with minimal delay, packets which pass through the multiqueue are routed in a store-and-forward manner across shared busses and, thus, incur a substantial latency penalty.

To address these issues, the cut-through chaos router is presented. This router provides the function of virtual cut-through without the requirement of extra buffering. Cut-through is implemented on all data paths, so packets only wait minimal amounts of time when resources are not available. The design is presented for general networks, although the focus will be on two-dimensional (mesh and torus) networks as an example.

Although the cut-through chaos router can be viewed as a conventional chaos router with a queue bypass path added, it is more informative to view it from the perspective of a straightforward minimal adaptive router. In a cut-through minimal adaptive router, as illustrated in Figure 2.4, packets enter the router into input frames, cross the core of the router via a crossbar and exit via output frames after having their headers updated to reflect their progress. In reality, to implement a minimal adaptive router, additional buffering would be needed to ensure deadlock freedom. However, for the purposes of this illustration, these details are not relevant. The cut-through nature of this router allows packets to be assigned paths to and from buffers whenever their headers have arrived. It is not necessary to wait for the entire packet to be buffered before routing assignments are made.

The cut-through minimal adaptive router is made into a cut-through chaos router

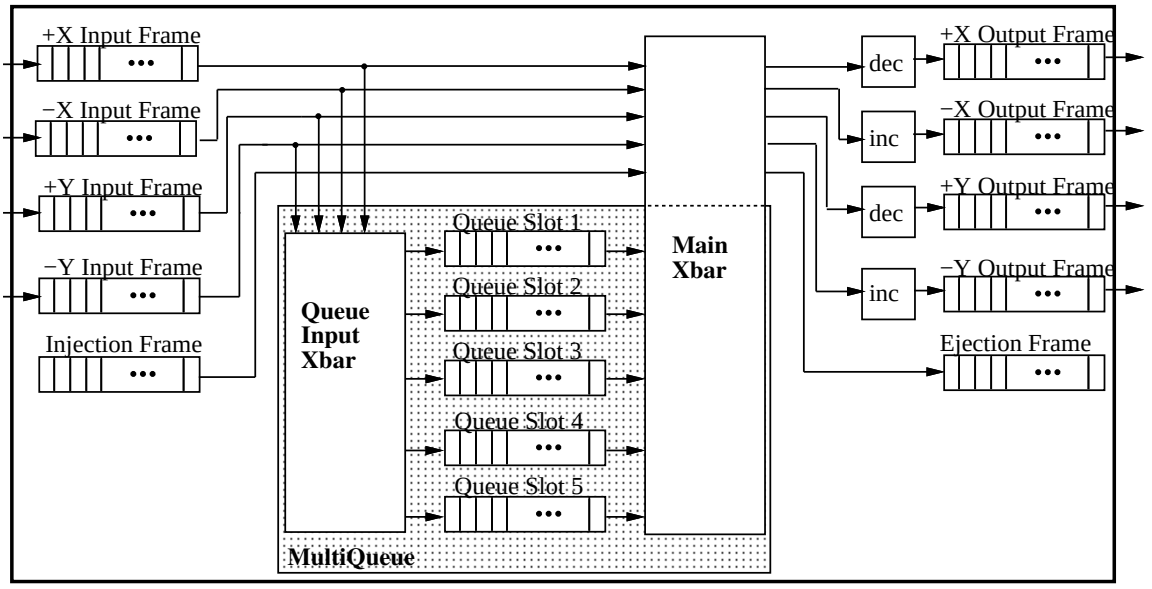



Figure 2.5: A conceptual two-dimensional cut-through chaos router.

by the addition of the multiqueue and associated data paths. As shown in Figure 2.5, all of the data paths of the minimal adaptive router are present in the cut-through chaos router. However, several *queue slots* to hold packets in the multiqueue have been added. To allow packets to get to and from the multiqueue, the main crossbar has been expanded to take inputs from the queue slots, and a second crossbar, the queue input crossbar, has been added. Packets in a cut-through chaos router can flow through any of the components, including the queue slots, in a cut-through fashion. However, the complexity of the control logic for the multiqueue may slow down packets which travel through the multiqueue. On the other hand, packets which directly cut-through via the main crossbar are not delayed any more than in a minimal adaptive router.

In the cut-through chaos router, packets enter the multiqueue if they have *stalled* for a "long" time in an input frame waiting for a profitable output frame to become available. A packet is considered stalled if the entire packet has arrived in the input frame, i.e., its tail has arrived in the input frame, yet its head has not been routed out. Thus, when packets have waited a "long" time for a profitable routing assignment without receiving any, they are moved out of the input frame to the multiqueue so that other packets may use the valuable channel resources.

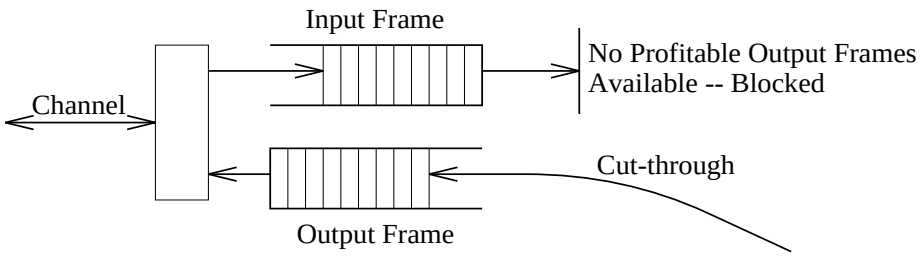



Figure 2.6: Violation of the packet-exchange protocol when using cut-through.

Packets leave the multiqueue in the same manner as in a conventional chaos router, although a full crossbar is provided instead of a bus. The only difference is that there may be competition with packets in input frames which desire the same output frames as packets in the multiqueue. In this case, packets in the multiqueue always have priority.

Unfortunately, the possibility of deadlock is introduced when using this cut-through scheme. The packet-exchange protocol mandates that whenever a packet is sent to output frame $i$, if a packet is in the corresponding input frame $i$, then the second packet must be removed from the input frame. The conventional chaos router achieves this by always following a *send* from the queue with a *read* to the queue from the same dimension. However, with cut-through, there is no guarantee that a destination output frame will be available to move the packet from the input frame when needed. Figure 2.6 illustrates this. A packet has been sent to the output frame for this channel via cut-through. However, the input frame is full, but no profitable output frames are available for the packet in the input frame to be moved to, so it can not cut-through to its destination. Since this violates the packet-exchange protocol, deadlock may occur.

The deadlock problem can be fixed, however, by reading the packet in the input frame into the multiqueue when a packet is sent to the corresponding output frame. Thus, the cut-through chaos router's operation is changed so that whenever a packet is sent to an output frame, either from the multiqueue or via cut-through, if the corresponding input frame has a packet in it, the packet in the input frame *must* be read into the multiqueue. In some cases it may be possible to send the packet from the input frame to a profitable output frame instead of the multiqueue. However, if this is done, the second destination may have a full input frame and require that

it be emptied in the same manner. Since this could cascade throughout each of the dimensions of the router, the multiqueue is always chosen as the destination in such a situation, regardless of the availability of output frames.

The cut-through chaos router provides a low-latency path for packets which have a desired output frame immediately available, while providing the advantages of non-minimal adaptivity by means of the multiqueue. Since most latency-critical packets will travel through the simple cut-through crossbar, the time penalty of the more complex multiqueue operations does not seriously penalize the majority of packets which pass through the router.

## *2.4 Comparisons with Other Routers*

The cut-through chaos router attempts to provide the advantages of adaptive routing while matching the performance of state-of-the-art oblivious routers. Because there are many competing router designs, it is necessary to compare the relative merits of each of them.

Because speed is of such importance in router design, complexity is very costly. The nature of the decisions which must be made by different classes of routers requires designs of varying complexity. At the heart of the design of a router is the "basic routing decision": assigning incoming packets to outgoing channels. This decision can be made with very little information for oblivious routers, but requires more work for adaptive routers.

### *2.4.1 The Oblivious Decision.*

For oblivious routers using dimension order routing, path selection is simple because it is defined in advance. All that is necessary is for the router to see if the header in the current dimension is non-zero: if so, the message "goes straight" along the same dimension, otherwise it "turns" to either the next dimension or the processor channel. It is not necessary to consider more than one dimension at a time or to consider whether buffers are full or links available when making the routing decision.

### *2.4.2 The Adaptive Decision.*

The adaptive routing decision is more complicated than the oblivious decision because the path taken must be computed based on dynamically changing information. Each message may have several profitable channels to choose from, some of which may be unavailable. The adaptive decision involves computing the profitable channels for an incoming message, masking out channels which are unavailable due to contention or faults, and selecting a single outgoing channel from those remaining.

The adaptive decision requires first decoding the header to find which outgoing links are desired. If all of the header information can be encoded into a single flit, the decoding can proceed immediately. Since each dimension can be decoded in parallel, there is no time lost over the oblivious decode step. However, whereas the oblivious router has finished its choice at this point, adaptive routers must continue on.

To make an informed choice among outgoing links, the router must use information on which outgoing links are available. Although this is simple, it requires information which is likely to be widely distributed about the routing chip, and thus causes some delay. More importantly, as incoming messages claim outgoing links, the status changes. Thus, the decision must either be serialized or controlled by more complex logic to deal with contention. Finally, once the profitable, available channels are identified, one must be selected from the set. This can be done simply by either random or ordered choice.

By closely examining the competing routers, several alternatives can be dismissed as inferior. The remaining contenders are then compared in detail.

## *2.5 Oblivious Routers – The State-of-the-art*

Oblivious routers are simple and of mature design – a combination that produces a very fast router which is hard to beat. A good example of this is the Caltech Mesh Routing Chip (MRC) [Flaig 87, Seitz & Su 93]. The MRC is a simple oblivious wormhole routing chip for open-ended two-dimensional mesh networks. Simplicity is the rule: deadlock is prevented by dimension order routing, messages block in place when links are unavailable, and wormhole routing reduces overhead to a minimum. The routing decision for such a router can be broken down into two cascaded stages, each making the simple decision "go straight or turn" for a single dimension.

The Caltech MRC has undergone several generations of design and implementation, resulting in a finely-tuned routing chip. The routing decision is extensively pipelined so that the bandwidth of the routing channels is determined only by the chip-to-chip delay and not affected by the logic internal to the chip. Within the chip, the logic is kept to a minimum due to the constraints of dimension order routing making input-to-output latency very low as well. Thus, the simplicity of oblivious dimension-order routing with minimal buffering results in a high bandwidth, low latency router.

Extensions to this router include simple FIFO buffering, non-blocking buffering using FIFO's and virtual channels [Dally 92], and virtual channels to provide deadlock prevention in torus networks[Dally & Seitz 86]. Each of these enhancements provides potential advantages at the cost of greater complexity.

Detailed analysis and comparisons between oblivious routers and the chaos router are provided in Chapters 3 and 4.

### *2.6 Minimal Adaptive Routers*

Minimal adaptive routers must not only make the adaptive routing decision, but also implement a deadlock-prevention protocol. Although the protocols in the literature vary widely, all rely on multiple classes of buffers or queues in the node to hold messages at different stages in their path from source to destination. The router must compute and select which buffer class a message belongs to before assigning a buffer. This may involve a computation on the destination of the message or on information tagged to the message. Either way, the additional complexity of the decision, as well as the hardware space required for the extra buffers results in longer times to route messages through a node.

Since some minimal adaptive designs restrict which minimal paths may be taken [Konstantinidou 90, Chien & Kim 92], the performance of the router is further diminished relative to fully-adaptive routers. Those algorithms which do allow all minimal paths still create asymmetries in network resource utilization. The deadlock prevention mechanisms, in general, rely on ordered classes of queues to store messages as they travel in the network [Pifarré et al. 91, Cypher & Gravano 92]. Since the transition from one class of queue to a higher class occurs only when a message experiences a certain event, such as "crossing the zero node," messages in different parts of the

router face different sets of circumstances. Similar asymmetries in oblivious deadlock prevention schemes have been shown to seriously degrade performance [Bolding 92, Adve & Vernon 93], so minimal adaptive routers are likely to exhibit degraded performance as well. Because chaotic routing exhibits no asymmetries, i.e. all nodes in the network are indistinguishable with respect to the routing algorithm, the results should not exhibit these problems.

In unpublished simulation studies, Konstantinidou found the Zenith minimal adaptive router to perform worse than non-minimal routers. An intuitive explanation might be: packets in adaptive routers "discover" congestion when all forward paths are blocked. For minimal adaptive routers this is "too late," since the minimum distance constraint forces the packet to wait, adding to the congestion. Nonminimal adaptive routers, however, allow messages to "go backwards" and possibly bypass the congestion.

### *2.7 Deflection Non-minimal Adaptive Routers*

Deflection routers require that all incoming packets arrive at a node during a single routing cycle, and all packets leave on the next routing cycle. This constraint has several consequences on complexity and performance.

Once again, the adaptive routing decision must be made. However, since all packets must be routed at the same time, the decision must be made in parallel for all channels. Essentially, the decision is transformed from "find a free profitable channel for one packet" into "map all incoming packets to outgoing channels in the *best* manner." This decision may be implemented in several ways, from simple greedy algorithms to multiple-pass algorithms with priorities. Although the decision is simplified because all outgoing channels will be free (until assigned), the decision may be very complex. Furthermore, the complexity grows with the node degree, limiting their usefulness in high-degree networks.[2]

The second consequence of the deflection technique is that the headers of all packets must arrive at and leave a node on the same cycle. Thus, all packets must be synchronized together so that their headers are aligned. The result is that neither

[2] The complexity of the deflection router used in the Tera computer resulted in changing the network, a degree 6 3D torus, to a degree 4 network (by removing X direction and Y direction wires in alternating planes) in order to make the routing decision within the 2.5 ns clock cycle.

virtual cut-through nor wormhole routing may be used: store-and-forward is the only applicable method. While this is of no consequence if an entire packet can be transmitted in a single cycle, limits on the bandwidth of channels cause most multicomputer networks to have multi-flit packets which must incur the large latency penalties of pure store-and-forward routing. For larger area networks with high-bandwidth cables or fiber-optic channels and less critical latency requirements, deflection routing may be applied more usefully.

Thus, although deflection routing provides two simplifying benefits: guaranteed availability of outgoing channels and no need to deal with an internal buffering, the complexity of the mapping decision and the inability to use virtual cut-through routing limit its use as a multicomputer router.

## *2.8 Queueing Non-minimal Adaptive Routers*

Queueing non-minimal adaptive routers, including the chaos router, must perform the adaptive routing decision, determine when packets enter and leave the queue, and select which packet to deroute in a manner which prevents livelock.

In a typical queueing router, packets arriving at the node are buffered in a central buffer pool or queue and wait for free profitable channels on a FIFO basis. Typically, this buffer is relatively small, ranging in size from five packets [Bolding & Snyder 92] to nineteen packets [Coates et al. 93] in implemented designs. When the internal queue becomes full and there are no profitable channels available for any packets in the queue, a packet is selected from those in the queue to be derouted to the next available channel, regardless of its destination.[3] The method of selecting which packet to deroute varies among designs, primarily depending on what livelock prevention mechanism is used (see Section 1.4.3). The presence of the internal queueing buffer allows for a fully adaptive router where waiting packets do not block channel resources.

### *2.8.1 Livelock prevention*

The primary distinction between different queueing non-minimal routers is in the method of livelock prevention as described in Section 1.4.3. Priority routers [Ngai &

[3] Packets are not derouted to the processor channel.

Seitz 89] use timestamps, chaotic routers [Konstantinidou & Snyder 90] use randomization, and others do not guard against livelock at all [Coates et al. 93].

The priority router is strictly more complex than a chaotic router because determining which among colliding packets is oldest adds a significant complication to the routing logic. The chaotic router does not need to make such decisions, giving it lower node latency and better overall performance [Konstantinidou 91]. Also, priority adaptive routers require the inclusion of a creation timestamp to each packet, using up bandwidth which could be used for data. Furthermore, the timestamp data must be in the header of a packet since the information is needed before the derouting decision can be made; this may extend the time to transmit the header to more than a single cycle, adding to the node latency.

Thus, within the class of queueing non-minimal routers, chaotic routing provides the lowest complexity livelock-free design at the cost of providing only probabilistic protection. However, since the finite bounded delivery time of deterministic livelock protection provides little benefit due to the enormous size of the bound, which is proportional to the number of packets the network can hold, chaotic routing provides, in a practical sense, the same function at a smaller cost.

### *2.8.2 Queue management*

Management of the central queue is a very complex and difficult task when compared with the simple "go straight or turn" oblivious routing decision. Packets must be allowed to enter the queue in any order, but be removed from the queue in FIFO order by outgoing channels. In other words, when an outgoing channel becomes free, the oldest packet in the queue which can profitably use it must be assigned to it. When the queue fills up, selection of which packet to deroute must take into account the policy for derouting as well. Add to this the need to implement virtual cut-through, which complicates matters because the queue may be filled with some complete packets and some partial packets, and the complexity becomes overwhelming.

Consequently, chaotic routing should perform better than minimal adaptive routing, deflection routing, and priority routing due to simpler design than minimal and priority routers (deadlock and livelock prevention) and the ability to use virtual cut-through switching which deflection routers cannot use. Thus, chaotic routing is the best contender to compare with state-of-the-art oblivious routers.

### *2.9 Fault-tolerance*

In the presence of faulty links or nodes, routing becomes a much more difficult problem. In oblivious routers, since there is only one possible path between a source $a$ and destination $b$, any fault on this path will render direct communication between $a$ and $b$ impossible. If the fault is *static*, that is, known to exist ahead of time, communication may be re-established through an intermediate node $c$ such that neither the path from $a$ to $c$ or the path from $c$ to $b$ relies on the faulty component. This solution requires advance knowledge of the faults in the network, construction of an "intermediary" table (which may be non-trivial in the presence of multiple faults) at each injecting node, and that the packet be received by the intermediate node and re-injected into the network to travel to its destination. Although it would be easy to add a "forward" bit in the header of a packet along with a second destination field so that forwarding could be accomplished entirely within a router, this would violate the dimension-order routing properties necessary to prevent deadlock. Thus, oblivious routing may tolerate static faults, but at a high cost.

When transient faults occur or static faults first appear, oblivious routing has no possibility of routing around failed paths and the only solution is to stop the network, map the faults, and re-start.

In chaotic routing, fault-avoidance is more natural. If a link is not functioning, it will appear to the router as a continually busy link and the router will deroute around it just as if congestion were the problem. Since all routes are available, fault-tolerance is established as long as nodes are not completely isolated by faults. Mechanisms to extend this "natural" fault-tolerance to be more efficient and robust at little cost have been outlined [Bolding & Snyder 91a].

Thus, chaotic routing provides fault-tolerance for both static and some transient faults without requiring additional overhead. On the other hand, oblivious routing can provide fault-tolerance only for already known static faults at a high overhead cost.

### *2.10 Out-of-order Packet Delivery*

Messages which are broken into packets for network transmission must be reconstructed before being delivered to the receiving node. If the packets arrive in the

same order as they are sent, this is trivial, provided there are no lost packets. On the other hand if packets may be lost or arrive in a different order than they were injected, mechanisms must be provided to re-order the packets into the original message.

Oblivious routing always delivers packets in order because there is a single path between a source and destination and packets do not pass each other on the same route.[4] For adaptive routers, this is not true. Since packets may take different paths and face differing levels of congestion, they may arrive out-of-order. Although the router itself does not deal with either packetization or reordering, the cost of using the network is increased because of the extra overhead needed.

The most straightforward solution to reordering is for each injecting node to keep count of the packets it injects to each destination node and place this information within each packet. Each receiving node places the packets it receives in a buffer until it can deliver them in order. Other than the obvious additional overhead at the sending and receiving nodes and the need to transmit the count with the packet, this solution also must deal with the problem of overflow of the counter or buffer space at the receiver. The chaos router actually uses this to its advantage in detecting lost packets for the purpose of fault-tolerance since overflow of the counter indicates either a lost packet or a very out-of-order packet [Bolding & Snyder 91a]. Other methods for dealing with this problem using counting protocols have been suggested [Ladner et al. 92]. Unlike the timestamp used in priority routers, though, the packet count is used only by the source and delivery nodes and is not used during the routing of the packet. Thus, the packet count does not add to the complexity of the routing decision. Nonetheless, the additional overhead needed for packetization and reordering are liabilities which oblivious routing is not concerned with and adaptive routing must face.

## *2.11 Summary*

The hypercube chaos router developed by Konstantinidou and Snyder has been shown to be able to operate on any finite network with bi-directional links. This requires the simple extension of the deadlock- and livelock-freedom proofs to include networks of arbitrary topology.

---

[4] If non-blocking buffering is added to an oblivious router, packets may be able to pass each other and delivery could be out-of-order.

The store-and-forward chaos router has been extended to operate in an efficient virtual cut-through manner without the need for any additional buffering. A modified chaotic routing algorithm is presented that allows both efficient cut-through operation and desirable adaptivity.

The cut-through chaos router is compared with other competing network routing algorithms and designs. It is shown by analysis that chaotic routing should have better performance than minimal adaptive routers, deflection routers, and priority queueing non-minimal routers. The efficient and simple design of oblivious routers, however, remains difficult to match, and thus, is considered the best contender with which to compare the performance of chaos.

# Chapter 3

# DETAILED CHAOS ROUTER DESIGN

Chapter 2 introduced the conceptual chaotic router design. In order to actually simulate and build a chaos router, though, more detail must be given. In this chapter, a design is given with sufficient detail to provide a basis for accurate analysis and simulation. This design is compared with the designs of competing routers to provide a rough comparative performance analysis.

## *3.1 Design Goals*

Two primary design goals govern network router design: low latency and high throughput. For chaotic routing to be viable, it must perform as well or better than competing routers. Presently, most multi-computer networks use simple, but fast, oblivious routers, so the chaos router has been designed to meet or surpass their performance levels. Since the mode of operation of routers using virtual cut-through or wormhole routing differs depending on the amount of congestion in the network, the uncongested and congested cases are considered separately.

### *3.1.1 Uncongested Network Performance*

When a network is uncongested there are few conflicts for resources, so packets should be able to acquire needed buffers and links with little or no delay at all times. Since the network will have much greater capacity to hold packets than needed, there should be no backup of congestion and throughput should be equal to the applied load in all cases. Therefore, in the uncongested case, there will be no apparent difference in throughput between oblivious and chaotic routing.

Since throughput is not a distinguishing factor between chaotic and oblivious routing when congestion is low, the minimum latency for packets going through the network becomes the figure of merit. Since the chance of conflicts in routing is small, packets will rarely enter the multiqueue in chaos routers. Because of this, it is almost

certain that packets will never be derouted, so packets will almost always follow minimal-length paths from their sources to their destinations. In oblivious routers, the paths are always of minimal length, so a given packet will have to take the same number of network hops in either chaotic or oblivious routing when loading is light. Thus, only the differences in routing latency between oblivious and chaotic routing for packets following paths of the same length are of concern.

Although there is a chance of conflict between two or more packets in the lightly-loaded case, only conflict-free routing is considered in the following analysis. Since chaotic routing allows packets to have a greater number of choices during each routing decision, packets will have fewer chances of conflict when using chaotic routing. Thus, the assumption of conflict-free routing is more likely to hold in chaotic routing. Since conflicts can only delay packets, this assumption tends to bias this analysis slightly in favor of oblivious routing.

The time for a packet to travel a minimum-length path from its source to its destination without conflicts is known as the *minimum network latency* for that packet. This figure includes all of the time from the injection of the packet header into the network until the entire packet is received at its destination. Since the path length is the same for both oblivious and chaotic routing, and conflicts are assumed not to occur, there are only two factors which affect the minimal network latency for a particular packet. These factors are the network cycle time, $c$, and the delay the message incurs at each node (in cycles), $d$. The minimum latency for a message of length $L$ flits traveling $D$ hops is then:

$$c \cdot [(D+1)d + L - 1)] \tag{3.1}$$

since the header of the packet is delayed $d$ cycles for each of the $D$ hops (plus one additional "hop" for delivery) and then the remaining $L-1$ flits follow in cut-through fashion. Since $c$ and $d$ are the only factors which can vary between chaotic and oblivious routers when congestion is small or non-existent, we compare these factors and their effects on the minimum network latency for the two routers.

*Cycle Time*

The network cycle time, $c$, is a critical factor in performance, as it is a multiplicative factor affecting the time to transfer each flit of the packet, as well as the time to

process the header of the packet at each hop. The network cycle time must be long enough for the slowest function computed in a cycle to be completed. For most of the routing logic, pipelining techniques can be applied to support very short cycle times, although a penalty will be paid in overall latency. However, a fundamental limit on the network cycle time is imposed by the chip-to-chip interface in the network since data cannot get on or off the chip at a rate faster than the I/O pads and wires connecting them can be cycled. For a given technology and interface voltage, the limits on the speed of this interface will be constant regardless of the logic inside of the routing chip. Thus, once the interface is defined, there is a maximum speed at which the chip can be run, and there is little point in attempting to make the internal logic run at a faster rate since data cannot get on or off of the chip any faster than the pad-driver limited speed.

The question remains as to whether the routers can be built fast enough to be limited only by the off-chip cycle time. With oblivious routers, pipelining of the simple routing decision, as in the Caltech MRC, allows the router to run as fast or faster than the pad drivers[1] [Seitz & Su 93]. Therefore, this limit is clearly achievable for the given technology, a 1.2$\mu$m CMOS process with a 5 V interface. For chaotic routing, the design presented in Chapter 5 achieves this using a multi-stage synchronous pipeline and the same technology. Thus, for both oblivious and chaotic routers, the network cycle time, $c$, is limited by the same factors and, thus, will be the same value.

*Routing Latency*

The second factor in comparing the minimum latency is $d$, the routing latency per node. This factor represents the time the header of a packet spends entering a node and waiting to be routed to an outgoing link, when there is no contention. Essentially, $d$ reflects the complexity of the routing decision, and it should be expected to be higher for the chaos router than for the oblivious router.

For an oblivious router with minimal buffering, the time allowed must be at least one cycle for the header to arrive on the inputs, one cycle for the routing decision to be made, and one cycle for the header to be routed to the outputs and updated. For two-dimensional networks, the routing decision is relatively simple, but the header update requires either a costly increment or decrement operation. On the other hand,

[1] The MRC is a self-timed chip, so there are no explicit clocks.

for hypercube networks, the routing decision becomes more complex due to the large number of dimensions, but the header update is simply a bit flip. Thus, for either network, the routing and updating operations together will require two cycles. Although it might be possible to combine these steps into a single cycle, this is unlikely given the clock speed constraint from the previous section. As interface technology moves toward lower voltage interfaces, the cycle time will be further reduced, possibly adding additional cycles to the routing decision as the pipeline depth is increased. Nonetheless, this results in a minimum of three cycles, so $d \geq 3$ cycles. The performance exhibited by the Caltech "Elko" MRC is consistent with our observation, having an asynchronous "cycle time" of 14.5 ns and minimum input-to-output latency of about 43.5 ns ($c = 14.5$ ns and $d = 3$ cycles)[Seitz & Su 93].

With chaotic routing, packets are delayed a total of four or more cycles by the routing pipeline (see Section 5.3.1 for details). When congestion is minimal or absent, incoming packets will immediately enter the routing pipeline and have no extra delay. Thus, for the chaos router, $d \geq 4$ cycles.

Therefore, the input-to-input latency for an oblivious router with buffering is three or more cycles, while it is four or more cycles for a chaos router. Since this delay is added only to the routing of the header flit, the minimum latency will be only one cycle per hop greater with chaotic routing than with oblivious routing, a factor which becomes negligible as the length of packets becomes large when compared to the diameter of the network. Figures 3.1–3.3 show the ratio of the minimum average latency with a chaos router to the minimum average latency with an oblivious router for differing networks and packet sizes. Each of these graphs is computed by substituting the average distance a packet must travel in the network, $D_{avg}$, computed from Equations 1.3, 1.4, and 1.5, into Equation 3.1 and varying the packet length. Since the hypercube network has a much lower diameter than the torus or mesh networks for same number of nodes, the penalty for chaotic routing is lower. Also, as packet sizes increase, the relative slowdown becomes less apparent because the additional latency due to chaotic routing is based only on the network size, and is not affected by the packet length.

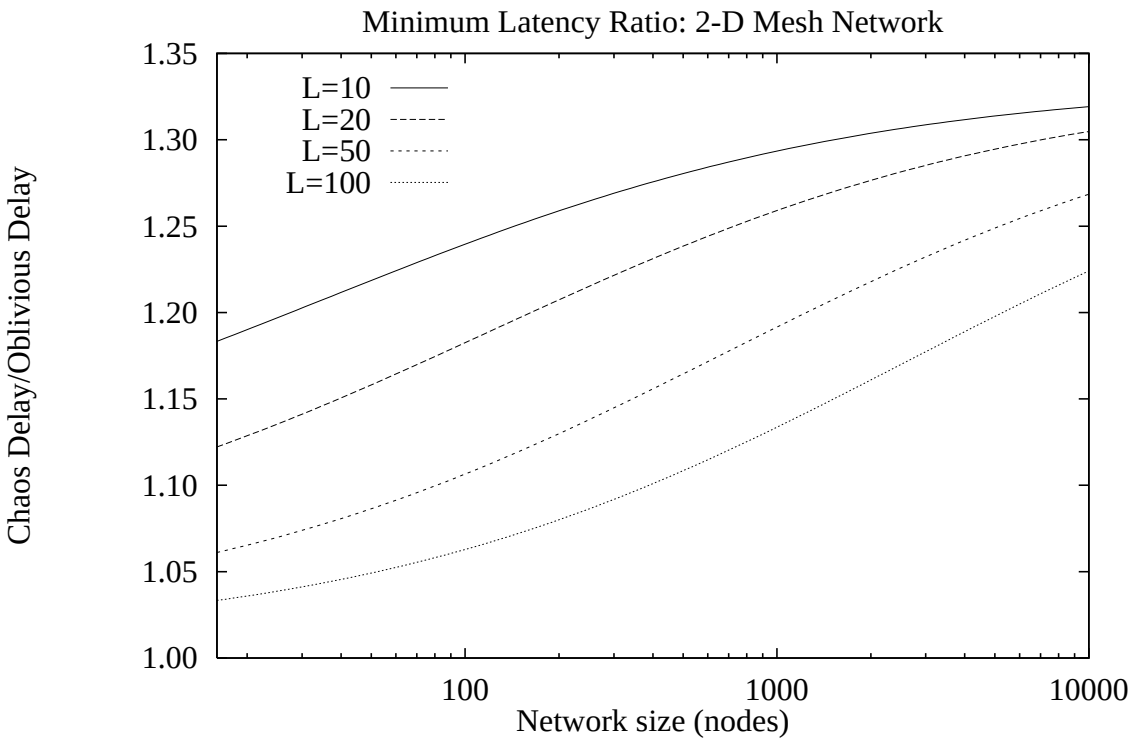



Figure 3.1: Ratio of minimum average latency with chaotic routing to minimum average latency with oblivious routing: 2-D mesh network.

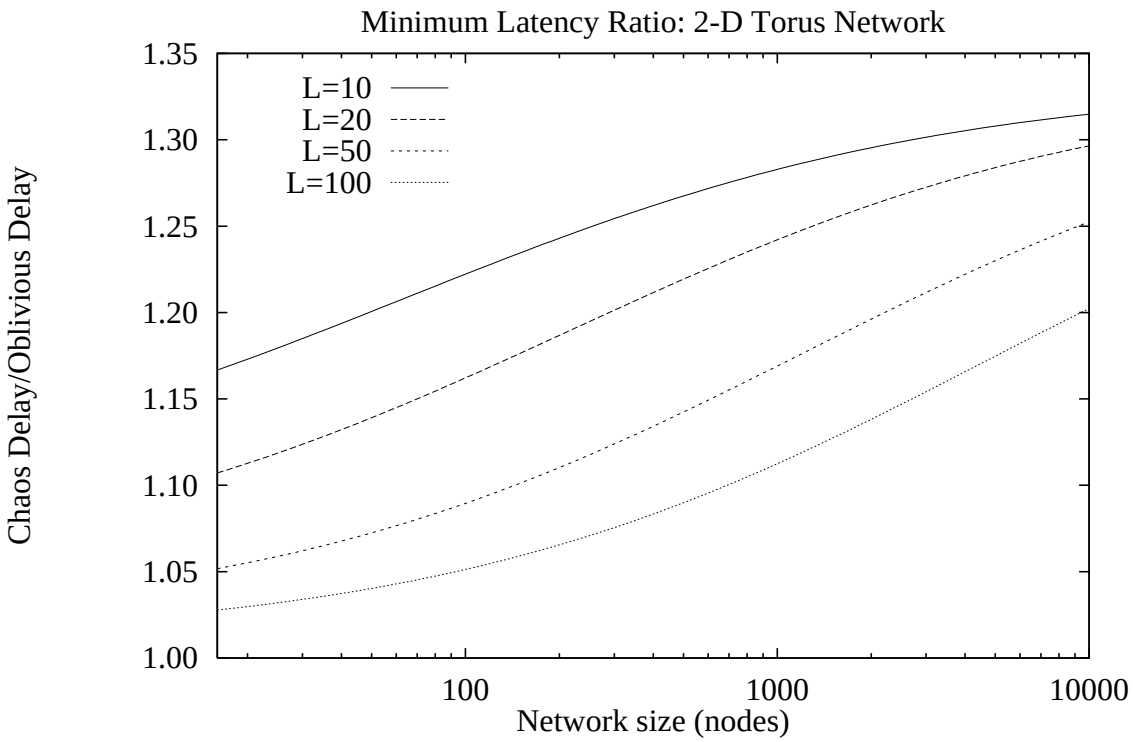



Figure 3.2: Ratio of minimum average latency with chaotic routing to minimum average latency with oblivious routing: 2-D torus network.

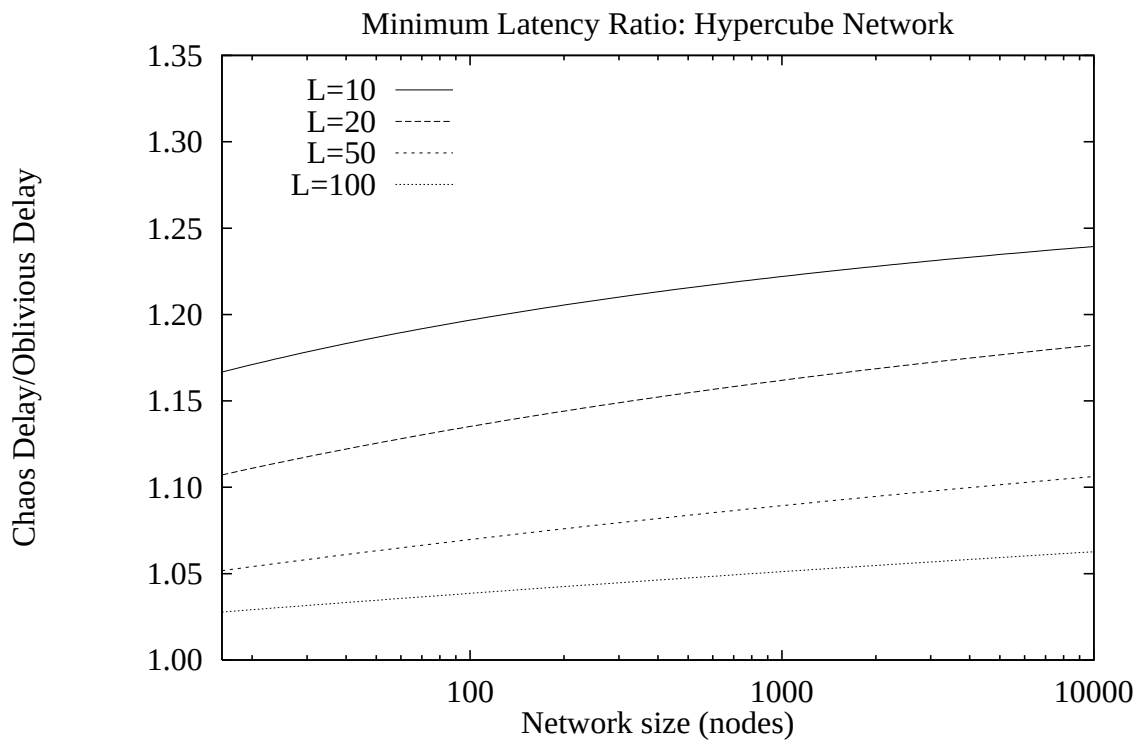



Figure 3.3: Ratio of minimum average latency with chaotic routing to minimum average latency with oblivious routing: Hypercube network.

### *3.1.2 Congested Network Performance*

As network congestion increases, the probability that packets will be routed without experiencing conflicts for resources diminishes. When the network load is in the range between "light" loading, where packets rarely experience conflicts for resources, and saturation, where the load is greater than the maximum the network can handle, packets will face some delays in routing, but the network throughput will keep up with the applied load.

In this range, a chaos network is expected to route almost all packets on minimal-length paths, although some packets may enter the multiqueue and wait for needed resources to free up. Oblivious networks will still, of course, route packets on minimal-length dimension-order paths. Because the chaos network will allow packets to choose from more than one profitable link at a time, packets should be able to reach their destinations somewhat more quickly than with oblivious networks. When the network is operating in this "moderately loaded" range, the latency gains made by the "quicker" oblivious router will be lost and chaotic routing will provide faster delivery of messages.

Once the network loading increases beyond a certain point, the network will reach saturation and will not deliver packets fast enough to keep up with the load. At this point, throughput is of more concern than latency, since the network with the higher throughput is delivering a greater number of messages, regardless of the length of their stay in the network. Because the adaptive nature of chaotic routing allows packets to have a greater number of choices when routing, a chaotic network should achieve a higher throughput before saturation than an oblivious network. Moreover, as the network nears saturation, chaos nodes will deroute packets which are not making progress, possibly moving them to less congested paths and improving performance. Overall, the saturation point will be higher with chaotic routing than with oblivious routing.

Due to the complexity of analyzing routing alogrithms with complicated buffering schemes, an analysis of congested performance is not provided. Detailed simulations are presented in Chapter 4 which confirm these expectations.

## *3.2 Chaos Router Design*

As outlined in Section 3.1, to compete with a state-of-the-art oblivious router, the chaos router must have comparable or better throughput capability at all loads and match the minimum latency experienced during light loads. The goal of equal throughput motivates a design in which the channels can be driven at the maximum rate allowed by the technology with no lengthening of the cycle time due to the router design. This is achieved by straightforward pipelining techniques at the expense of adding additional latency. In order to achieve acceptable latency across the router, the design of the router has been simplified as much as possible without sacrificing performance significantly.

The resulting chaos router has two main components: the primary "core" router, and the multiqueue. The multiqueue, though critical to the correct operation of the chaos router, can be viewed as a secondary unit for most purposes. During low load operation, when the critical measure of performance is low latency, packets do not enter the multiqueue at all. Thus, the latency-critical design centers on the core logic rather than on the more complex multiqueue logic. When congestion increases and packets enter the multiqueue more often, latency is not so critical since packets have probably already spent hundreds of cycles waiting for resources. Thus, the multiqueue design concentrates on making the best decisions in a *reasonable* amount of time with as little complexity as possible.

### *3.2.1 Packets and Data*

A critical assumption of chaotic routing is that data arrives in fixed-sized packets. This assumption is critical to the packet-exchange deadlock prevention protocol. Moreover, it allows for many simplifications in the design of the internal datapaths. By constructing the FIFOs so that they each hold exactly one packet, the number of possible states which are interesting to the router is minimized. The router must know when a packet is ready to be routed from a FIFO and when a new packet may be appended to the tail of a FIFO. This information can be obtained by keeping track of whether a header is present in a FIFO or not, and, if so, whether the header is at the front of the FIFO or not. Figure 3.4 shows the six distinguishable FIFO states. Also indicated is whether or not the FIFO is available as the destination of a packet,

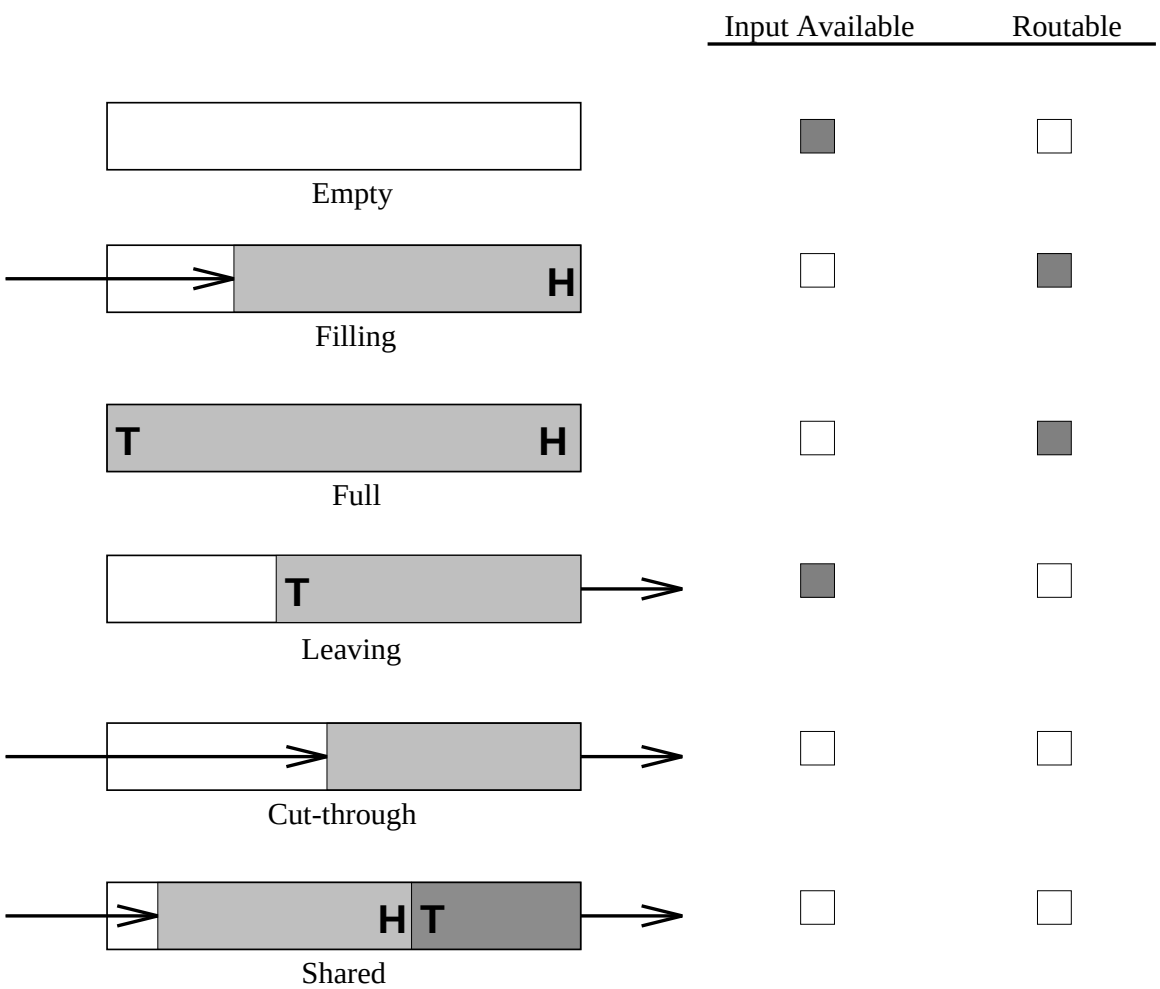



Figure 3.4: The six interesting FIFO states.

and whether or not the FIFO contains a *routable* packet. A FIFO contains a routable packet whenever there is a packet header at the front of the FIFO, as in the *Filling* and *Full* states. In the *Shared* state, the FIFO contains a header, but is blocked by the tail of the previous packet. It is not considered routable until the header reaches the head of the FIFO. Because the FIFO can only contain the number of flits in a single packet, there may only be one header in the FIFO at once, even if parts of two packets are in the FIFO, as in the *Shared* state. This invariant simplifies the logic needed to keep track of information on the FIFO states. The FIFO's input is available to receive a new packet if it is in either the *Empty* or *Leaving* states. When the FIFO is in the *Cut-through* or *Shared* states, both its input and output are being used, and it is neither available or routable. However, FIFOs in the *Shared* state are still considered as containing a packet since there is a header in the FIFO, while packets in the *Cut-through* state are simply functioning as a buffered set of wires.

### *3.3 Core Router Design*

The core router is responsible for setting up routes for all packets which cut through the router. This includes all packets which do not enter the multiqueue. The core router also must interface with the multiqueue to move packets to the multiqueue from input frames and from the multiqueue to output frames. Since the core router is latency-critical, all functions must be designed to consume minimal time.

Because the router must run at a clock speed to match the maximum speed of the pads, the design must be pipelined. The pipeline, which operates only on the header of each packet, has four primary stages:

1. Read the header into the input frame across the network channel.

2. Decode the header to identify profitable output channels for this packet.

3. Select a single output frame to route this packet to.

4. Move the header across the crossbar to the output frame and update the header to reflect the routing.

Unfortunately, new packets may be arriving in input frames, and others may be leaving output frames at any time during the routing decision. Add to this the

fact that there may be several packets desiring multiple output frames at once, and the complexity of the routing decision can be seen [Konstantinidou 91]. In order to reduce the complexity, the constraint is added that only one new route may be set up per cycle. Furthermore, during a cycle, a single output channel is selected to be considered as the destination for newly routed packets and only packets which can profitably use this output channel are considered. With these constraints, the complexity becomes manageable since the complex parallel decision of matching all packets to all destinations at once has been serialized so that the router only has to choose a single route to a fixed destination during a cycle.

The serialization constraints are enforced by the use of the *ActionDim* vector, which indicates the output frame that is currently being considered for the destination for newly routed packets. *ActionDim* is cyclicly incremented through the output frames, with the routing logic setting up routes from input frames to the output frame indicated by *ActionDim* when possible (see Figure 3.5).

To avoid "wasting" cycles where *ActionDim* points to an output frame which no packets can use, *ActionDim* is only cycled through *interesting* dimensions. A dimension $i$ is *interesting* when either:

- Output frame $i$ is available (has room for a new packet) *and*
  - A packet in the multiqueue desires output channel $i$ *or*
  - A packet in some input frame desires output channel $i$ *or*
  - The multiqueue is full (this is derouting mode).
- A packet in input frame $i$ is *stalled*, i.e., it is completely stored in the input frame, and the multiqueue is not full. This is used to detect packets which can no longer cut through and should be read into the multiqueue.

Once *ActionDim* is chosen, the router must set up the appropriate route to move a packet to the output frame specified by *ActionDim*. At the same time, a packet may be moved from the input frame pointed to by *ActionDim* to the multiqueue in order to enforce the packet-exchange deadlock prevention protocol. Once *ActionDim* has been set, the router takes the following actions.

For $ActionDim = i$:

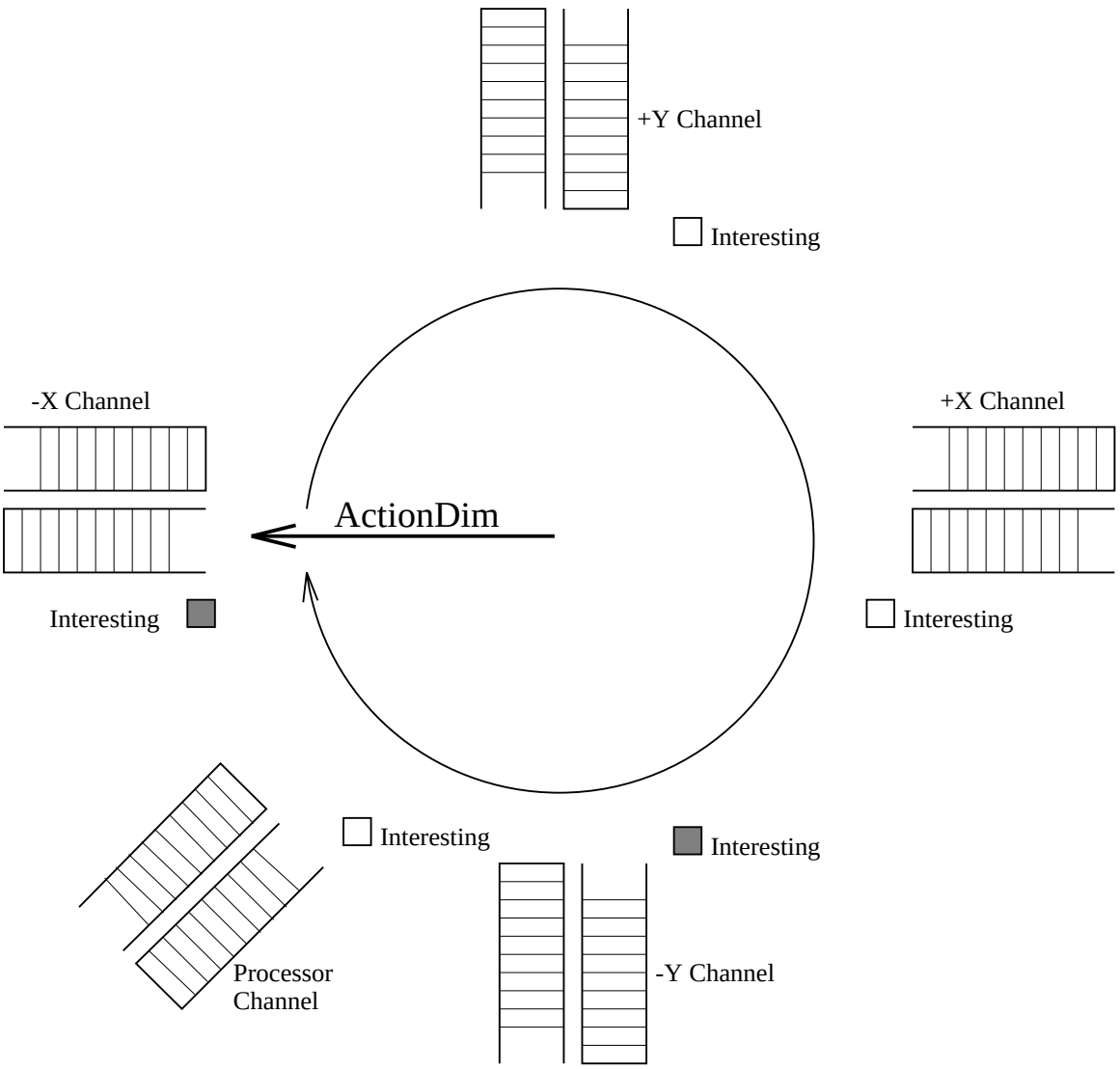



Figure 3.5: *ActionDim* selection for a two-dimensional router.

- If a packet in the multiqueue desires output channel $i$, and output frame $i$ is available, the multiqueue selects the oldest packet which can use channel $i$ and sends it to output frame $i$. *ActionDim* stalls for three cycles waiting for this process to complete. If the multiqueue is full and no packets want output channel $i$, a packet is randomly selected to be derouted to channel $i$. *ActionDim* must stall five or more cycles waiting for this process to complete. Details of this operation are presented in Section 3.4.

- Otherwise, if a routable packet in input frame $j$ desires output channel $i$ and output frame $i$ is available, the packet is routed directly (cuts through). This takes only a single cycle.

- Regardless of whether a packet is routed, if a routable packet is present in input frame $i$, it is read into the multiqueue. This operation implements the packet-exchange protocol and also provides the mechanism for reading "stalled" packets into the queue. This is concurrent with the above two operations and takes a single cycle unless the multiqueue is full, in which case *ActionDim* stalls until room becomes available in the multiqueue due to another packet leaving the multiqueue.

More details on the design of the control logic and multiqueue logic can be found in Sections 5.3 and 5.4.

In the case where there is only one routable packet in the router, its header will be decoded immediately. On the next cycle, *ActionDim* will be moved to point to an output frame the packet can be routed to, and it will be routed on the very next cycle. Allowing a single cycle for the header to originally arrive at the router and an additional cycle to update the header as it leaves results in the four cycles minimum latency for a packet to be routed in a chaos router.

### *3.4 Multiqueue Design*

Logically, the multiqueue is a set of FIFO queues holding packets waiting for each output frame. Since a packet may desire multiple output frames, it may be present in multiple queues simultaneously. An example of a logical multiqueue holding five packets is shown in Figure 3.6. In this figure, packets 1, 2, and 5 each desire more

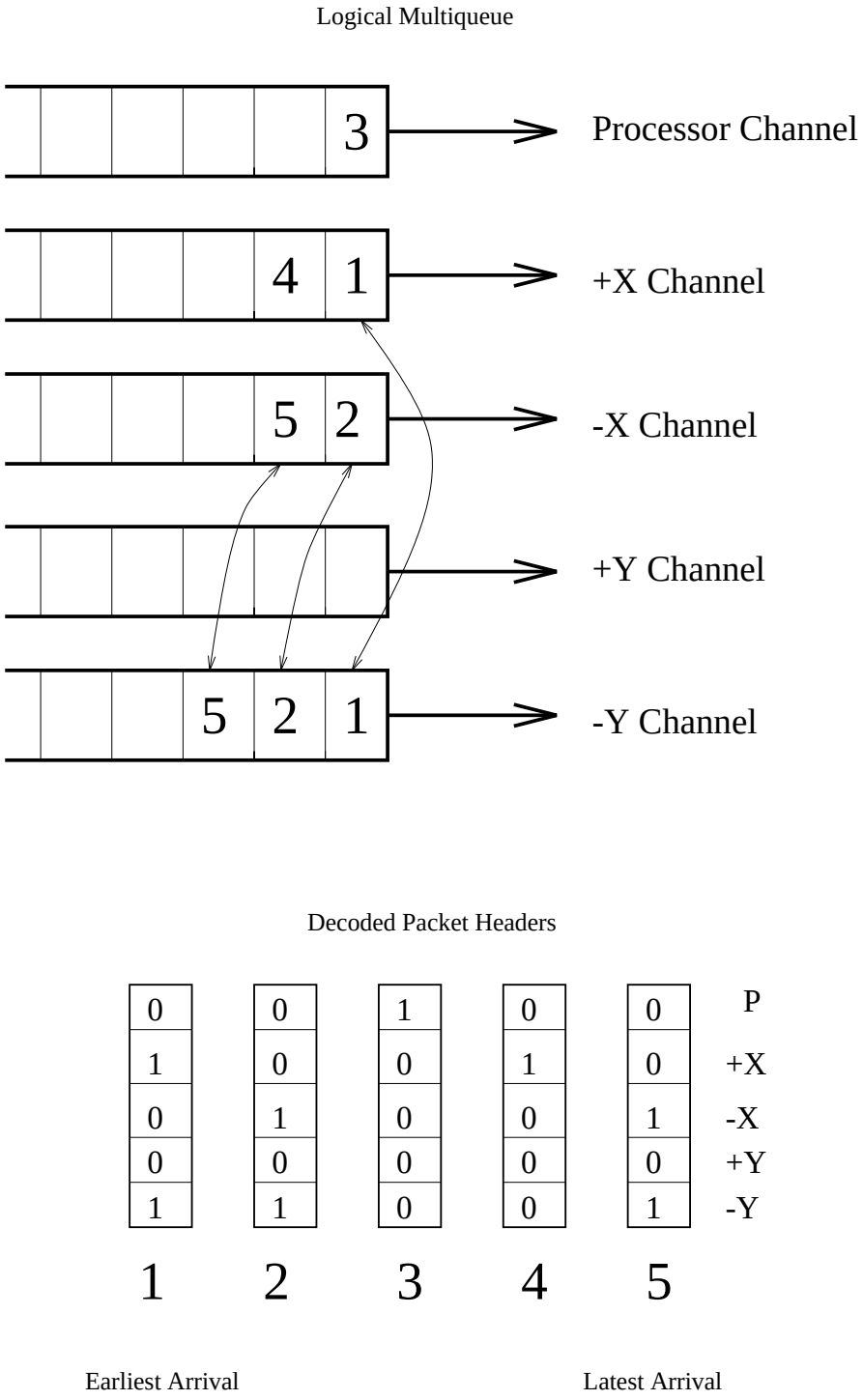



Figure 3.6: Logical MultiQueue diagram. $d = 5$ and $Q = 5$.

than one output channel, so they have entries in more than one queue. However, since the entries represent the same packet, if one of these packets advances out the head of one queue, its entries must be removed from all other queues, regardless of their position within the queue.

As with the core router logic, only one output channel is considered at a time in the multiqueue. The chosen channel is the one pointed to by *ActionDim* in the control logic. The multiqueue must choose the first (oldest) packet in the logical queue for this channel and send it to the output frame for the indicated channel.

Although the operation of a logical multiqueue is straightforward, its implementation is not as simple. Two problems arise: space for the maximum number of packets in the multiqueue must be provided in each separate queue, resulting in considerable wasted space, and entries which correspond to the same packet must somehow be linked together so that the same packet will not be routed twice. Since it may be the case that all packets in the multiqueue desire the same output channel, each channel's queue must be able to hold the maximum number of packets which can be present in the queue. Thus, the space requirements are $d \cdot Q$, where $d$ is the number of output channels and $Q$ is the number of packets the queue can hold.

To implement the multiqueue, the datapath is separated from the control logic. The datapath actually stores the packets and has only enough space to hold the total maximum queue population, $Q$ packets. Thus, there will be $Q$ FIFO buffers in a $Q$-slot multiqueue. The control logic keeps track of the ordering of packets and the mapping of packets to logical queues within the multiqueue.

The multiqueue datapath consists of $Q$ queue slots, each capable of holding a single packet. The core router control logic sets up transmissions from input frames to queue slots (via the queue input crossbar) and from queue slots to output frames (via the main crossbar) as directed by the multiqueue control logic. Once a transmission is set up by the control logic, it proceeds autonomously in a cut-through fashion without intervention until it is finished.

The multiqueue control logic has two primary tasks: selecting an empty queue slot for an incoming packet to be routed to and selecting a packet from the queue to be routed out to the output frame specified by *ActionDim*. The first task is relatively easy: any empty queue slot will do. The second is very complex because, during normal operation, the multiqueue must select, in FIFO priority, a packet in one of

the queue slots which desires the channel indicated by *ActionDim*. If the multiqueue is derouting, a packet must instead be selected randomly from among those in the multiqueue. To accomplish this, the control logic uses a *scoreboard* to keep track of where packets are in the multiqueue.

### *3.4.1 Multiqueue Scoreboarding*

The multiqueue scoreboard manages information about where packets are in the multiqueue and their relative ordering. The multiqueue must assign packets to queue slots without conflicts and remove them in a manner consistent with the logical multiqueue definition. The scoreboard provides a map of the pertinent information so that the control logic can correctly implement the multiqueue.

As new packets enter the multiqueue, their headers are decoded and information about the packet and where it is stored is loaded into the scoreboard. By keeping track of the headers, the assignment of queue packets to queue slots, and the chronological ordering of packet arrival, the scoreboard holds all of the information necessary to implement the logical multiqueue.

The scoreboard is composed of two primary units, the *destination matrix* and the *storage matrix*, as well as some supporting logic. Each matrix has $Q$ columns, one for each slot in the multiqueue. Figures 3.7 and 3.8 show the matrices for a two-dimensional ($d = 5$) router with $Q = 6$.

The relative age of a matrix entry is determined by its column position, with the entry for the packet which has been in the multiqueue the longest being in the rightmost position. Note that a packet's "age" in the multiqueue is relative to its entry into the multiqueue instead of its injection into the network. New entries into the matrices are made on the left side, preserving the right-to-left age ordering. When the header of a packet leaves the multiqueue, its entry is deleted from the scoreboard and the matrices are compressed to the right.

#### *Destination Matrix*

The destination matrix keeps track of information on the channels desired by packets in the multiqueue. It has two primary functions: to indicate which channels are desired by packets in the multiqueue and to identify which packets in the multiqueue can profitably use particular output dimensions. Each record in the destination ma-

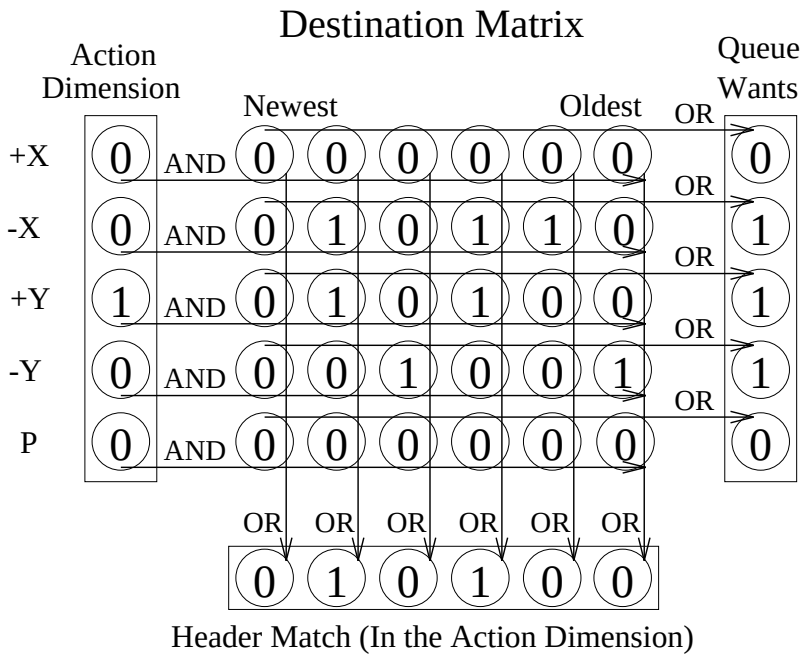



Figure 3.7: Multiqueue destination matrix.

trix, corresponding to a single column, contains the decoded header for the packet represented by that record. Thus, there is a single row in the matrix for each possible output dimension. Since records are loaded into the destination matrix from the left side, the records are ordered in a right-to-left oldest-to-youngest ordering. Therefore, each row of the destination matrix is a chronological queue for a particular output channel.

The destination matrix has as inputs the decoded headers of packets arriving in the multiqueue and the *ActionDim* vector. Its outputs are the two vectors *QueueWants* and *HeaderMatch*.

*QueueWants* is a vector indicating which output channels can be profitably used by at least one of the packets waiting in the multiqueue. Thus, if an element of *QueueWants* is '1', then there is at least one packet in the multiqueue which can profitably use the corresponding output channel. *QueueWants* is communicated to the core router control logic so that any output frames desired by some packet in the multiqueue will be aware of this fact and initiate routes from the multiqueue to themselves when possible. A special case occurs when the multiqueue is full and in *derouting* mode. Since the next available output frame will be chosen as the destination for

some (randomly chosen) packet in the multiqueue, all output dimensions (except the processor dimension) are desired by the multiqueue and, thus, all elements, except the one corresponding to the ejection channel, of *QueueWants* are asserted.

In Figure 3.7, an example of computing *QueueWants* is shown. In this example, the multiqueue contains five packets. By ORing all of the elements in a row together, the *QueueWants* vector is determined. In this case, the negative X, and both positive and negative Y channels are desired by at least one of the five packets.

The second output of the destination matrix is the *HeaderMatch* vector. This vector indicates which records in the matrix correspond to packets which can profitably use the channel indicated by *ActionDim*. Thus, if the $i$th element of *HeaderMatch* is asserted, then the $i$th record in the destination matrix corresponds to a packet which can profitably use channel[*ActionDim*]. Matched records are located by bitwise ANDing *ActionDim* with each column of the destination matrix. The selected records are then identified by ORing all of the elements in a column together to produce the *HeaderMatch* vector.

Figure 3.7 shows an example of the calculation of the *HeaderMatch* vector. In this example, *ActionDim* points to the positive Y channel. There are two packets in the multiqueue which desire this channel, and their scoreboard records are in the third and fifth positions in the destination matrix. Thus, the third and fifth elements of *HeaderMatch* are asserted.

### *Storage Matrix*

The second scoreboard matrix is the *storage matrix*. The storage matrix stores information on the correspondence between records in the scoreboard and packets in queue slots. As a packet enters the multiqueue, a one-hot vector indicating the queue slot the packet is in is loaded into the storage matrix. This happens simultaneously with the loading of the destination matrix so that the records for the same packet correspond. The columns of the storage matrix are locked together with the columns of the destination matrix so the record for a single packet in the multiqueue appears in the same column in both matrices.

The primary purpose of the storage matrix is to identify the queue slot which holds a packet whose record is selected in the destination matrix. This is done by transforming the *Priority* vector to the *QueueOut* vector. *Priority* is a vector which

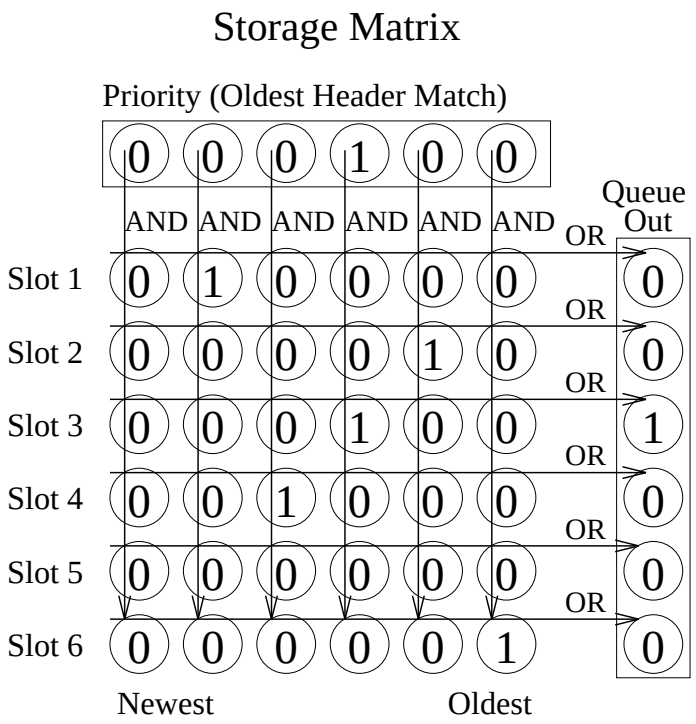



Figure 3.8: Multiqueue storage matrix.

identifies the *oldest* matched header in the *HeaderMatch* vector, unless the multiqueue is in derouting mode. In this case, *Priority* is selected randomly. Thus, *Priority* indicates the record for the packet selected to be routed out the channel indicated by *ActionDim*. *Priority* is then fed into the storage matrix and ANDed with each row to identify the proper queue slot. The result of this computation is then row-wise ORed to form the *QueueOut* vector that identifies the queue slot which holds the matched packet.

An example of the transformation of *Priority* to *QueueOut* is shown in Figure 3.8. In this example, there are five occupied queue slots in the multiqueue. The packet with the third record from the right has been selected to be routed to channel[*ActionDim*]. The matrix shows that this matches the third queue slot and, thus, asserts the third element of the *QueueOut* vector.

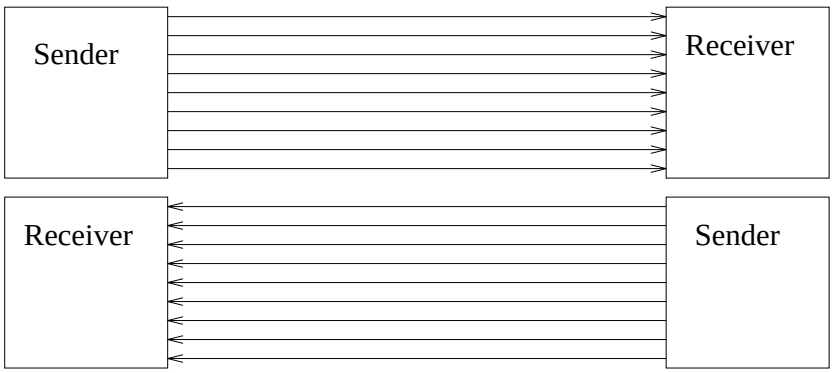
### *3.5 Channel Design*

Each of the external channels is implemented using a single set of bi-directional wires. The decision to use one set of shared wires rather than two sets of uni-directional wires is based on pin limitations. In order to use uni-directional channels with the same number of pins, each channel must be one-half the width it would be if bi-directional channels were used (See Figure 3.9). Actually, since the control signals must be duplicated when using two uni-directional channels, but only one set is necessary when using bi-directional channels, the uni-directional channels must have slightly less than half as many data wires as bi-directional channels would have. Using shared channels not only allows wider data words, but allows header flits to contain more routing information.

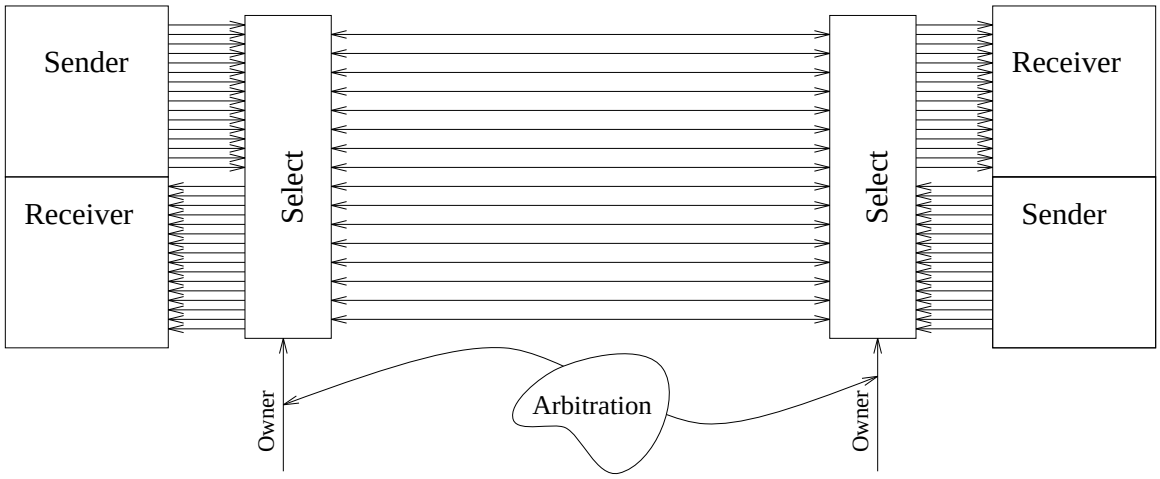
In order to share the channels, though, arbitration to determine which side has ownership at a given time must be provided. This arbitration introduces complexity, and requires information exchange between the owners to ensure mutual exclusivity of ownership. If the channel is constantly needed in both directions, then any fair arbitration protocol will give each side ownership 50% of the time. Therefore, under full loading, one can have either a dedicated half-width channel which is available all of the time or a shared full-width channel which is available one half of the time. Either way, the effective bandwidth is the same. Since uni-directional channels are easier to implement, this would seem to be the favored choice.

However, the situation changes when the channels cannot be fully utilized in both directions. Consider the case where a single message is being transmitted through the network, with no other traffic. With bi-directional channels, it will be able to use each channel 100% of the time since there is no opposing traffic. Since the channels are twice as wide as uni-directional channels, the message will be transmitted in considerably less time. Thus, in this case, the advantage of bi-directionally shared channels is clear.

Another consideration is that of matching the widths of the internal data paths with the widths of the external channels. Ideally, all data paths will be the same width so flits can be transmitted without extra effort to match widths. With bi-directional channels, the internal data paths, which are uni-directional, can be twice the width of those using uni-directional channels. This is illustrated in Figure 3.9. Since router chips tend to be pad-limited instead of area-limited, there is no penalty




Dual Uni-directional 8-bit Channels




Single Bi-directional 16-bit Channel

Figure 3.9: Uni-directional vs. Bi-directional Channels.

for the added on-chip area for the wider data paths. Thus, with bi-directional channels, the absolute bandwidth of the internal router is twice that of a router using uni-directional channels.

The use of bi-directional channels requires the use of some arbitration protocol, though. The protocol should avoid deadlock and starvation and be as fair as possible. Also, the protocol should try to maximize the channel utilization for a given load, within the previous constraints. One common way of sharing a channel is to pass the ownership back and forth on each cycle: on even-numbered cycles, side $A$ owns the channel; on odd-numbered cycles, side $B$ owns the channel. While this protocol is deadlock and starvation free, as well as fair, it does not maximize utilization. When the channel is desired in only one direction, the utilization will be only 50%, when it could be 100%.

A protocol which allows one side to use the channel exclusively when there is no opposing traffic is desired. Such a protocol, though, requires information about the conditions on either side of the channel to be shared. Both sides must know if there is a packet on the other side waiting to get through, as well as if there is room on the other side to receive a packet. To arbitrate on the transmission of each flit is much too costly, so the channel is designed to allow arbitration only at packet boundaries. Thus, once the transmission of a packet has started in one direction, it will continue until the entire packet has been transmitted. Since the packets are all fixed-size, the protocol remains fair. If packets are allowed to be of unbounded length, as in wormhole routing, the possibility of the monopolization of the channel by a single very long message exists.

There are two advantages to performing arbitration at packet boundaries. The first is easier implementation. Since arbitration occurs less often, there is more time to compute a more complex protocol which takes into account more information and, thus, achieves higher channel utilization. The second advantage is lower average latency. Consider two packets, traveling in opposite directions across a channel, which arrive at the channel at the same time. Assuming the packets are 20 flits in length, if the channel ownership changes every cycle, the first packet will be completely transmitted on the 39th cycle, and the second on the 40th, giving an average latency of 39.5 cycles. On the other hand, if channel ownership only changes at packet boundaries, then the first packet will be completely transmitted on the 20th cycle,

and the second on the 40th cycle. This results in an average latency of 30 cycles instead.

Details of the actual channel arbitration protocol used are presented in Chapter 5.

## *3.6 Performance*

Because some decisions made in the design of the chaos router reduce the rate at which new packets may be routed, their effect on the actual performance must be evaluated. The most apparent of these is the restriction that only one new packet may be routed per cycle (or less often in some cases). As before, the primary performance measures of concern are latency and throughput.

To analyze the performance of the router with respect to this constraint, the frequency at which *ActionDim* visits each channel in the worst case must be formulated. If no packets are being sent from the multiqueue, *ActionDim* will be incremented each cycle and each of the $d$ channels will be visited at least once every $d$ cycles. When packets are being sent from the multiqueue, though, *ActionDim* cannot be moved as quickly since it must stall and wait for the multiqueue operations to complete. If the router must stall $t_{mqStall}$ cycles for each multiqueue routing decision, then each output channel will be visited once every $t_{mqStall} \cdot d$ cycles instead of once every $d$ cycles.

### *3.6.1 Multiqueue Performance*

Since the multiqueue is not on the critical path of the router, optimization of its performance is not extremely important. Thus, the multiqueue may take several cycles to complete its operations. There are two primary operations: *read* and *send*. In a read operation, the core control logic specifies that a packet should be read into the multiqueue from an input frame. Since the multiqueue only has to select an empty queue slot to place the packet in, this operation usually takes only a single cycle and does not stall the router. In fact, it can be performed concurrently with the routing of another message across the main crossbar. Only if there are no queue slots which can accept packets, but the multiqueue must read in another packet, does the router stall until room is available in the multiqueue.

Selecting a packet to send from the multiqueue is a more complex operation.

When the multiqueue is not in derouting mode, it takes three cycles for it to match a packet to *ActionDim* so that packet can be sent to the output channel indicated by *ActionDim*. This delay, $t_{match}$, is, therefore, equal to three cycles. Since the packet selection is all that is needed to perform a *send* operation in non-derouting mode,

$$t_{mqStall} = t_{match} = 3 \text{ cycles}$$

When the multiqueue is derouting, though, the timing is more complicated.

When derouting, the multiqueue must first check if any packets in the multiqueue can *profitably* use the selected output channel by performing the same match operation as above. Assuming this test fails, a packet must be randomly picked to be derouted out the channel indicated by *ActionDim*. This process takes $t_{deroute}$ cycles. $t_{deroute}$ may be as little as one cycle, but may take longer, as described below. Finally, $t_{read}$ cycles are taken to read in a new packet to replace the derouted one. $t_{read}$ takes only a single cycle, and can normally be overlapped with $t_{match}$. However, when derouting, the read into the queue cannot begin until a vacant slot is left in the queue by the derouted packet. Thus, when derouting, the delay is:

$$t_{mqStall} = t_{match} + t_{deroute} + t_{read} \geq 5 \text{ cycles} \tag{3.2}$$

*Routable Queue Slots*

Because of the dynamic nature of the multiqueue, not all of the $Q$ queue slots will necessarily contain routable packets, even when the multiqueue is full and in derouting mode. This can happen when some of the queue slot FIFO's are in the *Shared* state (refer to Figure 3.4). In fact, it may be possible that none of the slots have immediately routable packets. Since only routable packets have entries in the multiqueue scoreboard and can be sent out of the multiqueue, the derouting process must narrow its range to include only those packets. If there are no routable packets in the multiqueue, the logic must stall and wait until at least one becomes routable and can be moved out of the multiqueue. An analysis of how many cycles it may take, in the worst case, for at least one packet to become routable follows.

Since packets are only non-routable if the tail of a previous packet is blocking their progress, the time to become routable is just the time it takes for the previous packet to completely leave the FIFO. The FIFO buffers each hold the number of flits

in a single packet, $L$. If the header of a non-routable packet is in a FIFO, only $L-1$ flits, maximum, of the previous packet may remain in the FIFO. Considering the fact that once a packet has begun leaving a FIFO, it will continue without interruption, it is easily shown that there is a limit of $L-1$ cycles on the time a packet may remain non-routable.

This bound, however, may be tightened considerably by observing that only one packet may be sent from the multiqueue at a time. Because of this serialization, the times at which packets leave the multiqueue are staggered and, thus, the length of time when no routable packets are available is reduced. Because not all packets can be routed out at once, the worst case is then when the packets are routed out of the multiqueue with the minimum time between them. Since, when derouting, the multiqueue can only route one packet every five cycles, even if all packets are routable, packets being sent from the queue must be spaced five cycles apart. Figure 3.10 shows the worst case situation for a queue which can hold five 20-flit packets. The order of arrival of the packets is indicated by $i$, with the lowest number corresponding to the most recent arrival. In this case, two packets are immediately routable, with the remaining three becoming routable at the rate of once every five cycles.

The general analysis of the number of routable packets in the multiqueue depends on the queue size and the packet length. Define $r_{i,n}$ as a boolean function which returns whether or not the $i$th most recently arrived packet (with $i = 1$ referring to the newest packet) in the multiqueue is routable $n$ cycles after the multiqueue enters derouting mode. In the worst-case steady-state[2] situation, where packets are constantly being derouted from the multiqueue and replaced by new packets, a packet must wait $L-1$ cycles to become routable. Thus, for packets of length $L$:

$$r_{i,n} = \begin{cases} 1, & \text{if } (L-1) - 5i - n \leq 0 \\ 0, & \text{otherwise} \end{cases} \tag{3.3}$$

The number of routable packets in a multiqueue which can hold $Q$ packets at a time $n$ cycles after derouting mode is entered is:

$$R_n = \sum_{i=1}^{Q} r_{i,n} \tag{3.4}$$

[2] There are a few possible situations where it may take longer for packets to become routable, but they are transient in nature.

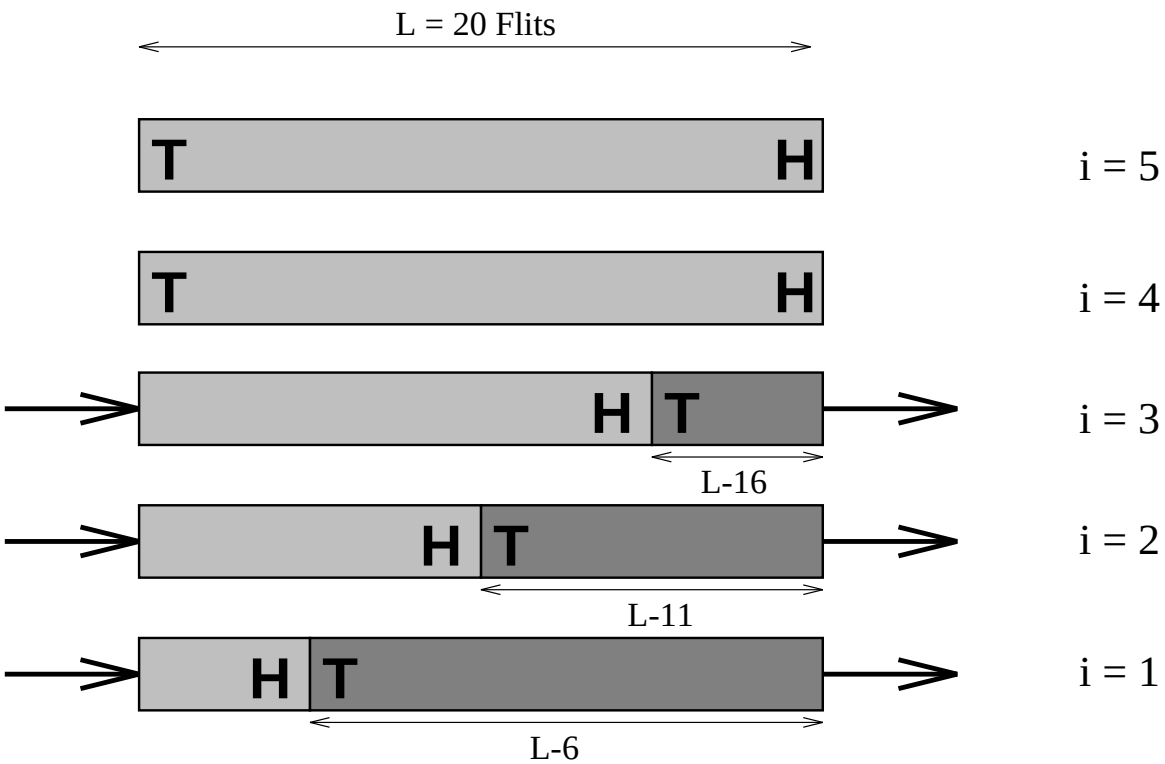



Figure 3.10: Worst case number of non-routable packets in the multiqueue with $Q = 5$ and $L = 20$.

The number of cycles the multiqueue must wait for at least one packet to become routable is the smallest $n$ for which $R_n$ is non-zero. Table 3.1 details information on how long it takes before packets become routable when the queue size and packet lengths are varied. The third column of the table shows the length of time the queue must wait, in the worst case, until at least one packet becomes routable.. As shown in Table 3.1, the larger the queue size and the smaller the packet length, the less time the logic must wait until at least one packet becomes routable. Indeed, for some combinations of queue size and packet length, there will always be at least one routable packet as derouting mode is entered. For these cases, the number of routable packets at $n = 0$ is shown in the fourth column of Table 3.1.

From Table 3.1, it can be seen that for a queue size of five packets and a packet length of 20 flits, the configuration chosen for implementation in Chapter 5, there will always be at least two routable packets in the multiqueue when derouting. Thus, there should be no extra waiting for a packet to become routable with the implemented configuration.

*Random Choice when Derouting*

While it is only necessary to wait for a single packet to become routable when choosing a packet to deroute, there are two problems. The first is the necessity of the derouting choice to be *random*. A random choice out of a set containing only one packet is hardly a random choice at all. In the cases where only one packet is routable when derouting, the first packet to become routable is the packet which has been in the multiqueue for the longest time. Thus, when the first packet to become routable is chosen to be derouted, the derouting decision is deterministic: the packet which has been in the multiqueue the longest is chosen to be derouted. A deterministic choice such as this upsets the basic source of randomness which is needed to provide probabilistic livelock freedom in chaotic routing.

The second problem with choosing from among the first routable packets when derouting is that it is difficult to implement. It is necessary to produce a random one-hot vector for which the single asserted bit is one of the routable packets. To produce this, the hardware must take into account exactly which packets are routable when generating a random choice. As it turns out (see Chapter 5), this is a somewhat difficult problem, the solution of which requires a large amount of hardware.

Table 3.1: Delay in waiting for packets to become routable for various packet lengths ($L$) and queue sizes ($Q$).

| $L$ | $Q$ | Delay (cycles) until $\geq 1$ routable packet | Number of routable packets when starting derouting. |
|---|---|---|---|
| 10 | 1 | 5 | 0 |
| 10 | 2 | 0 | 1 |
| 10 | 3 | 0 | 2 |
| 10 | 4 | 0 | 3 |
| 10 | 5 | 0 | 4 |
| 20 | 2 | 10 | 0 |
| 20 | 3 | 5 | 0 |
| 20 | 4 | 0 | 1 |
| 20 | 5 | 0 | 2 |
| 20 | 6 | 0 | 3 |
| 30 | 3 | 15 | 0 |
| 30 | 4 | 10 | 0 |
| 30 | 5 | 5 | 0 |
| 30 | 6 | 0 | 1 |
| 30 | 7 | 0 | 2 |
| 40 | 3 | 25 | 0 |
| 40 | 4 | 20 | 0 |
| 40 | 5 | 15 | 0 |
| 40 | 6 | 10 | 0 |
| 40 | 7 | 5 | 0 |
| 40 | 8 | 0 | 1 |
| 40 | 9 | 0 | 2 |

To solve both of these problems, a more probabilistic selection mechanism is used. Instead of choosing only from queue slots which contain routable packets, the multiqueue chooses randomly from among *all* queue slots. This solves the complexity problem, since the target space is fixed, and the randomness problem, since the choice is always over the whole multiqueue. However, a new difficulty is introduced: the selected packet may not be routable. It is possible to simply wait for the packet to become routable, as it is guaranteed to do so. However, this may cause a long delay, possibly as long as $L-4$ cycles before the packet becomes routable and can be routed out. A solution to this is to check if the chosen queue slot has a routable packet; if it does not, the random choice is repeated on the next cycle. Once again, the random logic may always choose the same, late packet, but the chance of this is much less.

The extension of the previous analysis of delay to include this new selection technique can be done by adding in the probabilistic nature of the new design. Recall that $R_n$ from Equation 3.3 gives the number of routable packets $n$ cycles after derouting has begun. The probability that a routable packet is chosen on the $n$th cycle, given that no packets were chosen on the previous $n-1$ cycles is $Hit_n$.

$$Hit_n = \frac{R_n}{Q} \tag{3.5}$$

Because the $n$th try is only possible if the previous $n-1$ tries failed, it is necessary to multiply $Hit_n$ by the probability that the previous $n-1$ tries failed. Adding the probability that a hit occurred on one of the previous $n-1$ cycles gives the probability that a hit occurs on the $n$th cycle or earlier, $Routed_n$.

$$Routed_n = Hit_n \cdot (1 - Routed_{n-1}) + Routed_{n-1} \tag{3.6}$$

Because all packets in the multiqueue will become routable after $L-4$ cycles, there is a finite bound on the number of cycles before $Routed_n$ is equal to one. However, how fast $Routed_n$ approaches this depends on the queue size and packet length. Table 3.2 shows the expected number of cycles until a hit occurs, the number of cycles until $Routed_n \geq 0.95$ and the number of cycles until $Routed_n = 1.0$. As can be seen from Table 3.2, the time the queue controller must delay when constantly derouting varies significantly with queue size and packet length. The worst case delays are a function only of the packet length, as it is possible that a routable packet is not chosen until the very last packet to enter the multiqueue becomes routable. However, as can be

Table 3.2: Queue delay characterization for a variety of packet and queue sizes.

| $L$ | $Q$ | Expected Delay (cycles) | $n \mid Routed_n \geq 0.95$ | $n \mid Routed_n = 1.0$ |
|---|---|---|---|---|
| 10 | 2 | 1.98 | 5 | 6 |
| 10 | 3 | 1.50 | 4 | 6 |
| 10 | 4 | 1.33 | 3 | 6 |
| 10 | 5 | 1.25 | 2 | 6 |
| 20 | 2 | 11.97 | 15 | 16 |
| 20 | 3 | 7.80 | 11 | 16 |
| 20 | 4 | 3.52 | 8 | 16 |
| 20 | 5 | 2.43 | 6 | 16 |
| 20 | 6 | 1.98 | 5 | 16 |
| 30 | 3 | 17.80 | 21 | 26 |
| 30 | 4 | 13.52 | 18 | 26 |
| 30 | 5 | 9.16 | 14 | 26 |
| 30 | 6 | 4.76 | 11 | 26 |
| 30 | 7 | 3.28 | 8 | 26 |
| 40 | 3 | 27.80 | 31 | 36 |
| 40 | 4 | 23.52 | 28 | 36 |
| 40 | 5 | 19.15 | 24 | 36 |
| 40 | 6 | 14.74 | 21 | 36 |
| 40 | 7 | 10.28 | 16 | 36 |
| 40 | 8 | 5.78 | 12 | 36 |
| 40 | 9 | 4.04 | 10 | 36 |

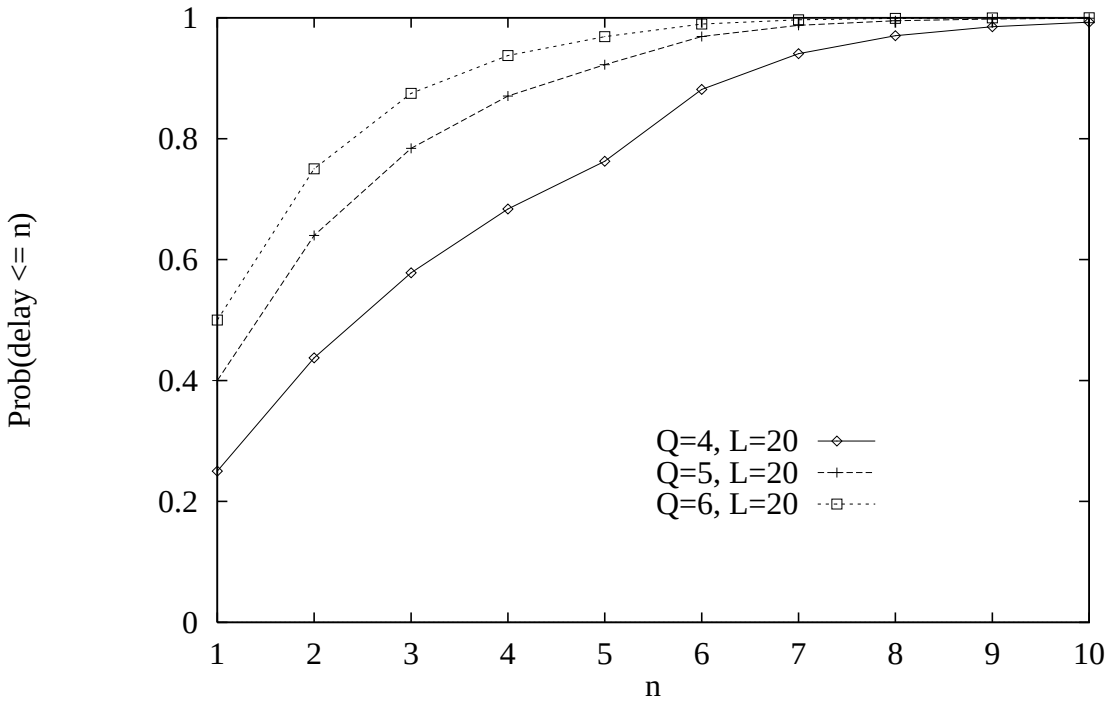



Figure 3.11: Probability of routing within $n$ cycles for various queue sizes, $L = 20$.

seen from the 95% probability levels, the chance of being delayed very long is very dependent on the queue size: the larger the queue, the more routable packets it will contain. Figure 3.11 shows the value of $Routed_n$, the probability that the delay is less than or equal to $n$, as a function of $n$ for queue sizes 4, 5, and 6 when the packets are of length 20 flits. As shown in the figure, the probability of being routed within $n$ cycles increases very quickly to nearly 100% when the queue size is five or six packets.

For the two dimensional chaos router implementation described in Chapter 5, with $L = 20$ and $Q = 5$, the average delay for derouting when constantly derouting, $t_{deroute}$, is 2.43 cycles. Substituting this into Equation 3.2, $t_{mqStall} = 6.43$ cycles. This means that, if the router experiences a loading pattern where it must constantly deroute and read new packets into the multiqueue, it will be able to route a packet every 6.43 cycles on average. If the more difficult and less random solution presented in the previous section were used for selecting packets to deroute, $t_{deroute}$ would be a single cycle and $t_{mqStall}$ would be 5.0 cycles. Thus, there will be some performance

loss with the easier solution.

### *3.6.2 Core router performance*

The frequency at which *ActionDim* visits each output channel affects both the latency of packets which go through the router and the throughput of the router itself. If the router is derouting packets, an additional $t_{mqStall} \cdot d$ cycles may be added to the latency of packets passing through the router. However, if the router is derouting packets, the network must be at least locally congested and packets will already be experiencing very inflated latencies regardless of the frequency at which they may be routed. On the other hand, when congestion is very light and latency is of greater importance, packets will not enter the multiqueue at all, and the added delay will be less than $d$ cycles. Moreover, the only time that a packet will experience any delay at all in receiving service is when there are other packets being serviced by the router already. The lighter the loading, the less likely it is that packets will have any extra delay at all. Thus, the serialization of the routing decision has very little actual impact on the latency that packets experience.

The throughput of the router is also affected by the frequency that *ActionDim* cycles through the dimensions. If packets arrive more frequently than the router can service them, a backlog of unrouted packets will build up and the router's control logic will become the bottleneck in the routing network. From the previous analysis, the worst-case service rate is a rate of once every $t_{mqStall} \cdot d$ cycles. Thus, new packet headers must not arrive more often than this. However, when the multiqueue is derouting, *ActionDim* cannot visit the processor channel, as derouting to the processor would mis-deliver packets, so $(d-1)$ should be used instead of $d$ in this analysis. Therefore, when derouting, each output channel should be visited at least once every $t_{mqStall} \cdot (d-1)$ cycles. For a two-dimensional router with packets of length 20 flits and a queue size of five packets, $d = 5$ and $s = 6.43$, so output channels should be visited at least once every 25.72 cycles.

Fortunately, the control logic only needs to service the *headers* of packets, and an upper bound on the frequency of header arrivals can be determined. Since packets are multiple flits in length, specifically, of length $L$, headers can arrive on incoming channels at most once every $L$ cycles, with $L-1$ data flits intervening. Thus, as long as $L \geq t_{mqstall} \cdot (d-1)$, the router will be able to keep up with the incoming

packets with no backlog. For a two-dimensional router mentioned above, $L = 20$ and $t_{mqStall} \cdot (d-1) = 25.72$. Thus, the router may not be able to avoid backlogs. However, this analysis is too conservative in that it does not take into account the fact that the channels are bi-directionally shared. Specifically, if the worst case happens and there is a packet derouted to each channel and new packets read into the multiqueue from each channel, the incoming and outgoing packets must share the same channels. Since the packets cannot simultaneously use the same channel, new headers will arrive only once every $2L$, or 40, cycles, and there will be no chance of a backlog occurring since this is greater than the 25.72 cycles between visits to each channel. Therefore, there will be no throughput limitations for a 20-flit packet size, two-dimensional router. Routers with shorter packet sizes or more channels may have problems, though, and should be re-subjected to this analysis.

## *3.7 Summary*

A detailed design for a chaos router was presented in this chapter. The design is motivated by two primary design goals: low latency and high throughput. By comparing the chaos design with the design of an oblivious router, it was found that the bandwidth of the routers will be limited by the speed of the pads in either case.

The low-load performance of both chaotic and oblivious routers should be about equal, although oblivious routing will achieve slightly lower latency. As the load increases, though, the advantages of chaotic adaptive routing will become more apparent and chaotic routing will perform better.

A pipelined design for chaotic routing is presented. In this design, only one new path may be set up through the router per cycle. However, this design does not limit the throughput of the router because so few of the flits transmitted are header flits. A design is proposed for the multiqueue which manages the necessarily complex decisions with reasonable cost. Since this design may cause the router to slow down, its performance is evaluated. It is shown that, for a multiqueue which can hold five twenty-flit packets, there should be no significant performance loss. For smaller queues or longer packets, however, there may be a performance loss due to the complexity of the multiqueue.

Altogether, a design is presented in this chapter that meets the performance goals of low latency and high throughput, with manageable complexity.

# Chapter 4

# SIMULATION STUDIES

Although the comparative analysis in Chapter 2 indicates that chaotic routing should be as good or better than oblivious routing, it is very difficult to gauge the performance of the routers based solely on design observations. To better estimate the performance of chaotic routing, a network simulator has been developed. This simulator uses a detailed model of chaotic and other routing schemes to produce simulated results for networks of varying size and topology with differing traffic patterns. The results indicate that chaotic routing is very competitive as a routing technology.

## *4.1 Routers Compared*

As motivated by the observations of Chapter 2, chaotic routing is compared with oblivious and deflection routing. Oblivious routing is chosen as the "baseline" routing scheme for two reasons. The first is oblivious routing's solid position as the "algorithm of choice" in almost all current multicomputers. The second reason is the fact that oblivious routing achieves good results at a modest cost, providing a good baseline to compare against. Deflection routing is also compared against chaotic routing because it has been chosen by some manufacturers to be used in their multicomputers. Also, deflection routing is the logical extension of chaotic routing to a model with a queue size of zero. The specific implementations of the chaos, oblivious, and deflection routers studied are described below.

Each router is configured in up to three arrangements, the two-dimensional mesh and torus and hypercube topologies. The two-dimensional router designs are constant with respect to network size, while the hypercube router designs change somewhat as the network grows larger. In the experiments reported here, each router is able to make a routing decision in a single cycle. The simulator has been updated more recently to reflect the proper times to make a routing decision, but results are not currently available for all experiments. The net effect of this is that the latency numbers are slightly smaller than they would be if the proper delays had been used.

Since the oblivious router can make its decisions faster than chaotic routing (three cycles instead of four cycles, as discussed in Chapter 3), the actual low-load latency numbers for oblivious routing will be slightly smaller than those for chaotic routing. To compare the old results with the new results, re-simulation has been done for a single experiment, 256-node chaos vs. oblivious torus, and the results are shown in Figure 4.9. See Section 4.6.1 for discussion of the differences.

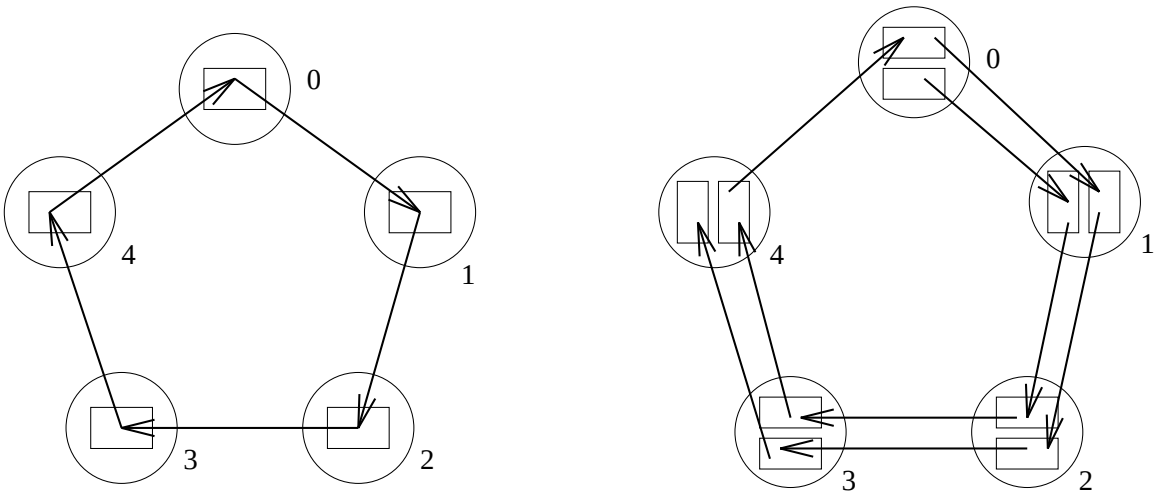
### *4.1.1 Chaotic Routing*

The chaotic router design used for these studies is based on the design presented in Chapters 2 and 3. The basic design shown in Figure 2.5 is that used for the simulated mesh and torus chaotic routing. For the hypercube router, the only difference is that the number of channels and input/output frames is dependent on the network size (specifically, there are $\log_2 N$ network channels in an $N$-node hypercube router). The queue size used varies based on the particular simulation study.

### *4.1.2 Oblivious Routing*

Oblivious routers found in the literature and in implementations vary widely in design, primarily with respect to flow control issues. The amount of buffering and the method of flow control in an oblivious router can affect performance greatly [Dally 92]. The oblivious router chosen for the simulation studies is based on a virtual cut-through design [Kermani & Kleinrock 79] and has a structure like that shown in Figure 2.4. This oblivious router has the same datapath structures as a comparable chaotic router, except that the multiqueue and associated data paths are not present. The design implements the "core" functions of the chaotic router, with the modification that packets must be routed on dimension-order minimal paths. This design has, compared to other oblivious routers, a relatively large amount of buffering which allows it to achieve very good performance.

Although the dimension-order deadlock prevention scheme is sufficient for mesh and hypercube oblivious routers, it will not prevent deadlock in torus networks due to the presence of cyclic dependencies within a single dimension. To address this problem, the virtual channel deadlock prevention scheme of Dally and Seitz is used [Dally & Seitz 87]. In this scheme, which requires splitting each physical channel into two virtual channels by the addition of buffers and multiplexors, cyclic paths




(a) Deadlock Configuration

(b) Virtual Channel Solution

Figure 4.1: Virtual channel deadlock prevention scheme

in the network are broken into separate paths. Figure 4.1 shows an example of this scheme. In Figure 4.1a, a deadlock configuration consisting of packets in five nodes is shown. By dividing each physical channel into two virtual channels and connecting as shown in Figure 4.1b, the cycle is broken and deadlock cannot occur. In this scheme, messages are injected into the "inner" set of virtual channels and use the "outer" set only after crossing the link from node four to node zero.

There are two drawbacks to this deadlock prevention scheme. The first is that the implementation of virtual channels is not without cost. In the simulated oblivious torus router, each virtual channel is implemented using a complete input/output frame set, thus doubling the amount of buffering needed. However, since the chaos router requires additional hardware to implement the multiqueue, the comparison remains fair. The second drawback is due to uneven buffer utilization. Essentially, because the scheme designates certain network links as "special," such as the link between nodes four and zero in the example, the uniformity of the network is upset. This has been shown to decrease significantly the overall network performance [Adve & Vernon 93, Bolding 92].

### *4.1.3 Deflection Routing*

Deflection, or "hot-potato" routing, is a non-minimal adaptive scheme which is similar to chaotic routing without the central multiqueue storage unit. Even though the latency of packets in deflection routers is significantly greater than the latency in chaotic routers, the routers are compared to gauge the throughput differences.

The deflection router modeled in these simulations has datapaths similar to those of the oblivious router described above. Packets arrive on input links and are routed across a crossbar to output frames. However, there are several constraints of deflection routing that significantly affect the model. In deflection routing, packets must all arrive simultaneously and be assigned output channels at the same time. In order to ensure that all packet headers arrive at the same time, the packets must all be sent in a store-and-forward fashion, without the ability to use virtual cut-through switching. Also, in order for all packets to be able to be routed out of the node at the same time, the inter-node channels cannot be shared bi-directionally, since the output channel must be free in both directions at the same time. Finally, since all packets must be routed out at the same time, the deflection router computes all routes simultaneously, according to the "priority" algorithm of Fang and Szymanski [Fang & Szymanski 91], as opposed to the "one-at-a-time" routing computation of chaotic routers.

## *4.2 Traffic Models*

In simulating the various routers, it is necessary to apply some sort of message traffic to the simulated network. Since traffic observed in multicomputers varies greatly among different applications, it would be ideal to simulate a set of benchmark programs and draw conclusions from the observed results. However, this is very difficult not only because of the typical problems with benchmarks of any sort, that is, that the benchmarks may or may not give results which are meaningful, but also because of the difficulty in simulating the necessary traffic patterns.

For the simulations presented here, two synthetic workloads are used instead. The first is *uniform random* traffic. In this workload, each node in the network injects packets into the network at random intervals determined by the applied loading factor. The destinations are chosen from a uniform random distribution of all of the nodes in the network. Although uniform random traffic may not represent any actual

workload, it is useful because it is the "natural" workload thought of, and it is the standard workload used in comparing router performance.

The applied load to the network is expressed in terms of the maximum injection rate possible considering the bisection bandwidth limited maximum throughput. This is determined by Equations 1.6–1.9.

The second synthetic workload is "hot spot" traffic. This workload is motivated by the observation that, in multicomputer networks, it is very common for a few nodes to have a disproportionately high amount of traffic destined to them [Pfister & Norton 85]. These "hot spot" nodes may contain locks or other commonly referenced variables. To emulate this network behavior, a workload with several hot spots is used. This workload is the same as the uniform random workload, except that ten randomly selected "hot" nodes are the destination for four times as many packets as the rest of the nodes. The total load on the network is still specified as a fraction of the bisection bandwidth limited injection rate, although, since the traffic is no longer uniformly random, this limit may be slightly inaccurate.

A difficulty arises in some networks with hot spot traffic: the "hot" nodes cannot deliver packets across their processor channel fast enough to avoid a backlog of packets. In many networks this is not a problem because the bisection bandwidth limits the number of packets destined to hot nodes to be within the processor channel's bandwidth. However, if the network has a high enough relative bisection bandwidth, this will be a problem. Specifically, if four times the maximum delivery rate (which is the same as the maximum injection rate, *MaxInj*) is greater than one flit/cycle, than four times hot spot traffic will be limited by the delivery channel bandwidth instead of the network bandwidth.

In the networks simulated, this occurs for all hypercube networks, since their bandwidth allows an injection rate of one flit/cycle, and for 64-node torus networks. Since the goal of these simulations is to test *network* behavior, not the processor channel bandwidth, the processor channel's delivery rate is increased by a factor of four in these simulations so that the processor channel will no longer be a bottleneck.

## *4.3 Simulation Methodology*

Simulations for the networks and routers studied were conducted using a flit-based simulator written in C. The simulations are based on the *cycle* time unit. One cycle

is the time necessary to transmit a single flit across a channel. In the simulations, routing decisions can be made in a single cycle, although, as discussed in Section 4.1, this is unrealistic. Thus, if a message header enters a router at cycle $t$, it may enter the next router as early as cycle $t + 1$.[1]

Simulations were run by applying the simulated load to the network in a continuous manner. Statistics were computed in intervals in which each node of the network has injected at least 50 messages. Average throughput and average latency were computed for each statistics interval and convergence was determined when the standard deviation of both of the measures over the most recent five intervals were less than 3%. The results presented here represent the averages and standard deviations of 3 to 5 runs.

The statistics reported here are the average throughput of the network normalized to the maximum throughput under uniform random load and the average latency of messages in cycles. Latency is defined as the time from presentation of a message to the network until the message has been completely removed from the network at its destination. (Notice that this does *not* include source queueing time.)

### *4.4 MultiQueue Size Experiments*

The first set of experiments measure the performance of chaotic routing with various multiqueue sizes. The expected results are that throughput should increase as the multiqueue is made larger, as adding buffering to a network almost always increases the capacity of the network to delivery messages. This happens because the additional buffering allows packets which have resources available to them to pass other packets which are waiting on some resource to become free. At the same time, it is expected that latency may increase as the queue size increases, as packets will wait in the multiqueue for longer periods of time before being derouted around congestion since it can hold more packets and will not fill up as quickly.

The following experiments concentrate on 256-node mesh and torus networks. The motivation for choosing these networks is that this is the target network for the chaos router prototype chip (see Chapter 5). Results for these networks may not generalize to networks of all topologies or sizes. The primary results of concern are

[1] For the deflection router, since the entire message must be received before transmission can begin, a message entering a router at cycle $t$ will not leave until cycle $t + L$.

Table 4.1: 256-node mesh throughput as a function of queue size (Uniform traffic)

| Queue Size | Percentage of Maximum Load | | | | |
|---|---|---|---|---|---|
| | 80 | 85 | 90 | 95 | 100 |
| 4 | 75.23 | 77.10 | 78.20 | 80.50 | 80.00 |
| 5 | 79.87 | 83.50 | 86.47 | 88.80 | 90.10 |
| 6 | 80.07 | 83.77 | 86.20 | 88.57 | 90.37 |
| 7 | 80.13 | 83.87 | 86.87 | 88.63 | 90.20 |
| 10 | 79.97 | 83.93 | 86.30 | 88.80 | 90.23 |

Table 4.2: 256-node mesh latency as a function of queue size (Uniform traffic)

| Queue Size | Percentage of Maximum Load | | | | |
|---|---|---|---|---|---|
| | 80 | 85 | 90 | 95 | 100 |
| 4 | 643.56 | 743.64 | 843.04 | 885.45 | 950.75 |
| 5 | 241.20 | 315.33 | 388.12 | 450.03 | 500.86 |
| 6 | 235.70 | 302.26 | 369.65 | 416.54 | 466.22 |
| 7 | 234.53 | 297.08 | 357.79 | 423.68 | 470.06 |
| 10 | 238.37 | 297.65 | 365.54 | 421.93 | 475.92 |

the throughput and latency at relatively high loads. At lower loads, very few packets will enter the multiqueue at all, and its size is irrelevant.

The queue size results for 256-node mesh network with uniform random traffic are shown in Tables 4.1 and 4.2. The throughput results show a remarkable improvement in performance between a queue size of four and a queue size of five. On the other hand, there is very little improvement as the queue is made larger than five queue slots. When the latency results are examined, a similar trend is seen. With a queue size of five packets, the latency is roughly half that seen when the queue hold only four packets. There is a slight improvement in latency as the queue size is made larger than five, although, at high loads, the latency increases slightly as the queue size grows.

Under hot spot traffic, the results are similar. From Tables 4.3 and 4.4, it can be seen that, once again, a queue of size five produces much better performance than a

Table 4.3: 256-node mesh throughput as a function of queue size (Hot spot traffic)

| Queue Size | Percentage of Maximum Load | | | | |
|---|---|---|---|---|---|
| | 80 | 85 | 90 | 95 | 100 |
| 4 | 73.53 | 77.20 | 77.87 | 78.80 | 76.77 |
| 5 | 80.10 | 84.27 | 86.57 | 88.77 | 89.73 |
| 6 | 80.00 | 84.07 | 87.03 | 88.70 | 90.10 |
| 7 | 80.10 | 83.97 | 87.20 | 88.67 | 90.87 |
| 10 | 80.00 | 84.10 | 86.97 | 88.93 | 90.67 |

Table 4.4: 256-node mesh latency as a function of queue size (Hot spot traffic)

| Queue Size | Percentage of Maximum Load | | | | |
|---|---|---|---|---|---|
| | 80 | 85 | 90 | 95 | 100 |
| 4 | 703.44 | 775.12 | 866.23 | 956.71 | 1039.53 |
| 5 | 232.76 | 308.61 | 393.29 | 463.72 | 515.29 |
| 6 | 235.45 | 301.67 | 361.43 | 440.53 | 478.83 |
| 7 | 232.79 | 305.52 | 360.19 | 436.53 | 476.67 |
| 10 | 236.96 | 301.39 | 373.75 | 416.57 | 486.55 |

Table 4.5: 256-node torus throughput as a function of queue size (Uniform traffic)

| Queue Size | Percentage of Maximum Load | | | | |
|---|---|---|---|---|---|
| | 85 | 90 | 93 | 96 | 100 |
| 4 | 85.33 | 89.73 | 92.70 | 95.60 | 94.87 |
| 5 | 85.27 | 89.70 | 92.83 | 96.03 | 96.67 |
| 6 | 85.17 | 89.90 | 93.03 | 95.90 | 96.90 |
| 7 | 85.13 | 89.87 | 92.67 | 95.60 | 96.77 |
| 10 | 85.20 | 89.87 | 92.67 | 96.03 | 97.37 |

Table 4.6: 256-node torus latency as a function of queue size (Uniform traffic)

| Queue Size | Percentage of Maximum Load | | | | |
|---|---|---|---|---|---|
| | 85 | 90 | 93 | 96 | 100 |
| 4 | 215.90 | 254.40 | 328.51 | 423.44 | 531.77 |
| 5 | 213.77 | 261.58 | 311.01 | 383.42 | 468.00 |
| 6 | 215.24 | 255.55 | 309.25 | 385.51 | 456.32 |
| 7 | 213.88 | 256.82 | 312.10 | 388.63 | 465.67 |
| 10 | 212.10 | 261.09 | 308.02 | 387.77 | 459.35 |

queue of size four. The throughput reaches almost its peak with a queue size of five, with very little improvement as it is made larger. On the other hand, the latency numbers seem to reach very close to their minimum with a queue size of six packets. However, since the results are almost as good with queue of five packets, the most efficient size is probably five packets.

For 256-node torus networks with uniform random traffic, the result are presented in Tables 4.5 and 4.6. In this case, almost all sizes with four or more packets achieved near-optimal throughput. As with the mesh results, though, a size of five packets is better than a size of four packets, but the advantages of six or more packets are very limited. The latency results show the same trend as the throughput results. Thus, with random traffic, a queue which holds five packets is probably optimal.

When hot spots are added to the torus traffic, the advantages of the larger queue sizes are more apparent. The throughput, shown in Table 4.7, shows improvement all

Table 4.7: 256-node torus throughput as a function of queue size (Hot spot traffic)

| Queue Size | Percentage of Maximum Load | | | | |
|---|---|---|---|---|---|
| | 85 | 90 | 93 | 96 | 100 |
| 4 | 85.30 | 88.62 | 90.38 | 89.10 | 89.38 |
| 5 | 85.13 | 89.38 | 91.62 | 91.38 | 91.92 |
| 6 | 85.03 | 88.88 | 91.93 | 92.30 | 93.47 |
| 7 | 85.08 | 89.43 | 91.85 | 92.60 | 93.85 |
| 10 | 85.17 | 89.95 | 91.70 | 93.80 | 95.13 |

Table 4.8: 256-node torus latency as a function of queue size (Hot spot traffic)

| Queue Size | Percentage of Maximum Load | | | | |
|---|---|---|---|---|---|
| | 85 | 90 | 93 | 96 | 100 |
| 4 | 244.22 | 323.77 | 445.79 | 546.70 | 571.29 |
| 5 | 239.71 | 298.41 | 354.28 | 447.71 | 486.96 |
| 6 | 237.83 | 292.50 | 344.75 | 423.25 | 463.60 |
| 7 | 235.22 | 285.06 | 343.99 | 406.99 | 469.92 |
| 10 | 231.61 | 282.50 | 344.26 | 409.99 | 464.98 |

they way up to a queue size of ten. However, due to the consistently good performance of chaotic routing with queues of four or more packets, the difference between a router with queue size four and one with a queue of ten is relatively small. The latency results, shown in Table 4.8, show improving results through a queue of size six, and then little or no improvement.

The result of the mesh and torus queue sizing experiments is that a queue of size five is considerably better than a queue of size four, but adding more queue slots does not improve performance very much. Since there is considerable expense in adding each queue slot, the "best" chaotic router for 256-node mesh and torus networks should contain five queue slots.

For other size mesh and torus networks, a queue size of five will probably give good results as well. This is because, at saturation, the routers will experience about the same load regardless of the size of the network, and similar results should be expected. On the other hand, for hypercube and other topologies, it is not clear what size the queue should be made. Some limited experiments have suggested that a queue size equal to the degree of the node (including the processor channel) may be the minimum size necessary to give good performance. This may be due to the equalization of the number of queue slots to the number of channels, which allows one packet from each channel to enter the multiqueue without causing derouting. For this reason, further experiments using hypercubic networks use a queue of size $d + 1$ for a $d$-dimensional hypercube.

## *4.5 Shared vs. Non-shared Channels Experiments*

As discussed in Section 3.5, there are tradeoffs between implementing the network channels as two sets of uni-directional wires or as a single set of bi-directional wires. To avoid confusion in terminology, these will be referred to as shared (bi-directional) and non-shared (uni-directional) channels in the remainder of this dissertation. Although the analysis in Section 3.5 shows that the low-load performance of shared channels should be superior to that of non-shared channels, it is not clear how the performance will compare at higher loadings. The following experiments examine the differences.

For these experiments, chaotic routing with a queue of size five is used. For shared channels, packets contain 20 flits of data. Since, with the same number of wires, flits on non-shared channel networks must be half as wide as for shared channel networks,

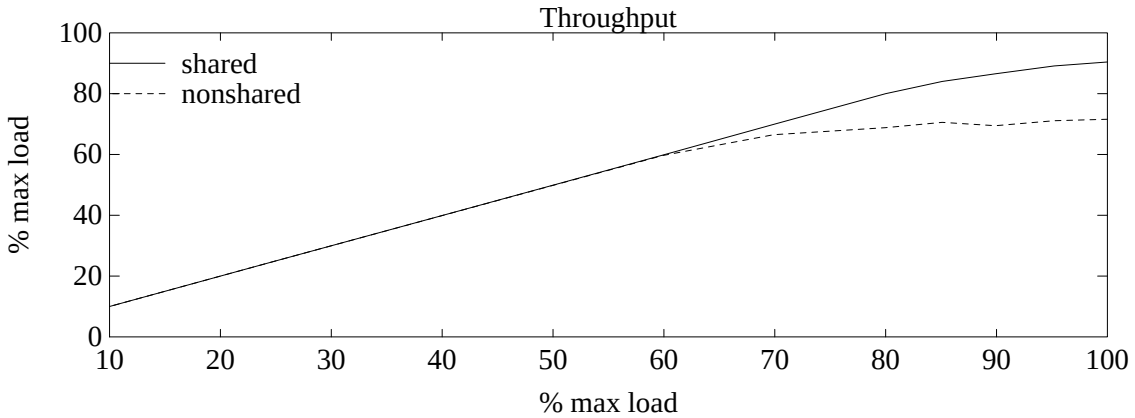
the number of flits in the non-shared channel experiments is increased to 40 so that all packets contain the same amount of data. As with the multiqueue size experiments, the results presented concentrate on 256-node mesh and torus networks.

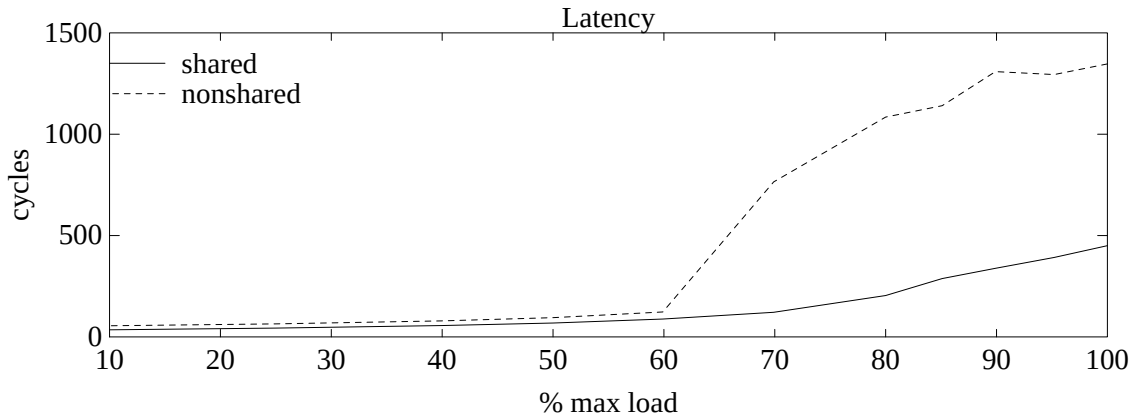
Figures 4.2 and 4.3 show the results for 256-node mesh networks with uniform random and hot spot traffic. Both of these experiments show very similar results, with shared channels providing consistently higher throughput and lower latency at all loads. Figures 4.4 and 4.5 show the same results for torus networks. Once again, the trends are the same, with throughput always higher and latency always lower with shared channels. However, the effect of hot spots on the results is much more magnified in the torus networks.

The lower latency, especially at low loads, is expected due to the shorter packet length (in flits) which reduce the minimum transmission time. However, the much lower latency and better throughput that the shared channel network exhibits at the higher loads is not as expected. There are two possible reasons for the reduced throughput and higher latency due to the non-shared channels. The first has to do with the multiqueue's operation.

In Section 3.6.1, it is shown that the multiqueue's performance is dependent on the queue size and the packet length. In the chaos router simulated in these experiments, the selection of packets to deroute is limited to routable packets (the first method described in Section 3.6.1). As can be seen from Table 3.1, with a queue size of five and packets of 20 flits, as is the case with shared channels, there will be at least two packets to choose from whenever derouting from the multiqueue. However, when the packet length is increase to 40 flits, in the worst case, the router must stall fifteen cycles before selecting a packet to deroute from the multiqueue. This delay will significantly affect both the latency of messages and the throughput of the network. As the amount of derouting increases, as is the case with hot spots, the effect of this reduction in performance will become more apparent.

The second reason for decreased performance with non-shared channels is the lower internal bandwidth of the router. Even though the external router channels have the same total bandwidth with both shared and non-shared channels, the internal router paths have more bandwidth in the shared case because the data paths are twice as wide (see Figure 3.9). Because of this, the router with non-shared channels will face more bottlenecks inside the router than will the router with shared channels.



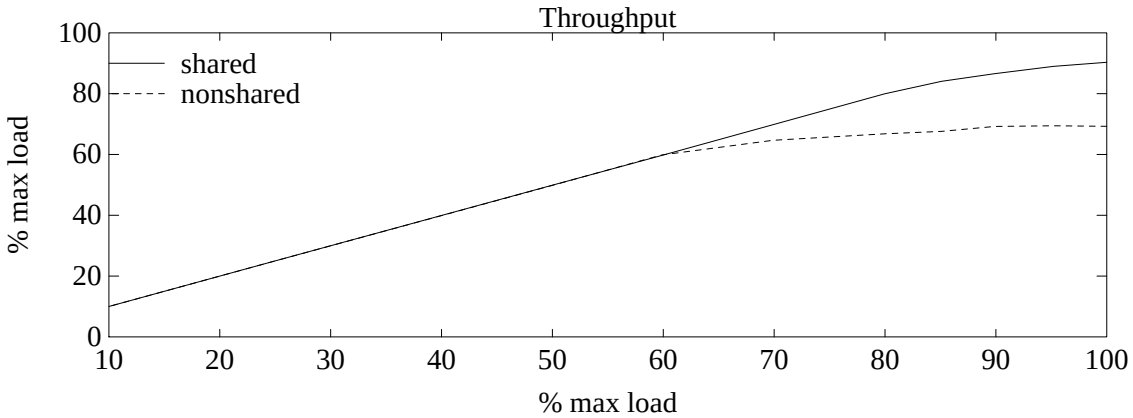

Shared channels

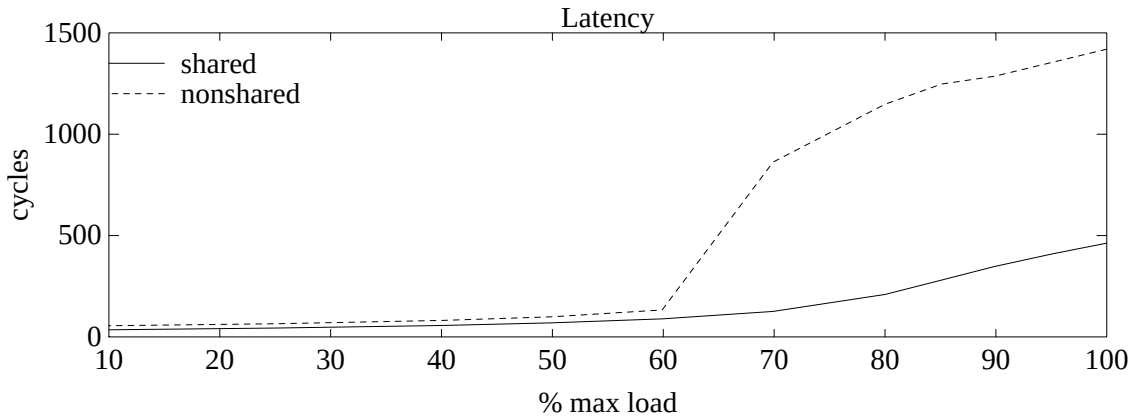
| % load | 10 | 25 | 40 | 50 | 60 | 70 | 80 | 85 | 90 | 95 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| mean xpt | 10.00 | 25.00 | 39.92 | 49.90 | 59.80 | 69.90 | 80.00 | 84.02 | 86.54 | 89.12 | 90.42 |
| std xpt | 0.00 | 0.00 | 0.04 | 0.00 | 0.13 | 0.06 | 0.14 | 0.50 | 0.42 | 0.32 | 0.39 |
| mean lat | 35.01 | 43.51 | 56.10 | 68.50 | 88.11 | 121.4 | 204.1 | 287.8 | 338.4 | 392.0 | 450.1 |
| std lat | 0.02 | 0.12 | 0.27 | 0.77 | 0.59 | 2.96 | 11.81 | 7.41 | 12.59 | 15.42 | 11.87 |

Nonshared channels

| % load | 10 | 25 | 40 | 50 | 60 | 70 | 80 | 85 | 90 | 95 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| mean xpt | 10.00 | 25.00 | 39.90 | 49.84 | 59.66 | 66.50 | 68.82 | 70.56 | 69.46 | 71.10 | 71.58 |
| std xpt | 0.00 | 0.00 | 0.00 | 0.05 | 0.12 | 0.41 | 0.78 | 0.42 | 0.39 | 0.69 | 0.92 |
| mean lat | 54.99 | 64.25 | 79.04 | 95.00 | 123.0 | 764.8 | 1084.5 | 1140.7 | 1308.8 | 1294.3 | 1346.8 |
| std lat | 0.09 | 0.15 | 0.53 | 0.75 | 1.38 | 39.61 | 32.04 | 36.96 | 22.16 | 32.06 | 41.25 |

Figure 4.2: 256-node chaotic mesh: shared vs nonshared (Uniform traffic)



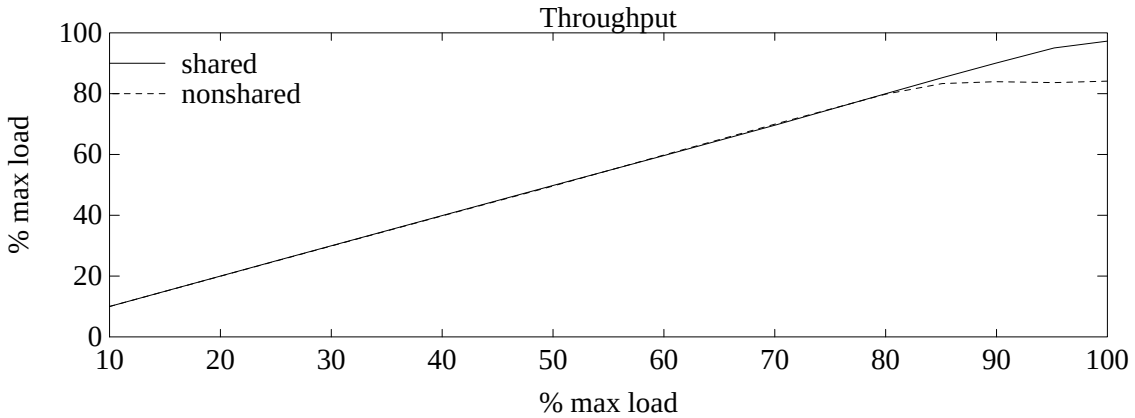

Shared channels

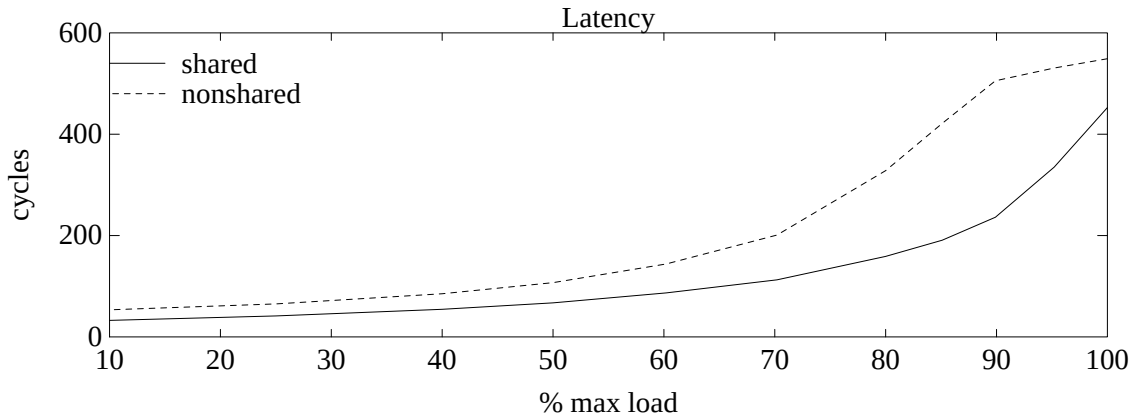
| % load | 10 | 25 | 40 | 50 | 60 | 70 | 80 | 85 | 90 | 95 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| mean xpt | 10.00 | 25.00 | 39.94 | 49.90 | 59.72 | 69.80 | 79.98 | 84.06 | 86.58 | 88.98 | 90.32 |
| std xpt | 0.00 | 0.00 | 0.05 | 0.00 | 0.04 | 0.00 | 0.12 | 0.48 | 0.38 | 0.54 | 0.65 |
| mean lat | 35.07 | 43.71 | 56.29 | 69.47 | 88.99 | 125.6 | 209.4 | 280.3 | 347.6 | 410.8 | 462.5 |
| std lat | 0.04 | 0.16 | 0.44 | 0.55 | 2.14 | 1.95 | 13.81 | 5.85 | 15.35 | 13.42 | 11.05 |

Nonshared channels

| % load | 10 | 25 | 40 | 50 | 60 | 70 | 80 | 85 | 90 | 95 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| mean xpt | 10.00 | 25.00 | 39.90 | 49.84 | 59.86 | 64.66 | 66.80 | 67.60 | 69.22 | 69.44 | 69.26 |
| std xpt | 0.00 | 0.00 | 0.00 | 0.05 | 0.29 | 1.45 | 2.71 | 2.22 | 1.56 | 1.92 | 2.61 |
| mean lat | 55.18 | 64.86 | 81.27 | 99.31 | 133.0 | 862.3 | 1147.8 | 1247.5 | 1286.2 | 1356.1 | 1419.7 |
| std lat | 0.04 | 0.28 | 1.07 | 1.11 | 7.14 | 120.67 | 122.40 | 109.84 | 49.71 | 81.63 | 112.17 |

Figure 4.3: 256-node chaotic mesh: shared vs nonshared (4X hot spot traffic)



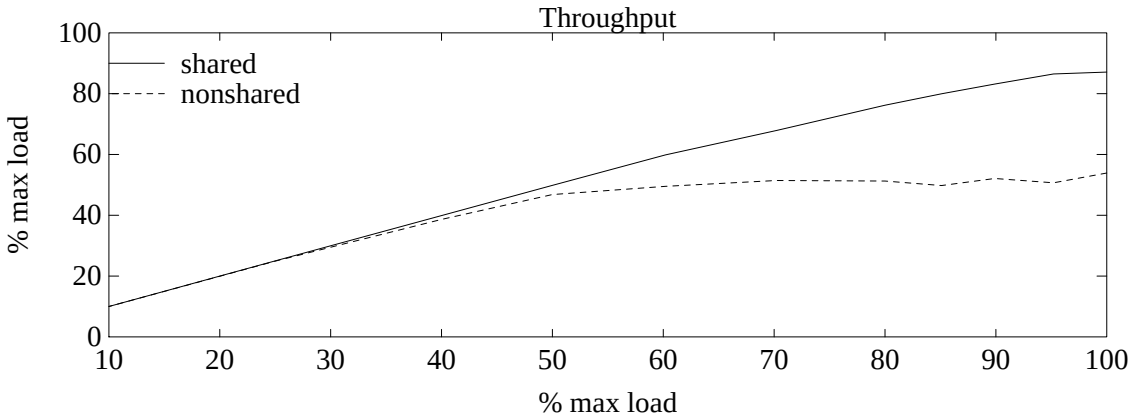

Shared channels

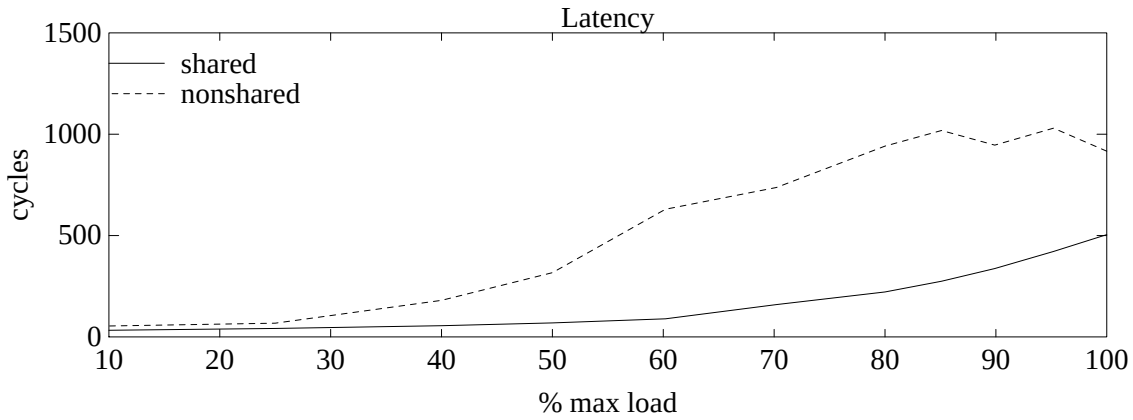
| % load | 10 | 25 | 40 | 50 | 60 | 70 | 80 | 85 | 90 | 95 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| mean xpt | 10.00 | 25.00 | 39.90 | 49.80 | 59.84 | 69.82 | 79.98 | 85.24 | 90.00 | 95.04 | 97.28 |
| std xpt | 0.00 | 0.00 | 0.00 | 0.00 | 0.19 | 0.35 | 0.16 | 0.23 | 0.21 | 0.10 | 1.03 |
| mean lat | 32.60 | 41.41 | 54.49 | 67.21 | 86.82 | 112.6 | 158.8 | 190.8 | 236.1 | 335.0 | 452.6 |
| std lat | 0.09 | 0.08 | 0.49 | 0.68 | 1.47 | 0.57 | 2.40 | 4.22 | 10.76 | 6.39 | 12.81 |

Nonshared channels

| % load | 10 | 25 | 40 | 50 | 60 | 70 | 80 | 85 | 90 | 95 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| mean xpt | 10.00 | 24.94 | 39.80 | 49.62 | 60.02 | 70.16 | 79.84 | 83.32 | 83.94 | 83.64 | 84.12 |
| std xpt | 0.00 | 0.05 | 0.00 | 0.04 | 0.32 | 0.14 | 0.37 | 0.28 | 0.59 | 0.21 | 0.62 |
| mean lat | 53.45 | 65.11 | 85.17 | 107.0 | 143.9 | 201.0 | 328.1 | 421.3 | 505.4 | 530.6 | 548.9 |
| std lat | 0.07 | 0.28 | 0.40 | 1.40 | 2.67 | 1.76 | 5.00 | 15.00 | 5.84 | 11.17 | 4.90 |

Figure 4.4: 256-node chaotic torus: shared vs nonshared (Uniform traffic)




Shared channels

| % load | 10 | 25 | 40 | 50 | 60 | 70 | 80 | 85 | 90 | 95 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| mean xpt | 10.00 | 25.00 | 39.90 | 49.84 | 59.86 | 67.88 | 76.20 | 79.96 | 83.16 | 86.48 | 87.10 |
| std xpt | 0.00 | 0.00 | 0.00 | 0.08 | 0.20 | 4.35 | 7.35 | 10.13 | 12.78 | 14.05 | 12.96 |
| mean lat | 32.70 | 41.73 | 55.31 | 69.05 | 89.25 | 159.4 | 221.8 | 274.7 | 337.2 | 421.6 | 504.5 |
| std lat | 0.10 | 0.16 | 0.50 | 0.97 | 1.77 | 84.49 | 106.18 | 134.05 | 146.58 | 108.28 | 82.81 |

Nonshared channels

| % load | 10 | 25 | 40 | 50 | 60 | 70 | 80 | 85 | 90 | 95 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| mean xpt | 10.00 | 24.92 | 38.64 | 46.82 | 49.52 | 51.44 | 51.26 | 49.78 | 52.10 | 50.68 | 53.90 |
| std xpt | 0.00 | 0.04 | 2.42 | 6.01 | 8.72 | 8.72 | 8.92 | 8.54 | 7.66 | 8.52 | 2.91 |
| mean lat | 53.87 | 67.71 | 180.3 | 316.9 | 629.8 | 737.1 | 941.3 | 1018.3 | 946.2 | 1029.8 | 916.2 |
| std lat | 0.08 | 0.80 | 168.14 | 302.30 | 350.40 | 280.40 | 292.74 | 283.15 | 208.75 | 278.84 | 46.50 |

Figure 4.5: 256-node chaotic torus: shared vs nonshared (4X hot spot traffic)

Both of these factors work together to give the chaos router a large advantage when using shared channels instead of non-shared channels. For this reason, when choosing the configuration for the chaos router chip, bi-directionally shared channels are used despite the extra area and complexity needed to implement them.

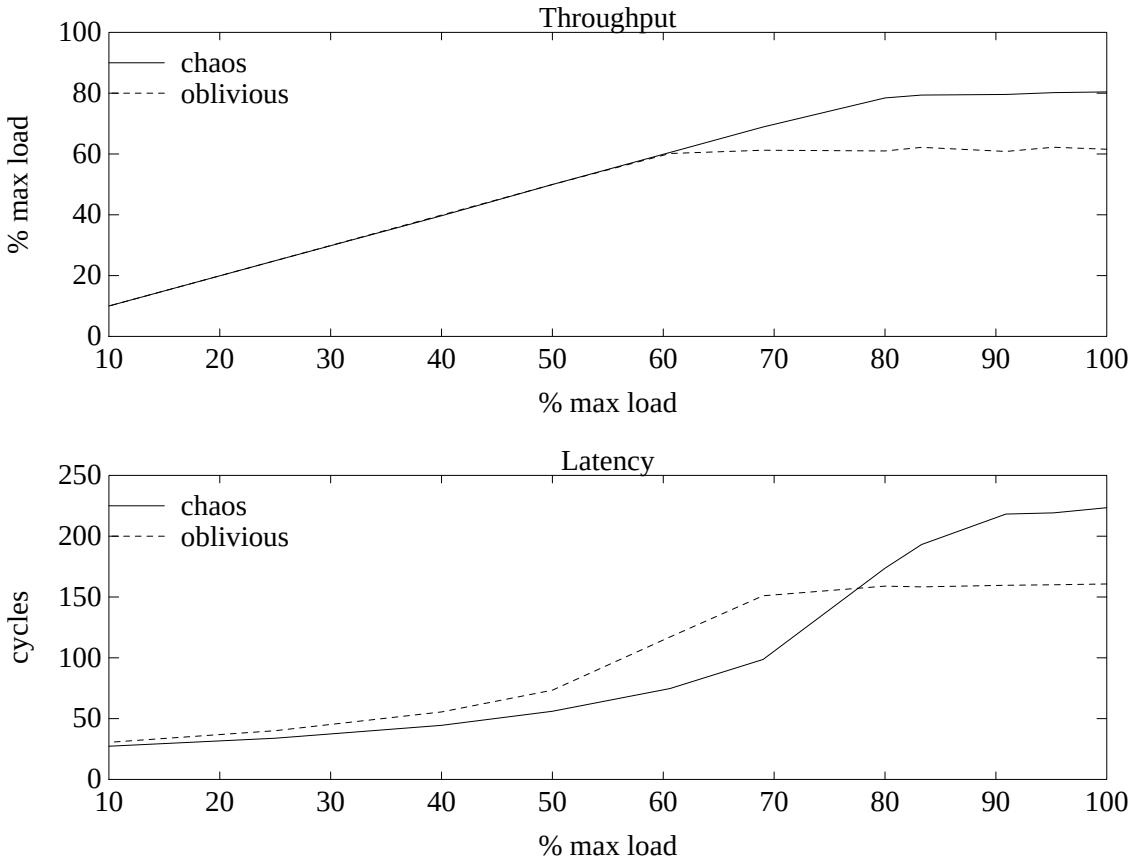
## *4.6 Chaotic vs. Oblivious Routing Experiments*

To compare the performance of chaotic and oblivious routers, experiments were performed on hypercube, mesh, and torus networks of 64, 256, and 1024 nodes using both uniform random and hot spot traffic. In the following sections, only data on 256-node networks is presented. However, results are presented for 64- and 1024-node networks in Appendices A and B.

In the following comparisons, the most important figures are the high-load throughput and medium-load latencies. For low loads, all routing schemes are able to deliver the entire applied load without difficulty. The point at which the network saturates and the network is not able to keep up with the applied load is the interesting point in this case. Also, the shape of the throughput curve above saturation is important – *i.e.* does throughput ever decrease with increasing applied load? Latency is a more critical issue during lower load periods. At loads above saturation, since the network cannot keep up with the load applied, the latency of messages which do get through becomes of only peripheral interest. However, when the network is operating below saturation, it is latency that is the critical figure of merit. Thus, the throughput saturation points and below-saturation latencies are the figures of merit for the networks studied.

### *4.6.1 Uniform Random Traffic*

The results for 256-node hypercube, mesh, and torus networks with uniform random traffic are shown in Figures 4.6 through 4.8. In the hypercube network, chaotic routing achieves considerably higher throughput than oblivious routing, and the latency is less throughout most of the simulated range. As the load becomes very high, though, oblivious routing appears to deliver packets more quickly, as the reported latency is lower. However, at this range, the applied load is much greater than the throughput delivered, meaning that source queueing time must be growing without bound. Thus,



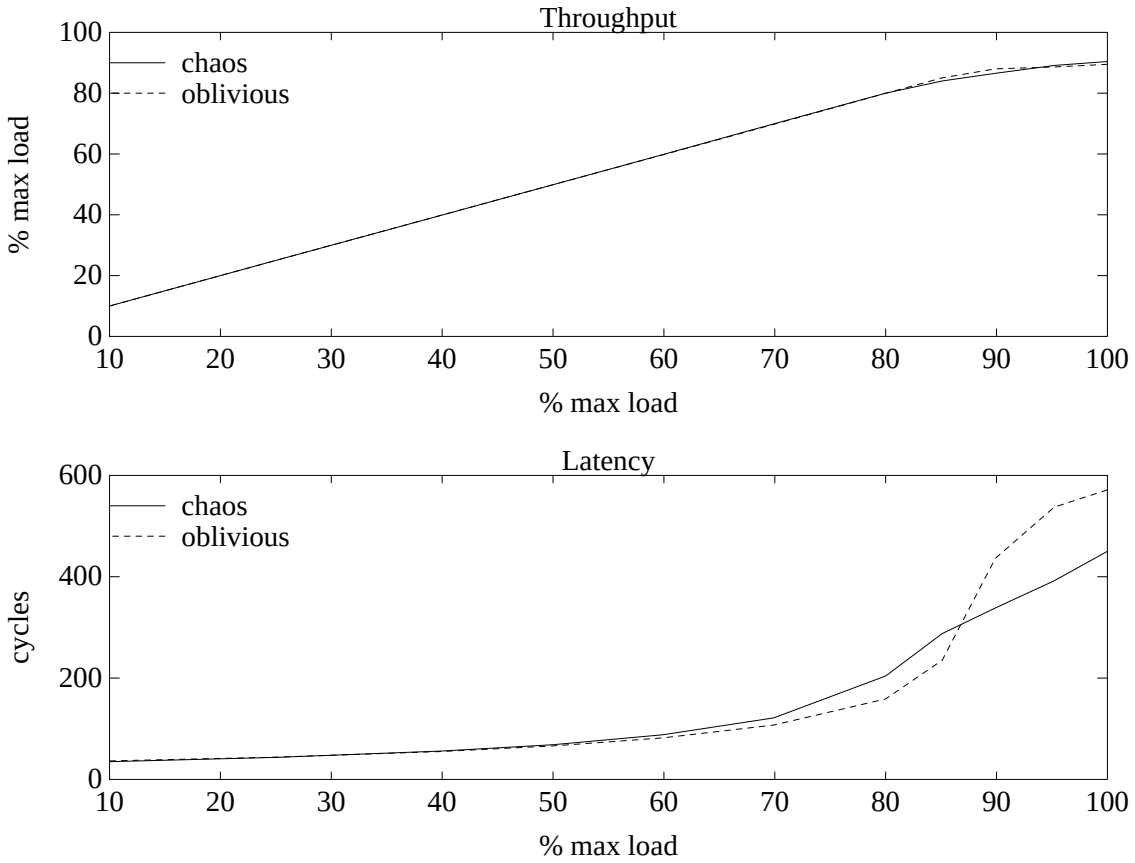

Chaotic Routing

| % load | 10 | 25 | 40 | 50 | 61 | 69 | 80 | 83 | 91 | 95 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| mean xpt | 10.00 | 24.90 | 39.68 | 49.96 | 60.50 | 68.88 | 78.44 | 79.38 | 79.58 | 80.18 | 80.42 |
| std xpt | 0.00 | 0.00 | 0.16 | 0.08 | 0.09 | 0.12 | 0.88 | 0.43 | 0.33 | 1.12 | 0.78 |
| mean lat | 27.30 | 33.88 | 44.48 | 56.05 | 74.71 | 98.63 | 173.68 | 193.15 | 218.19 | 219.19 | 223.43 |
| std lat | 0.05 | 0.07 | 0.25 | 0.39 | 0.55 | 1.73 | 4.38 | 8.15 | 3.55 | 6.06 | 2.33 |

Oblivious Routing

| % load | 10 | 25 | 40 | 50 | 61 | 69 | 80 | 83 | 91 | 95 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| mean xpt | 10.00 | 24.90 | 39.90 | 49.96 | 60.16 | 61.24 | 61.00 | 62.18 | 60.84 | 62.22 | 61.56 |
| std xpt | 0.00 | 0.00 | 0.25 | 0.12 | 0.63 | 0.59 | 1.66 | 0.87 | 0.90 | 0.46 | 1.08 |
| mean lat | 30.50 | 40.07 | 55.50 | 73.38 | 117.12 | 151.02 | 158.94 | 158.33 | 159.64 | 160.06 | 160.71 |
| std lat | 0.07 | 0.20 | 0.28 | 1.40 | 6.44 | 3.64 | 1.34 | 2.40 | 1.29 | 2.02 | 1.61 |

Figure 4.6: 256-node shared channel hypercube: chaotic vs oblivious (Uniform traffic)



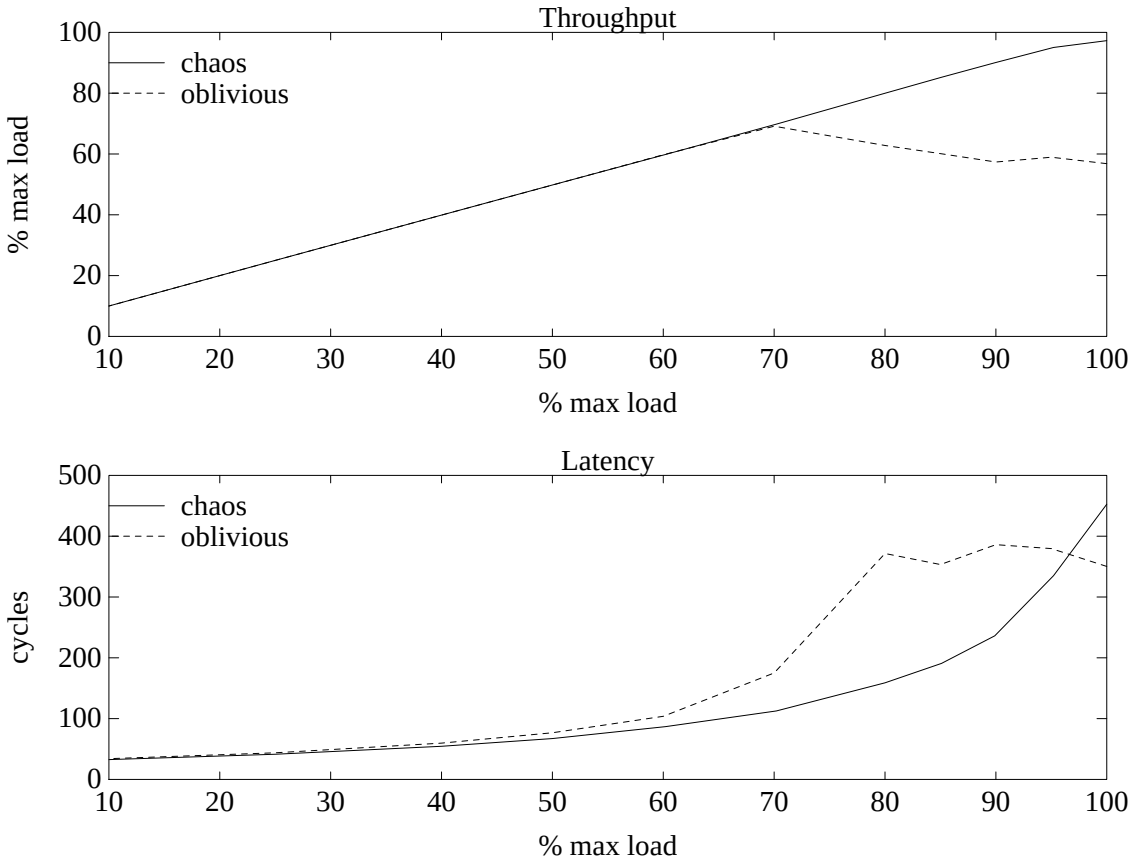

Chaotic Routing

| % load | 10 | 25 | 40 | 50 | 60 | 70 | 80 | 85 | 90 | 95 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| mean xpt | 10.00 | 25.00 | 39.92 | 49.90 | 59.80 | 69.90 | 80.00 | 84.02 | 86.54 | 89.12 | 90.42 |
| std xpt | 0.00 | 0.00 | 0.04 | 0.00 | 0.13 | 0.06 | 0.14 | 0.50 | 0.42 | 0.32 | 0.39 |
| mean lat | 35.01 | 43.51 | 56.10 | 68.50 | 88.11 | 121.41 | 204.12 | 287.84 | 338.42 | 392.01 | 450.16 |
| std lat | 0.02 | 0.12 | 0.27 | 0.77 | 0.59 | 2.96 | 11.81 | 7.41 | 12.59 | 15.42 | 11.87 |

Oblivious Routing

| % load | 10 | 25 | 40 | 50 | 60 | 70 | 80 | 85 | 90 | 95 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| mean xpt | 10.00 | 25.00 | 39.94 | 49.90 | 59.72 | 69.74 | 79.96 | 85.04 | 88.00 | 88.62 | 89.50 |
| std xpt | 0.00 | 0.00 | 0.05 | 0.00 | 0.04 | 0.21 | 0.14 | 0.16 | 0.66 | 1.20 | 0.67 |
| mean lat | 36.54 | 43.92 | 55.16 | 66.08 | 81.98 | 107.29 | 158.84 | 235.17 | 436.44 | 537.33 | 571.43 |
| std lat | 0.09 | 0.10 | 0.21 | 0.51 | 0.72 | 1.75 | 4.47 | 8.50 | 47.99 | 18.90 | 12.16 |

Figure 4.7: 256-node shared channel mesh: chaotic vs oblivious (Uniform traffic)



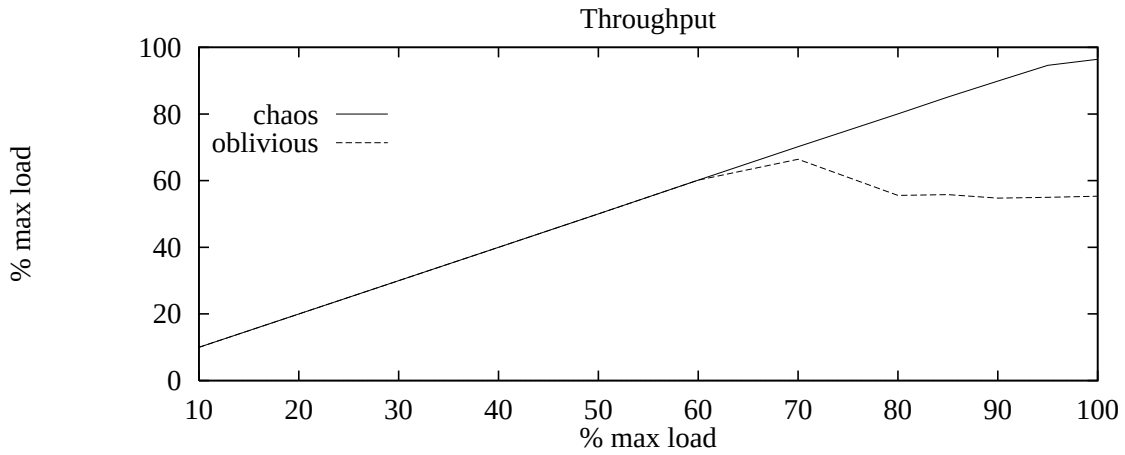

Chaotic Routing

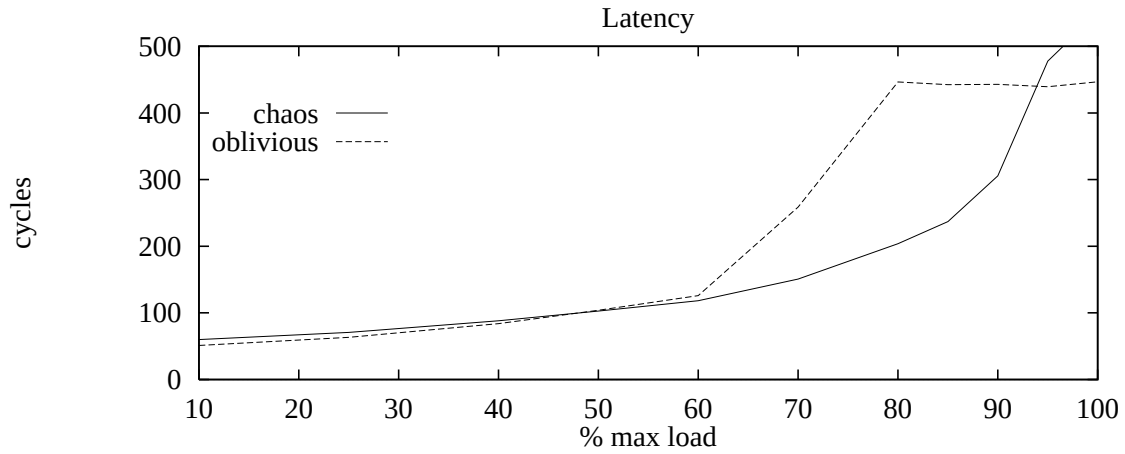
| % load | 10 | 25 | 40 | 50 | 60 | 70 | 80 | 85 | 90 | 95 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| mean xpt | 10.00 | 25.00 | 39.90 | 49.80 | 59.84 | 69.82 | 79.98 | 85.24 | 90.00 | 95.04 | 97.28 |
| std xpt | 0.00 | 0.00 | 0.00 | 0.00 | 0.19 | 0.35 | 0.16 | 0.23 | 0.21 | 0.10 | 1.03 |
| mean lat | 32.60 | 41.41 | 54.49 | 67.21 | 86.82 | 112.63 | 158.89 | 190.83 | 236.16 | 335.06 | 452.56 |
| std lat | 0.09 | 0.08 | 0.49 | 0.68 | 1.47 | 0.57 | 2.40 | 4.22 | 10.76 | 6.39 | 12.81 |

Oblivious Routing

| % load | 10 | 25 | 40 | 50 | 60 | 70 | 80 | 85 | 90 | 95 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| mean xpt | 10.00 | 25.00 | 39.90 | 49.78 | 59.74 | 69.14 | 62.80 | 60.12 | 57.36 | 58.94 | 56.82 |
| std xpt | 0.00 | 0.00 | 0.00 | 0.04 | 0.19 | 0.99 | 2.68 | 4.58 | 2.45 | 2.84 | 1.26 |
| mean lat | 34.11 | 43.78 | 59.69 | 76.75 | 103.82 | 175.07 | 371.54 | 353.40 | 386.08 | 379.63 | 350.29 |
| std lat | 0.03 | 0.07 | 0.56 | 1.14 | 2.03 | 9.58 | 54.28 | 25.24 | 22.03 | 42.25 | 31.39 |

Figure 4.8: 256-node shared channel torus: chaotic vs oblivious (Uniform traffic)




Chaotic Routing

| % load | 10 | 25 | 40 | 50 | 60 | 70 | 80 | 85 | 90 | 95 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| mean xpt | 10.00 | 25.00 | 40.00 | 50.00 | 60.15 | 70.17 | 80.01 | 85.11 | 89.89 | 94.60 | 96.44 |
| mean lat | 60.06 | 70.62 | 88.11 | 102.70 | 118.34 | 150.74 | 203.74 | 237.04 | 305.65 | 477.71 | 549.67 |

Oblivious Routing

| % load | 10 | 25 | 40 | 50 | 60 | 70 | 80 | 85 | 90 | 95 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| mean xpt | 10.00 | 25.00 | 40.00 | 50.00 | 60.15 | 66.43 | 55.58 | 55.79 | 54.76 | 54.97 | 55.35 |
| mean lat | 51.18 | 63.23 | 83.96 | 103.90 | 125.90 | 258.81 | 446.56 | 442.51 | 442.85 | 439.40 | 446.84 |

Figure 4.9: Updated results: 256-node shared channel torus: chaotic vs oblivious (Uniform traffic)

although packets which get into the network are delivered faster, the length of time a packet must wait to get into the network is extremely long, and approaching an infinite length. Overall, for the hypercube network, chaotic routing performs better than oblivious routing.

When a 256-node mesh network is simulated, though, the differences between chaotic and oblivious routing are less apparent. The throughput curves appear to be nearly coincident, with the slightest of variations at loading above 80% of maximum. Oblivious routing achieves slightly better latency during the 60–85% range, and above this chaotic routing performs better. Compared with the hypercube results, these results are very disappointing for chaotic routing. The reasons for the poor relative performance of chaotic routing on the mesh network are due to the presence of a large central "hot spot" in the mesh.

In a mesh-connected network, under uniform random traffic, a disproportionately large number of packets will have shortest paths which lead them through the very center of the network. This results in the creation of a very large permanent hot spot in the center of the network. Figure 4.10 shows a diagram of the average injection delay for a 256-node mesh with chaotic routing. This diagram gives an idea of the amount of congestion in each node of the network. It can be clearly seen that the central nodes of the network are much more congested than the edge nodes. Thus, any packets which travel through the center of the network will be delayed much more than they would otherwise, if they stayed on the "edges" of the mesh.

With chaotic routing, packets which encounter the central hot spot will repeatedly "bounce" off of the "wall" created by the hot spot, wasting bandwidth and increasing their latency. On the other hand, with dimension-order oblivious routing, packets are sent on 'L'-shaped paths due to the X, then Y routing constraints. Since packets will tend to travel on "non-diagonal" paths, they will be less likely to travel through the center and contribute to the large hot spot seen with chaotic routing. Because of this, there will be less congestion in the center, and the bisection channels will be better utilized [Pertel 92a].

Since the central hot spot of the mesh network causes such difficulty for chaotic routing, it is interesting to compare these results with those of torus networks. In a torus-connected network, all nodes are (topologically) indistinguishable, and there should not be, under random traffic, any areas which have more congestion than

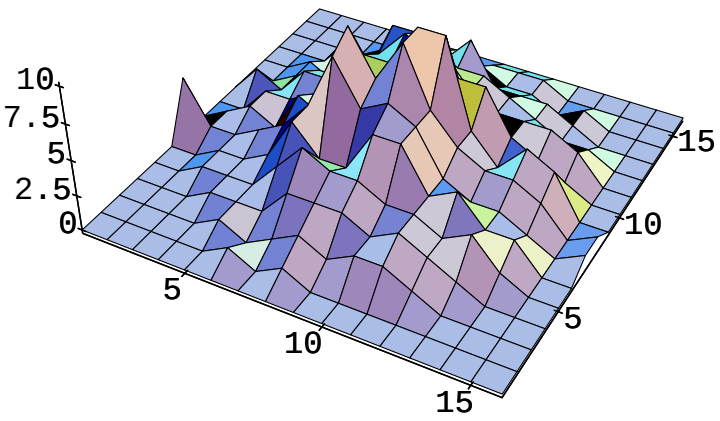



Figure 4.10: Average injection delay for a 256-node mesh with chaotic routing

others. As can be seen in Figure 4.8, chaotic routing performs much better than oblivious routing on a torus network. The uniformity of the torus helps to smooth out the difficulties formed by the mesh hot spot. At the same time, the oblivious router's performance is particularly bad, compared with its performance on the mesh network.

The poor oblivious performance is due primarily to non-uniformities introduced by the deadlock-prevention mechanism. Because different nodes on the network appear to have different amounts of buffering under this scheme, the performance of each oblivious node is dependent upon its location in the network [Bolding 92]. Because of this, drawbacks similar to the mesh hot spot are seen when using a torus oblivious router.

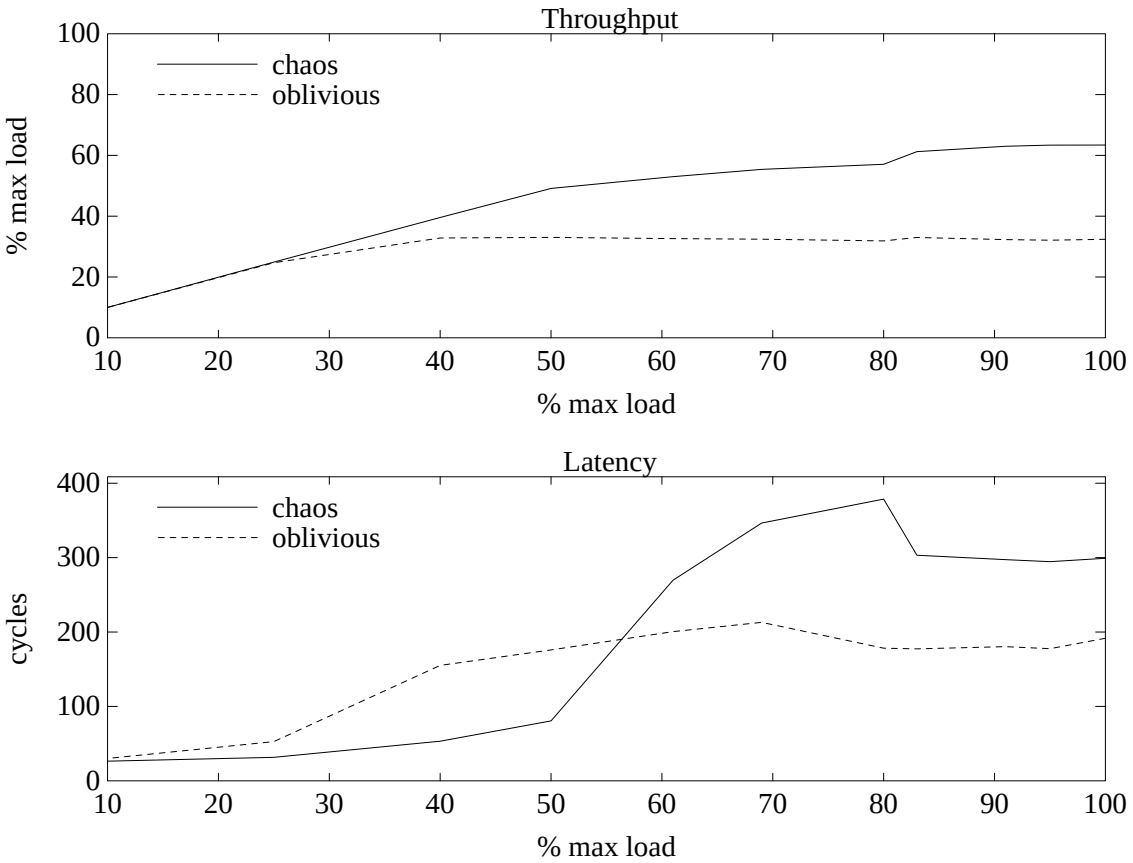
As noted previously, the simulation results presented here use a model which has subsequently been improved. Specifically, the time to make a routing decision is assumed to be one cycle in all networks, when it should be three cycles for an oblivious router and four or more cycles for a chaotic router. Simulations based on a new model incorporating this data are reported for the 256-node torus network in Figure 4.9. As can be seen from the figure, the oblivious router demonstrates slightly lower latencies at low-to-medium loads, but the results are virtually unchanged as the network becomes more loaded.

Overall, with uniform random traffic, chaotic routing does as well or better than oblivious routing. On 256-node hypercube and torus networks, chaotic routing performs much better than oblivious routing. However, on the mesh, performance is not much different than oblivious routing.

### *4.6.2 Hot Spot Traffic*

Figures 4.11–4.13 show the results of simulating hot spot workloads on the hypercube, mesh, and torus networks. In each of these simulations, chaotic routing performs better than oblivious routing.

On the hypercube, both chaotic and oblivious routing perform much more poorly than with a uniform random traffic load. However, the gap between chaotic and oblivious performance grows very wide in this simulation, with oblivious routing reaching only half of the throughput that chaotic routing achieves. It appears that the ability of chaotic routing to route around congested hot spots pays off very well



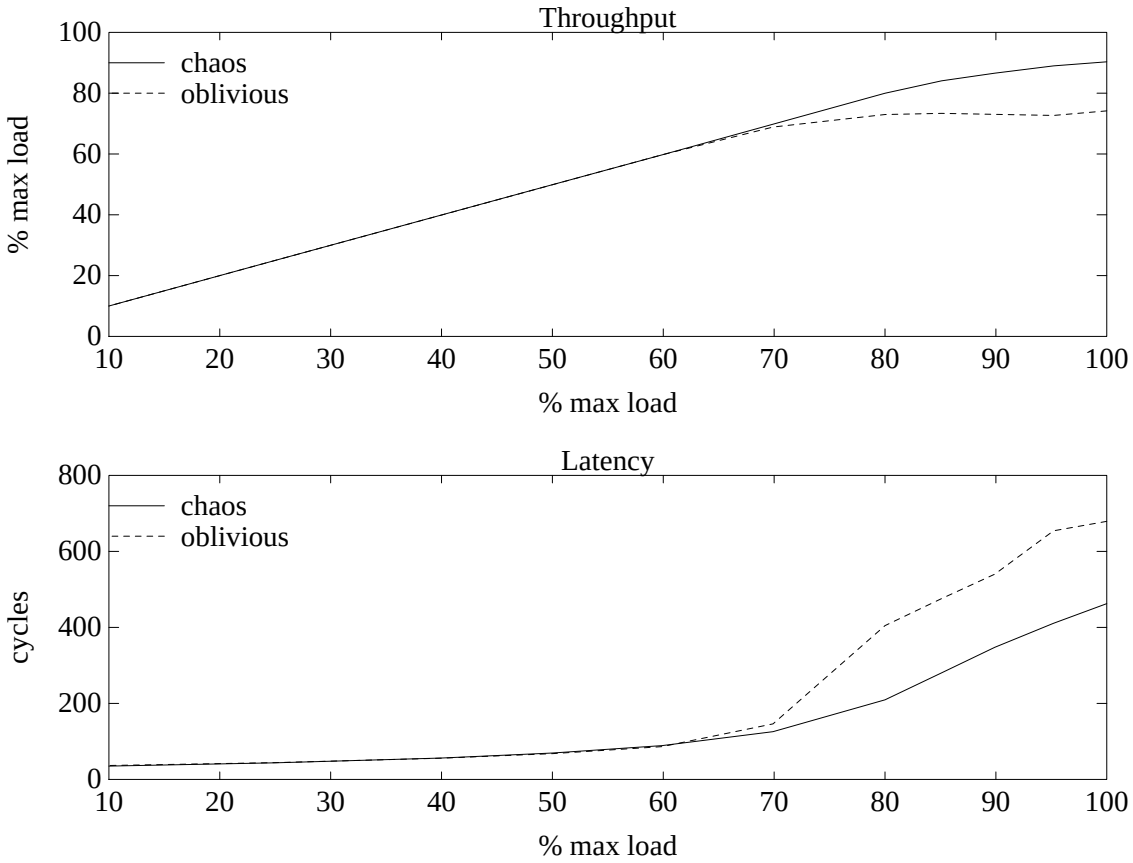

Chaotic Routing (4X delivery rate)

| % load | 10 | 25 | 40 | 50 | 61 | 69 | 80 | 83 | 91 | 95 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| mean xpt | 10.00 | 24.90 | 39.57 | 49.13 | 53.00 | 55.40 | 57.07 | 61.23 | 63.00 | 63.37 | 63.40 |
| std xpt | 0.00 | 0.00 | 0.05 | 1.30 | 9.76 | 13.52 | 12.61 | 6.03 | 3.25 | 4.96 | 4.74 |
| mean lat | 26.40 | 31.57 | 53.24 | 80.54 | 269.61 | 346.51 | 378.66 | 303.20 | 297.30 | 294.57 | 299.11 |
| std lat | 0.01 | 0.08 | 18.71 | 42.60 | 277.09 | 329.32 | 202.58 | 58.39 | 5.43 | 37.80 | 29.15 |

Oblivious Routing (4X delivery rate)

| % load | 10 | 25 | 40 | 50 | 61 | 69 | 80 | 83 | 91 | 95 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| mean xpt | 10.00 | 24.70 | 32.83 | 33.03 | 32.63 | 32.43 | 31.87 | 33.00 | 32.30 | 32.10 | 32.40 |
| std xpt | 0.00 | 0.28 | 4.69 | 4.20 | 3.66 | 4.13 | 2.53 | 2.90 | 4.05 | 4.04 | 3.69 |
| mean lat | 30.07 | 52.72 | 155.14 | 175.93 | 200.54 | 212.98 | 178.28 | 177.35 | 180.36 | 177.59 | 191.62 |
| std lat | 0.08 | 19.47 | 13.43 | 5.61 | 25.21 | 10.98 | 5.60 | 6.37 | 4.96 | 2.08 | 13.72 |

Figure 4.11: 256-node shared channel hypercube: chaotic vs oblivious (4X hot spot traffic)



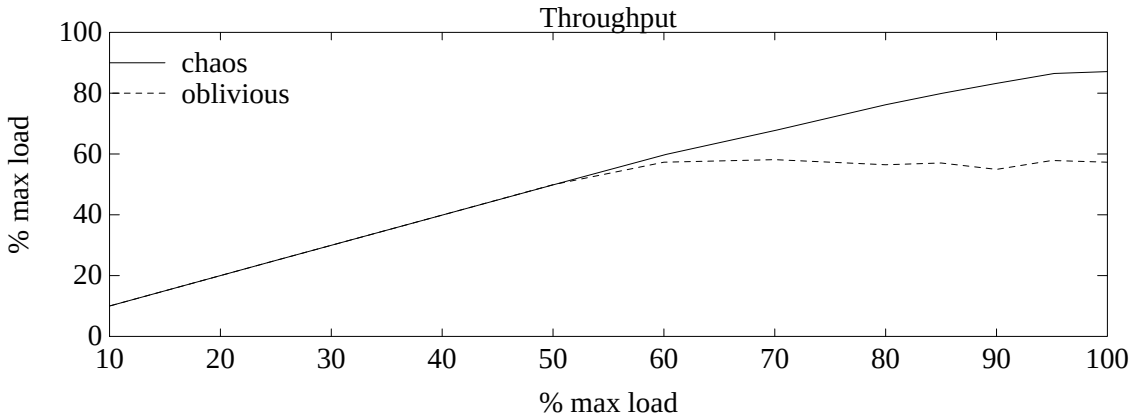

Chaotic Routing

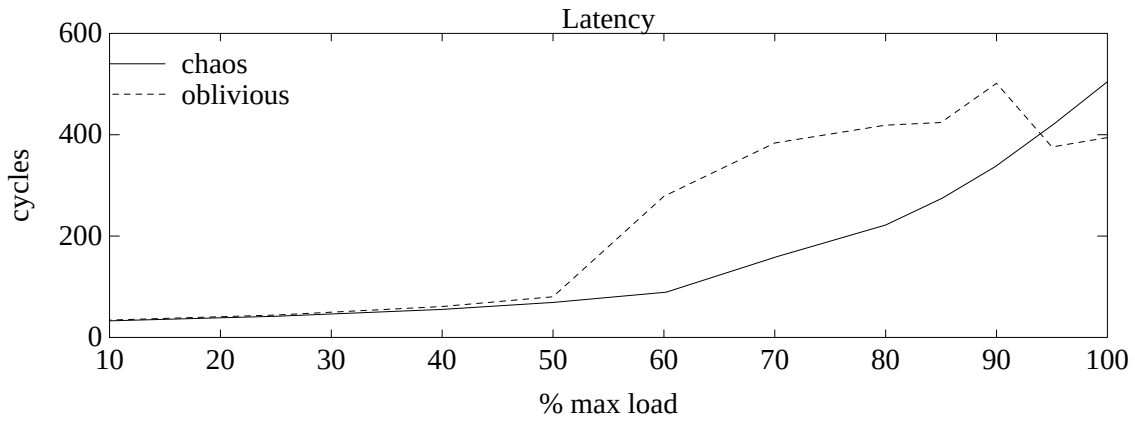
| % load | 10 | 25 | 40 | 50 | 60 | 70 | 80 | 85 | 90 | 95 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| mean xpt | 10.00 | 25.00 | 39.94 | 49.90 | 59.72 | 69.80 | 79.98 | 84.06 | 86.58 | 88.98 | 90.32 |
| std xpt | 0.00 | 0.00 | 0.05 | 0.00 | 0.04 | 0.00 | 0.12 | 0.48 | 0.38 | 0.54 | 0.65 |
| mean lat | 35.07 | 43.71 | 56.29 | 69.47 | 88.99 | 125.68 | 209.44 | 280.39 | 347.62 | 410.86 | 462.55 |
| std lat | 0.04 | 0.16 | 0.44 | 0.55 | 2.14 | 1.95 | 13.81 | 5.85 | 15.35 | 13.42 | 11.05 |

Oblivious Routing

| % load | 10 | 25 | 40 | 50 | 60 | 70 | 80 | 85 | 90 | 95 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| mean xpt | 10.00 | 25.00 | 39.92 | 49.88 | 59.74 | 68.86 | 72.98 | 73.36 | 73.04 | 72.68 | 74.18 |
| std xpt | 0.00 | 0.00 | 0.04 | 0.04 | 0.08 | 2.03 | 3.06 | 3.92 | 3.15 | 4.19 | 4.22 |
| mean lat | 36.65 | 44.20 | 55.89 | 67.85 | 86.33 | 146.17 | 404.74 | 475.77 | 540.07 | 654.20 | 679.07 |
| std lat | 0.09 | 0.10 | 0.32 | 0.97 | 1.65 | 51.82 | 20.18 | 31.20 | 31.55 | 55.79 | 34.17 |

Figure 4.12: 256-node shared channel mesh: chaotic vs oblivious (4X hot spot traffic)




Chaotic Routing

| % load | 10 | 25 | 40 | 50 | 60 | 70 | 80 | 85 | 90 | 95 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| mean xpt | 10.00 | 25.00 | 39.90 | 49.84 | 59.86 | 67.88 | 76.20 | 79.96 | 83.16 | 86.48 | 87.10 |
| std xpt | 0.00 | 0.00 | 0.00 | 0.08 | 0.20 | 4.35 | 7.35 | 10.13 | 12.78 | 14.05 | 12.96 |
| mean lat | 32.70 | 41.73 | 55.31 | 69.05 | 89.25 | 159.46 | 221.82 | 274.74 | 337.27 | 421.60 | 504.57 |
| std lat | 0.10 | 0.16 | 0.50 | 0.97 | 1.77 | 84.49 | 106.18 | 134.05 | 146.58 | 108.28 | 82.81 |

Oblivious Routing

| % load | 10 | 25 | 40 | 50 | 60 | 70 | 80 | 85 | 90 | 95 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| mean xpt | 10.00 | 25.00 | 39.90 | 49.90 | 57.30 | 58.14 | 56.46 | 57.04 | 54.94 | 57.86 | 57.30 |
| std xpt | 0.00 | 0.00 | 0.00 | 0.17 | 3.08 | 2.53 | 0.60 | 1.42 | 2.38 | 2.02 | 0.86 |
| mean lat | 34.19 | 44.21 | 61.00 | 80.25 | 278.57 | 383.71 | 418.91 | 424.15 | 501.58 | 375.87 | 394.41 |
| std lat | 0.06 | 0.22 | 0.22 | 1.22 | 63.39 | 39.07 | 54.33 | 45.40 | 60.84 | 17.15 | 66.11 |

Figure 4.13: 256-node shared channel torus: chaotic vs oblivious (4X hot spot traffic)

in this traffic model. However, at the highly saturated upper end of the loading in this simulation, the latency of packets with chaotic routing is much greater than that of packets with oblivious routing. As noted above, if source queueing is included in the latency measures, both oblivious and chaotic routing have unbounded latencies at this loading range.

When the mesh network is simulated using hot spot traffic, the results swing in favor of chaotic routing also. Chaotic routing achieves both higher throughput and lower latency throughout the loading range. This is due to chaos's good ability to route around local congestion, which makes up for its difficulty in handling the central network-induced mesh hot spot. However, when the network is increased to 1024 nodes (see Figure B.4), the advantage of chaotic routing is lost again. This may be due to two factors. The first is that the effect of ten hot spots on a 1024-node network is smaller than is on a 256-node network. It may be that the large central hot spot overwhelms the ten smaller hot spots in this simulation. The second reason is that since the central hot spot will be larger in a 1024-node network, it may be a more formidable barrier to chaotic routing, and results in decreased performance. Either way, chaotic routing still manages to perform at least as well as oblivious routing in almost all the simulated mesh networks.

On the torus network, chaotic routing once again performs much better than oblivious routing. The performance gap is widened with the introduction of hot spots into the traffic pattern. In fact, chaotic routing gives almost as good results with hot spots as without, while the performance of oblivious routing is significantly reduced by hot spots. This indicates that, on a torus network, chaotic routing does a very good job of allowing packets to detour around congestion without causing much extra delay.

Overall, these experiments have shown that chaotic routing is, in general, superior to oblivious routing. The oblivious router used for comparison was given a significant amount of buffer space, which gives it much better performance than many current oblivious routers [Dally 92]. Even with this, though, chaotic routing almost always achieves equal or better throughput at lower latency.

## 4.7 Chaotic vs. Deflection Routing Experiments

To compare chaotic and deflection routing, mesh and torus networks are simulated with both types of routing using random and hot spot traffic. Because deflection routing requires the use of uni-directional channels, both routers are compared using this configuration. Although this provides a more "fair" basis for comparison, the results of Section 4.5 show that chaotic routing with shared channels significantly outperforms chaotic routing with non-shared channels. Thus, ultimately, the results of deflection routing should be compared against the "shared channel" chaotic results.

Figures 4.14 and 4.15 show the results for 256-node mesh and torus networks. In the mesh network, deflection routing achieves higher throughput than chaotic routing.[2] The latency of deflection routing, however, is extremely poor for low loading. This is due to the store-and-forward constraint of deflection routing. Since chaotic routing can take advantage of virtual cut-through, it achieves much lower latency than deflection routing. On the torus network, chaotic routing gives superior throughput and latency when compared to deflection routing at all loading ranges.

## 4.8 Summary

Many simulations were performed to help in designing the best chaotic router. It was found that the best chaotic performance occurred with a queue size of five packets (for a 256-node mesh or torus router) and with bi-directionally shared channels.

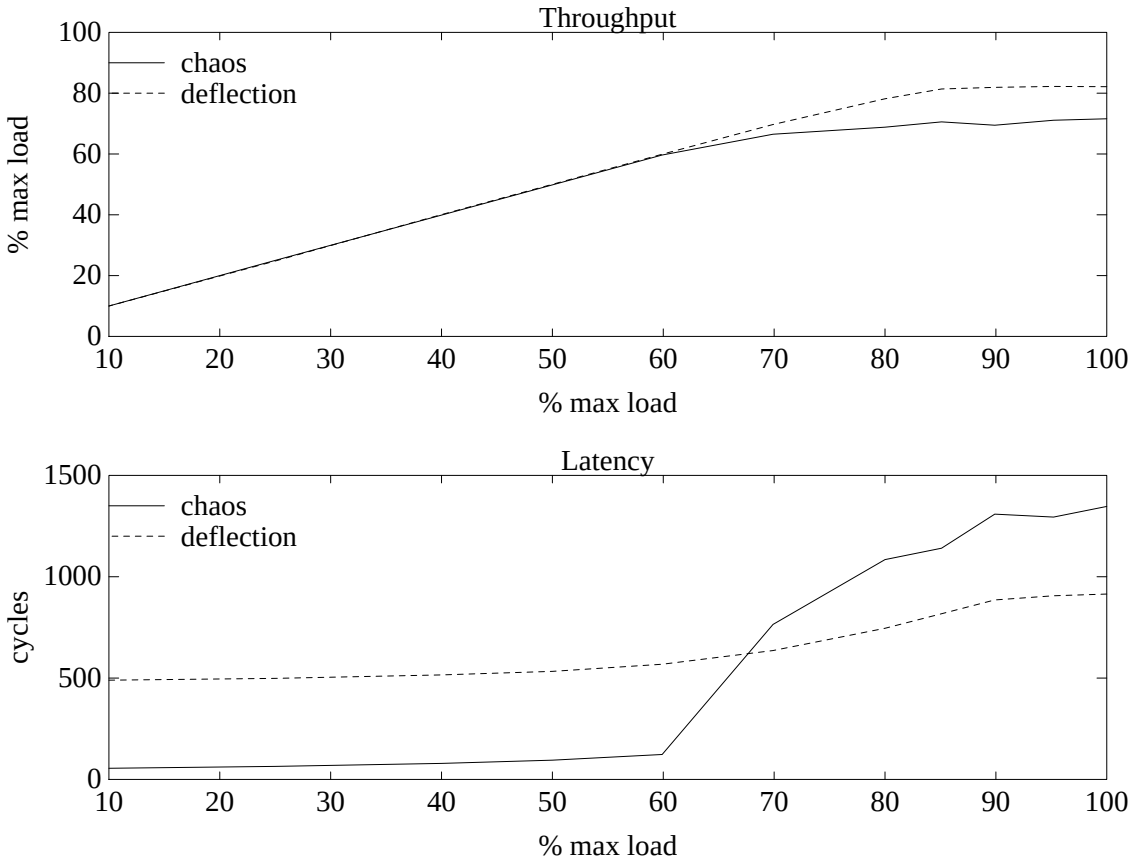
This chaotic router was then compared with oblivious and deflection routers using two traffic models: uniform random and hot spot traffic. The hypercube and torus chaotic routers perform significantly better than oblivious routers for both kinds of traffic. On the mesh, though, with random traffic, the network-induced central hot spot causes oblivious routing to do as well as chaotic routing. However, with the introduction of hot spots, chaotic routing performs better than oblivious routing.

A chaotic router was compared with a deflection router. The results show that chaotic routing provides much lower latency, and, with shared channels, higher throughput than the deflection router.

Overall, it was found that for 256-node torus networks, chaotic routing performs very much better than either oblivious or deflection routing.

---

[2] Using shared channels, chaotic routing achieves higher throughput (see Figure 4.4).



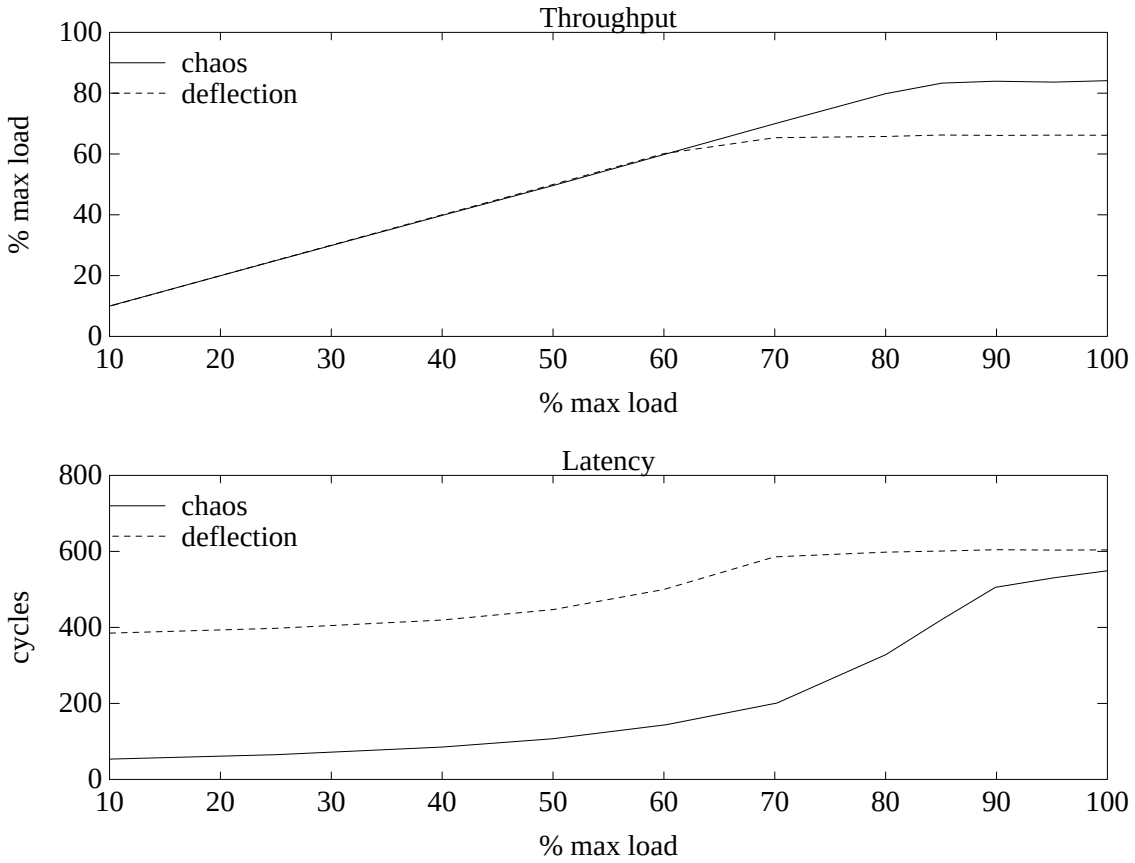

Chaotic Routing

| % load | 10 | 25 | 40 | 50 | 60 | 70 | 80 | 85 | 90 | 95 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| mean xpt | 10.00 | 25.00 | 39.90 | 49.84 | 59.66 | 66.50 | 68.82 | 70.56 | 69.46 | 71.10 | 71.58 |
| std xpt | 0.00 | 0.00 | 0.00 | 0.05 | 0.12 | 0.41 | 0.78 | 0.42 | 0.39 | 0.69 | 0.92 |
| mean lat | 54.99 | 64.25 | 79.04 | 95.00 | 123.08 | 764.8 | 1084.5 | 1140.8 | 1308.9 | 1294.4 | 1346.9 |
| std lat | 0.09 | 0.15 | 0.53 | 0.75 | 1.38 | 39.61 | 32.04 | 36.96 | 22.16 | 32.06 | 41.25 |

Deflection Routing

| % load | 10 | 25 | 40 | 50 | 60 | 70 | 80 | 85 | 90 | 95 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| mean xpt | 10.00 | 24.80 | 40.06 | 50.02 | 59.92 | 69.70 | 78.16 | 81.38 | 81.92 | 82.22 | 82.14 |
| std xpt | 0.00 | 0.00 | 0.05 | 0.07 | 0.04 | 0.14 | 0.39 | 0.77 | 0.70 | 0.35 | 0.30 |
| mean lat | 489.6 | 498.5 | 515.6 | 533.1 | 568.3 | 635.8 | 745.7 | 817.4 | 885.5 | 906.1 | 914.6 |
| std lat | 1.09 | 1.66 | 1.70 | 2.32 | 2.15 | 4.60 | 10.89 | 15.70 | 9.24 | 5.31 | 8.85 |

Figure 4.14: 256-node nonshared channel mesh: chaotic vs deflection (Uniform traffic)




Chaotic Routing

| % load | 10 | 25 | 40 | 50 | 60 | 70 | 80 | 85 | 90 | 95 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| mean xpt | 10.00 | 24.94 | 39.80 | 49.62 | 60.02 | 70.16 | 79.84 | 83.32 | 83.94 | 83.64 | 84.12 |
| std xpt | 0.00 | 0.05 | 0.00 | 0.04 | 0.32 | 0.14 | 0.37 | 0.28 | 0.59 | 0.21 | 0.62 |
| mean lat | 53.45 | 65.11 | 85.17 | 107.08 | 143.90 | 201.04 | 328.14 | 421.30 | 505.45 | 530.64 | 548.98 |
| std lat | 0.07 | 0.28 | 0.40 | 1.40 | 2.67 | 1.76 | 5.00 | 15.00 | 5.84 | 11.17 | 4.90 |

Deflection Routing

| % load | 10 | 25 | 40 | 50 | 60 | 70 | 80 | 85 | 90 | 95 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| mean xpt | 9.92 | 25.02 | 40.00 | 49.96 | 60.12 | 65.36 | 65.76 | 66.24 | 66.12 | 66.20 | 66.16 |
| std xpt | 0.04 | 0.04 | 0.00 | 0.05 | 0.12 | 0.49 | 0.59 | 0.62 | 0.10 | 0.48 | 0.58 |
| mean lat | 384.9 | 397.6 | 419.4 | 446.8 | 499.8 | 585.6 | 598.0 | 600.9 | 604.4 | 603.4 | 603.9 |
| std lat | 0.83 | 0.58 | 0.59 | 1.41 | 4.54 | 5.22 | 3.10 | 1.24 | 2.60 | 2.01 | 2.56 |

Figure 4.15: 256-node nonshared channel torus: chaotic vs deflection (Uniform traffic)

# Chapter 5

# IMPLEMENTATION – MESH/TORUS ROUTER CHIP

The design presented in Chapter 3 provides a framework to begin the implementation of a chaos router chip. Because the design includes a large amount of area-consuming buffering and must meet difficult timing constraints, it is necessary to actually build a prototype chip to confirm the feasibility and performance of chaotic routing.

Thus, a prototype chaos router chip has been designed and built to meet the following primary goals:

- Speed: The limit to the clock speed should be imposed by the pads. The pads should run as fast as possible given the technology.
- Area: The chip must be small enough to fit in a standard size package.
- Functionality: The chip must represent a *real* router with all its functionality. It should not be a scaled-down version.

By building a router which meets the above goals, it has been established in a direct manner that it is possible to build a competitive adaptive router using present technology. In order to provide a basis for comparison, the router has been designed to work in both the fully-adaptive *chaos* mode as well as a dimension-order *oblivious* mode. By implementing the oblivious subset, the implementation effort may be salvaged even if design errors in the adaptive logic manifest themselves at the chip testing time.

## *5.1 Methodology*

For this prototype router chip, a mesh/torus chaotic router with packets of 20 16-bit flits is chosen. The multiqueue holds five packets, as suggested by simulation studies (see Section 4.4). The chip is designed to conform to MOSIS's scalable CMOS rules,

with a 1.2$\mu$m process at a target. A limit of 132 pins for the chip package is set by the need to be able to mount the chip in a standard test fixture.

The design methodology encompasses several levels of customization. Parts where timing or area requirements are critical are custom designed. These parts include the crossbars, FIFO's, multiqueue controller, and clock driver circuits. On the other hand, the remaining logic, which includes the channel controllers, the increment/decrement circuits, and all of the control logic, is implemented using standard cell arrays. The standard cells are generated by the ChipCrafter design tools. All of the parts are then placed and routed using ChipCrafter's placement and routing tools.

A two-phase, level clocking strategy is used throughout the chip. In order to do this, the chip has as input a two-phase non-overlapping clock, as shown in Figure 5.1. The two phases of the clock are distributed throughout the chip and inverted locally

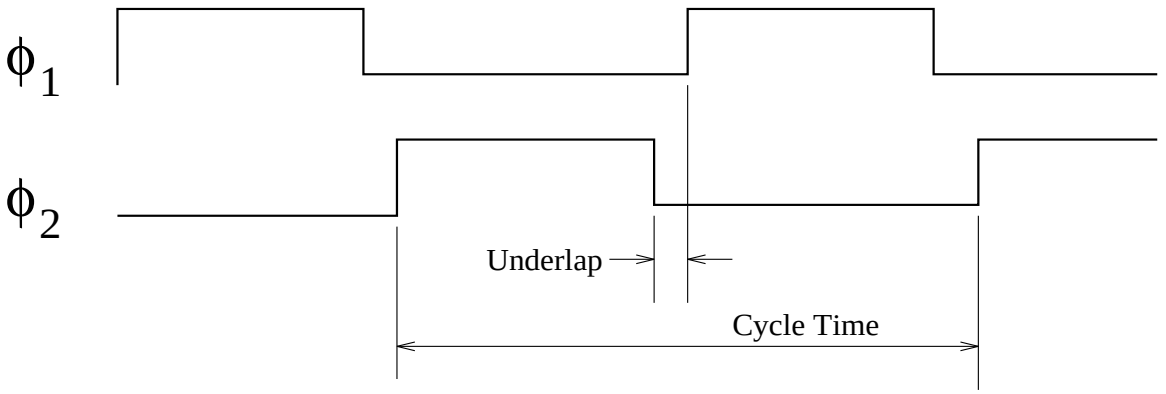



Figure 5.1: Two phase non-overlapping clock.

when necessary. Level-sensitive dynamic latches control the flow of data through the chip and can be used to easily construct multi-stage pipelines, as shown in Figure 5.2. By using level-sensitive clocking, pipelined stages may use the full clock cycle to perform their computation even though each pipeline stage is only "on" during one-half of the clock cycle. This is due to the level-sensitive nature of the latches: they are "transparent" when the clock signal is asserted and will pass signals directly from input to output during the entire clock phase. This contrasts with edge-triggered clocking, where the latched signal is not available until the end of the clock phase. This design style leads to high-performance results with reasonable design complexity.

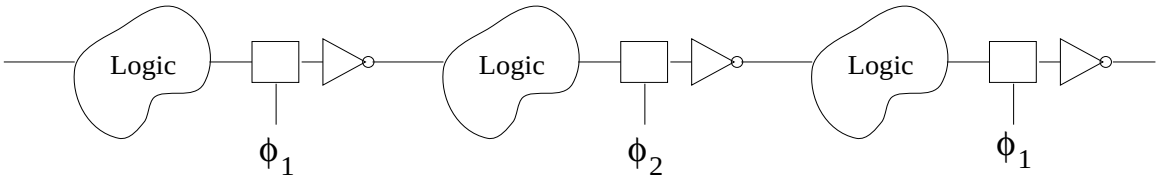



Figure 5.2: Pipeline constructed with two-phase level-sensitive dynamic latches.

### *5.2 Data paths*

The primary path that packets follow through the router is: input channel → input frame → main crossbar → incrementer / decrementer → output frame → output channel. If a message goes through the multiqueue, the path is: input channel → input frame → queue input crossbar → queue slot → main crossbar → incrementer / decrementer → output frame → output channel. Each of these components communicates using a standard bus protocol.

The on-chip busses follow a standard two-wire synchronous protocol. Each bus consists of 16 data bits, a parity bit[1], an end-of-message (EOM) bit, and two control lines: data valid (DV) and taking data (TD). On chip, all busses are uni-directional. The communication protocol uses the two control wires, DV and TD, to control the flow of data across the on-chip busses. When a sending component has valid data, it asserts DV at the beginning of the clock phase. When a receiving component has room to receive data, it asserts TD, also at the beginning of the clock phase. At the end of the phase, both the sending and receiving ends compare the DV and TD signals. If both are asserted, the transmission is considered successful and complete. If either or both is not asserted, then the transmission was not successful and must be retried.

Each of the main datapath components is discussed in detail below.

#### *5.2.1 Channel*

The datapath portion of the channel is straightforward in design. Two internal uni-directional busses connected to an input frame and an output frame are multiplexed on to a single external bi-directional bus. Ownership of the bus is determined by

[1] Although all of the logic and datapaths for parity are present on the prototype chip, it is not actually implemented due to pin limitations.

the channel controller (see Section 5.5). Data to be sent out the channel is latched and held for a full clock cycle to allow time for the pads to respond. Incoming data is latched immediately after arriving on the pads. With this arrangement, the only delay added to the pad delay is the delay to latch the data, so the pads can be cycled as fast as possible. For the bidirectional pads used, the sum of the worst case delays for driving the output (with a 50pF load) and receiving on the input is about 15ns [Cascade Design Automation Corporation 91]. Thus, the target clock cycle time for the router chip has been set at 15ns.

### *5.2.2 FIFO*

The FIFO is a critical part of the router, as it is the heart of the input and output frames as well as the queue slots. Each FIFO is capable of buffering 20 16-bit words. Since latency is a critical factor in building the chaos chip, the FIFO's are designed so that packets "cutting-through" experience minimal delay. In order to achieve this, each FIFO is implemented in two main components as shown in Figure 5.3: a RAM and a control section. The data is stored in a dual-ported RAM which allows concurrent reads and writes (to separate locations). The control logic consists of head and tail pointers and logic to determine the full/empty status of the FIFO.

Data arriving at a FIFO is stored in the RAM at the location specified by the tail pointer, assuming the FIFO is not full. When data is requested from the FIFO, it is read out of the RAM from the location pointed to by the head pointer. Whenever data is read/written, the corresponding pointer is shifted by one to indicate the change in the FIFO's status. In order to easily distinguish between a full and an empty FIFO, one unusable flit of storage is needed. The FIFO is empty if the head and tail pointers point to the same place, and it is full if the head pointer points to the location prior to the tail pointer. The control logic determines the empty status of the FIFO by ANDing the corresponding bits of the head/tail shift registers and then using a precharged OR to determine if the pointers point to the same place. To determine the full status of the FIFO, the same logic is used, except that the bits which are ANDed together are shifted by one so that the logic will indicate if the tail points to the location before the head.

The critical path of the FIFO is the logic which determines whether or not a read/write has occurred and directs the shifting of the head/tail shift registers. To

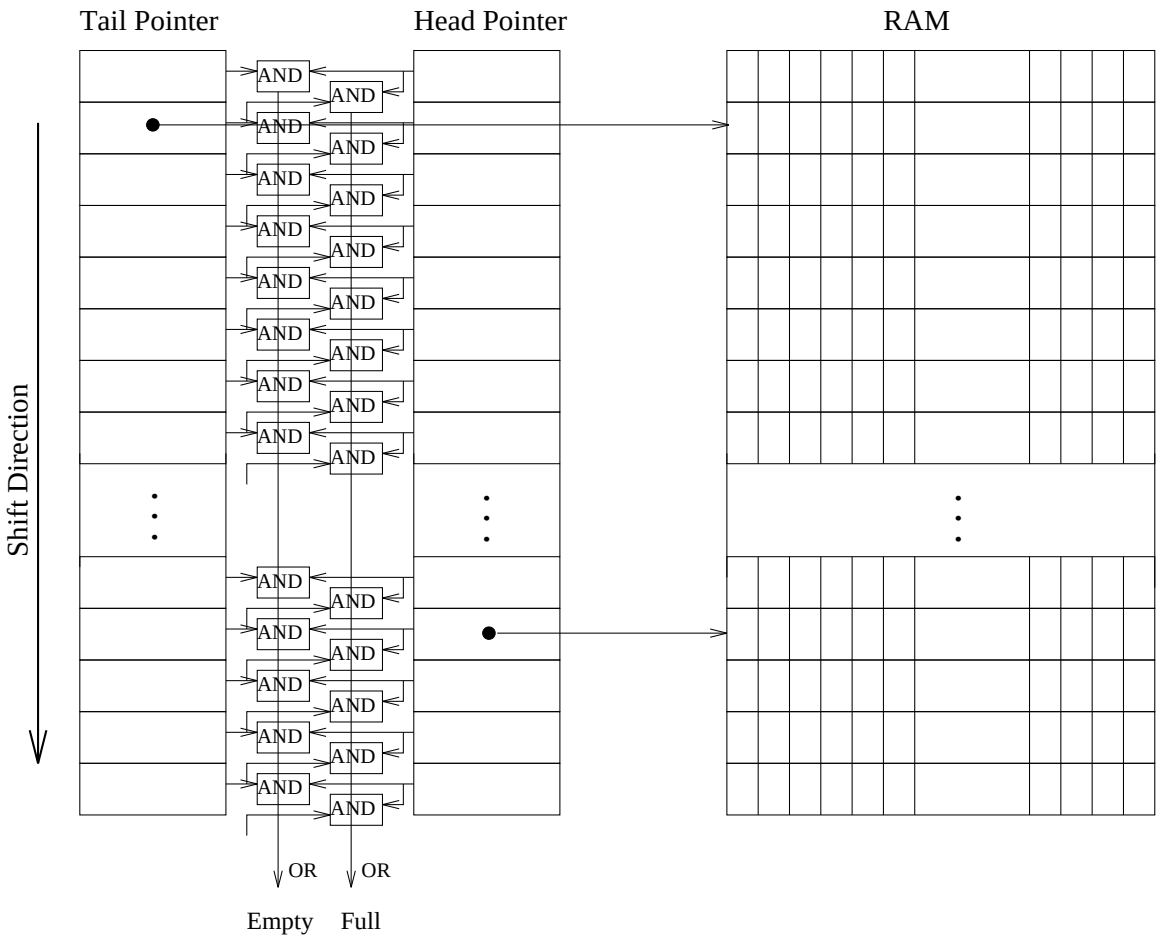



Figure 5.3: FIFO diagram.

determine if a word has been written into the FIFO, the logic must examine the DV signal coming in with the input data. If the incoming DV is asserted and the FIFO has asserted TD, then a write has successfully completed, and the tail pointer should be shifted. Unfortunately, the incoming DV signal may be connected through the crossbar, which makes it a rather late arriving signal. Thus, there is little time for the logic to determine whether to shift or not. This is the slowest, and thus, the most timing-critical path in the FIFO. For a *read* from the FIFO, the same path is critical, except that it depends on the incoming TD signal instead of the incoming DV signal.

The read/write critical path is necessary if the FIFO must separately control each flit which passes through it. However, since packets in a chaos network do not stall midway through transmission, it is guaranteed that once the first flit of a packet arrives in a FIFO, all remaining flits will immediately follow without delay. Thus, it is possible to control packets at the packet level instead of the flit level. This should simplify this critical path and reduce the complexity of the FIFO. However, it was decided not to change the design of the FIFO because the time to redesign and reconfirm correctness was too great. Since this critical path does not limit the clock speed (it is shadowed by the time taken for the pads to cycle), this does not limit the performance of the chip. In future designs, which may have faster pads, the FIFO's may be redesigned to remove this critical path.

The FIFO is implemented using full custom logic. In 1.2$\mu$m CMOS, it takes up an area of 684 by 820 $\mu m^2$. The layout is shown in Figure 5.4. For this figure, a small FIFO which holds only 10 8-bit words is shown. Also, only the first-level metal and poly layers are shown in order to increase clarity. The dual-ported RAM is on the right side, and consumes most of the area when the full 16-bit data path is implemented. To the left are the head and tail pointer shift registers and full/empty logic. Above and below the RAM are the input and output drivers, and the communication protocol logic is above and below the shift registers.

### *5.2.3 Crossbar*

The largest component on the chaos chip is the crossbar unit, which actually contains two crossbars: the queue input crossbar and the main crossbar. The queue input crossbar connects the five input frames to the five queue slots, and the main crossbar connects the five input frames and five queue slots to the five output frames. The

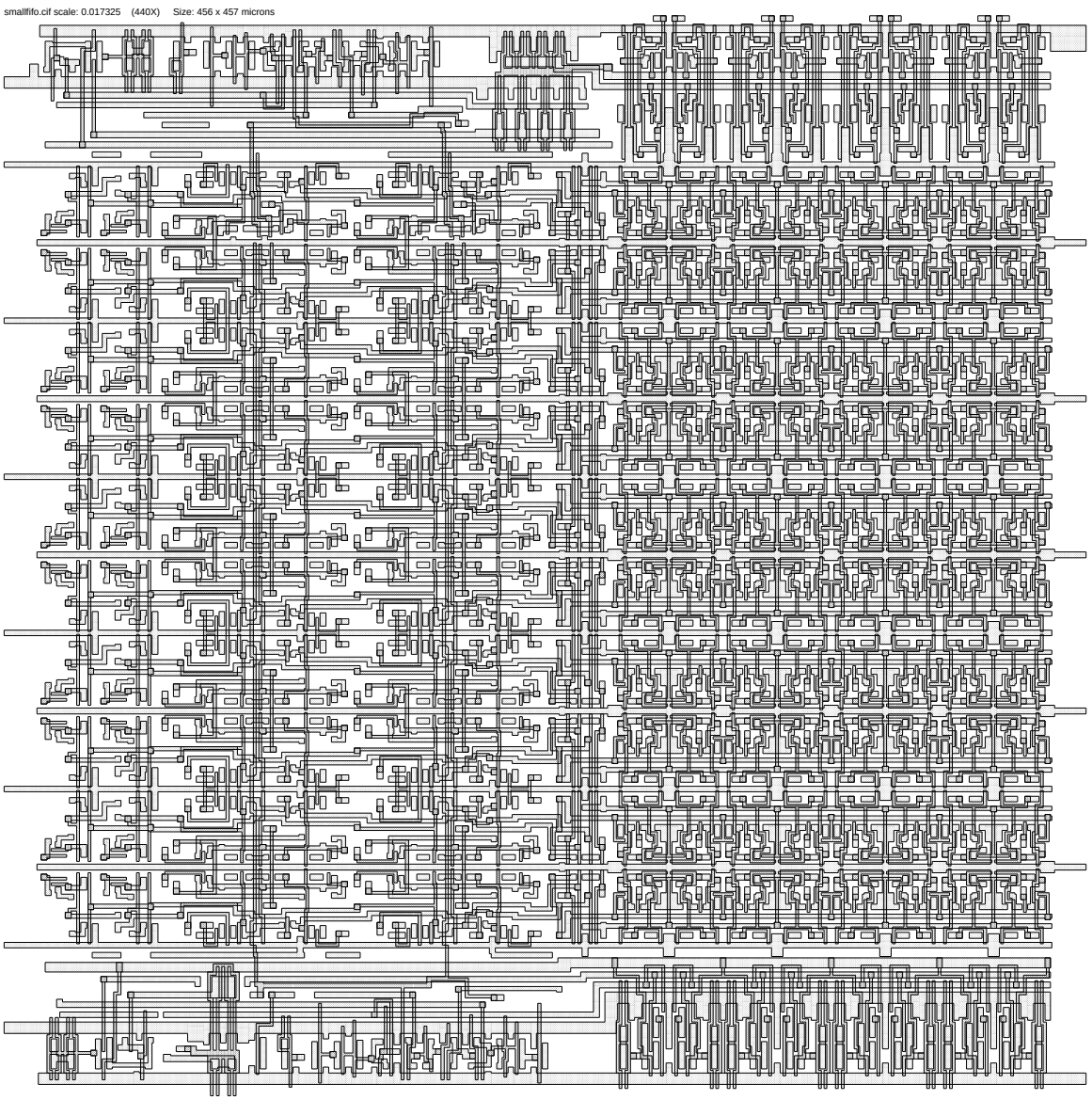



Figure 5.4: FIFO layout.

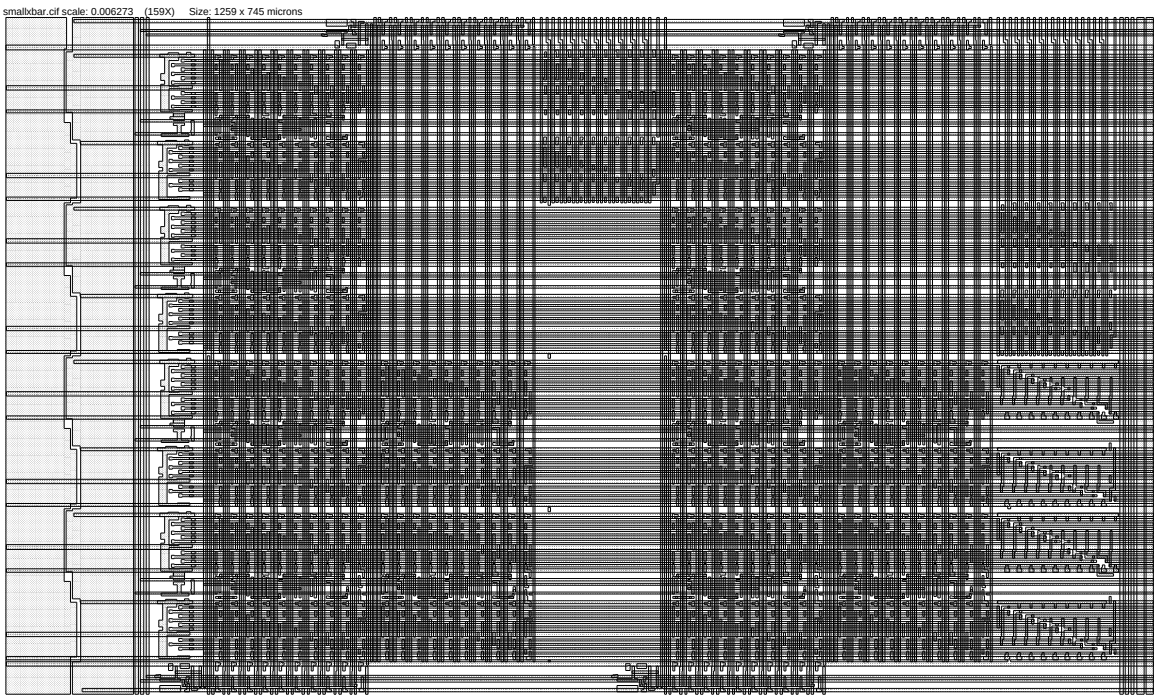



Figure 5.5: Crossbar unit layout.

two are integrated into one unit so that the input frame drivers can be shared.

Each of the crossbars registers its control settings so that they are stable until reset. The crossbar configuration is set during phase two of the clock by asserting the appropriate source and destination signals. Once a path in the crossbar is set, it persists until it is reset. Resetting is automatically done when the tail of a packet passes through the crossbar.

The data path is straightforward. Data comes in the input lines on phase one of the clock cycle. For each connection which is set (there should be no more than one source for each destination and vice versa), the data is used to pull down the precharged output bus. The output bus is then inverted and sent to its final destination.

Since the crossbar is the largest component of the chip, minimizing the area is important. The result is that almost all of the area is covered with wires for the input and output busses, and very little space is left over. The two-crossbar unit covers and area of 2758 by 1750 $\mu$m$^2$. Figure 5.5 shows a reduced crossbar unit which

would connect to two input frames, two output frames, and two queue slots. Only the two metallization layers are shown for clarity. The input busses run vertically and the output busses run horizontally. The two input frame inputs are spaced along the bottom, and the two queue slot inputs run along the top. The bottom two output busses go to the output frames and exit the crossbar along the right edge. The top two output busses go to the queue slots and exit the crossbar along the top edge, each adjacent to the input bus for the same queue slot. As can be seen in the diagram, the output frame busses have connections to both the input frames and the queue slots, corresponding to the main crossbar connections. On the other hand, the output busses which feed the queue slots only have connections to the input frames, corresponding to the queue input crossbar.

Inside the crossbar are the connect cells which hold the crossbar settings and pull down the output busses. A connect cell is shown in Figure 5.6. In this diagram, only the first and second-level metal and poly layers are shown. In the center is the register which holds the switch setting. It is controlled by the *source* line and the *destination* line. Above and below the register are the output lines which run horizontally across the chip. The data inputs to the switch cell run vertically in first-level metal and are connected to the data outputs by the pull-down circuitry. This circuitry consists of a NOR which logically ANDs the switch setting and the data value and pulls down the bus if both are set. The sizing and layout of this cell is adjusted to minimize the layout area.

### *5.2.4 Incrementer/Decrementer*

As a packet moves out an output channel, its header must be updated to reflect its progress. Packets exiting the positive X and Y channels must have the X or Y displacements of their headers decremented by one, and packets exiting on the negative X and Y channels must have their displacements incremented by one. Packets destined for the processor do not need their headers updated. The incrementers and decrementers are implemented in standard cells. When the header of a packet arrives at an output frame, the header bypasses the FIFO so that it can be updated during the half-cycle delay the FIFO would normally add to its transmission time. By doing this, the time to update the header is effectively hidden.

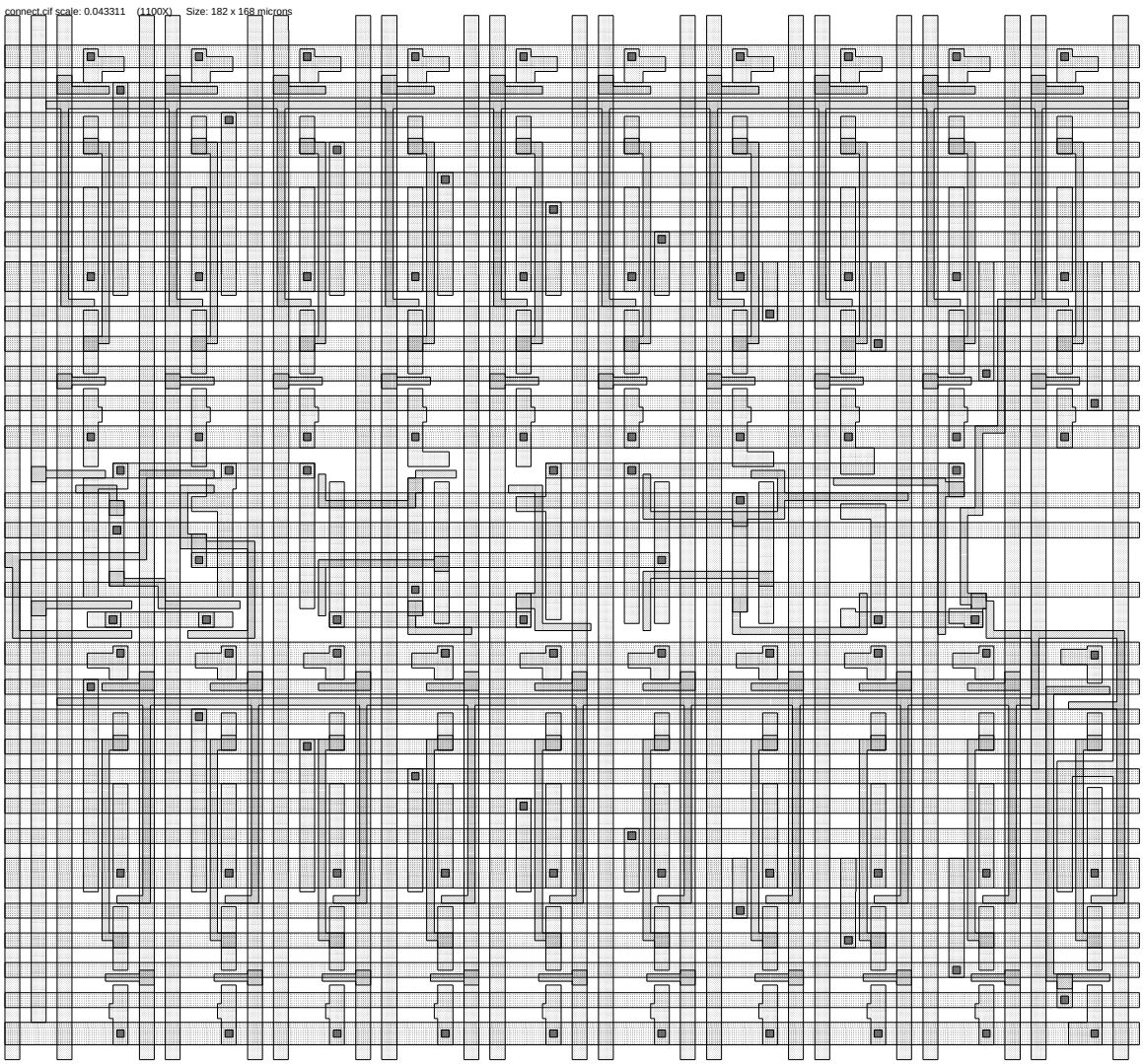



Figure 5.6: Crossbar Connect cell layout.

## *5.3 Control logic*

The control logic is responsible for routing messages between input frames, output frames, and the multiqueue. The high-level design of the control logic is presented in Chapter 3, and details are given in this section.

### *5.3.1 Routing pipeline*

The four stage routing pipeline decodes the headers of incoming messages, matches them with a destination, sets the crossbars to move the data, and then updates the header as the message leaves the router. Bubbles are introduced into the pipeline when there are no messages to be routed or when the router stalls waiting for input from the multiqueue.

The main routing pipeline is shown in Figure 5.7. In this diagram, the main functions are shown in blocks, and the level-sensitive latches are indicated as horizontal bars. The clock phase of each latch is indicated next to the latch. Packets which enter the router and encounter no contention will be routed along the main diagonal path of the pipeline. As can be seen in the diagram, the header must go through at least eight latches (four cycles) as follows:

1. Entering the chip through the input channel.

2. Still entering the chip through input channel.

3. Decode the header.

4. Select output frames which are available and desired by this message.

5. Pick one of the selected output frames to route to. Select input frame to be routed from.

6. Set up crossbar connection from input frame to output frame.

7. Transmit header across crossbar.

8. Increment or decrement the header.

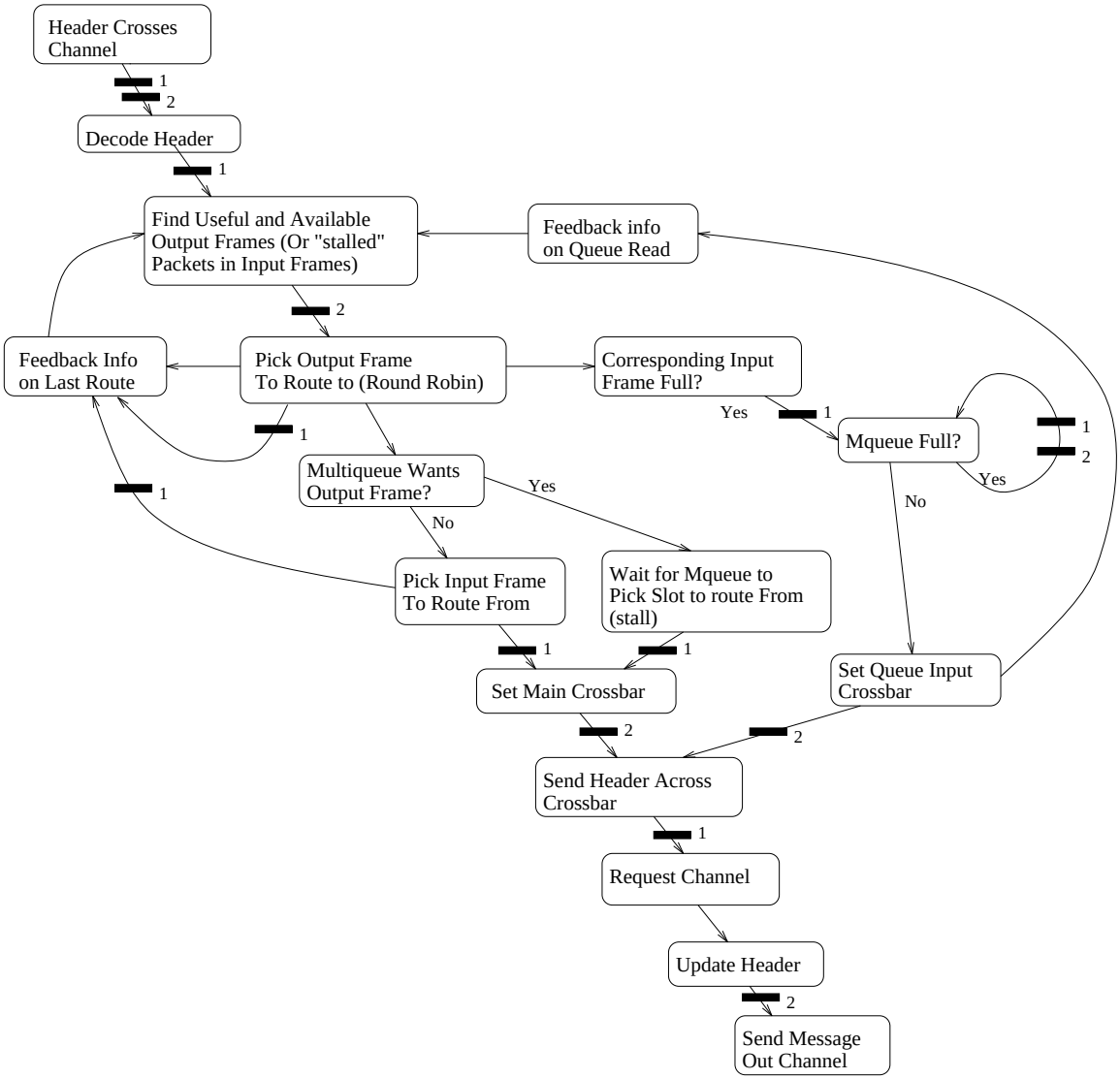



Figure 5.7: Control Logic routing Pipeline.

When a packet is routed to an output frame, a check is made to see if the corresponding input frame has a packet in it. If so, the packet in the input frame is read into the multiqueue if there is room in the multiqueue. This is done at the same time as the primary route, so there is no extra delay. If the multiqueue is full, however, the router stalls and waits until there is room in the multiqueue to read the new packet. This mechanism is also used to read messages which have "stalled" in input frames into the multiqueue. If a message has stalled, the corresponding output frame will be chosen as the current dimension, even if there are no messages which desire that output frame. Although no messages will be routed to the output frame, the logic will see that the input frame is full and read the message into the multiqueue. If the router is in oblivious mode, however, no packets are ever read into the multiqueue for any reason. This, in conjunction with the use of dimension-order routing, provides a simple way to implement a deadlock-free oblivious router.

When there are packets in the multiqueue that want a particular output frame, they have priority over packets in input frames. Once an output frame is selected as the destination for the current route, if a packet in the multiqueue desires it, the control logic instructs the multiqueue to send it the appropriate packet. This process takes several cycles and requires that the control logic stall while waiting.

Because a packet header may have started into the pipeline before the previous routing has completed, it is necessary to feed back information on the previous route into the pipeline. Otherwise, the new route is based on stale information and more than one packet may be routed to the same output frame at once. The feedback path through the queue read takes the longest, making it the critical cycle time for the control logic. Currently, it requires about 13 ns for the cycle.

### *5.3.2 Header decode*

As the header of a packet enters an input frame, the header is decoded to determine the profitable output frames for that packet. The 16 data bits in the header are divided into an 8 bit X displacement field and an 8 bit Y displacement field. The displacements are two's complement numbers representing the displacement of the destination node from the current node in either the X or Y direction. Thus, the addresses are relative instead of absolute and must be updated each time the packet moves in the network.

The router must decode the header into a list of profitable output frames. In adaptive mode, this is straightforward. For both the X and Y fields, if the displacement is non-zero, then the positive direction is profitable if the displacement is negative, and the negative direction is profitable if the displacement is positive. This is because the router is attempting to bring the displacement to zero. If both displacements are zero, then the packet is at its destination and, thus, the processor channel is the only profitable channel. When in oblivious mode, the only difference is that either Y direction is profitable only if the X displacement is zero. This implements the dimension order (X, then Y) mesh routing scheme.

The header is decoded as it arrives at an input frame and held in a register until the header flit is at the front of the input frame FIFO. At this point the decoded header information is made available to the router and the packet is available for routing. As soon as the main control logic assigns a route from an input frame to some destination, the decoded header is cleared and the packet is no longer available for routing. This is accomplished by the feedback paths in the pipeline.

### *5.3.3 Output Frame*

During each cycle, each output frame determines if it is *available* and *wanted* by some packet. An output frame is *available* if it is not full and there is no packet currently flowing into it. It is *wanted* if any input frame or the multiqueue has a packet which can profitably use it.[2] To compute this, each output frame has as inputs six *wants* signals. Five are for the five input frames, and the sixth indicates that some packet in the multiqueue desires this output frame. If an output frame is both available and wanted, it is considered *interesting*. An output frame is also considered interesting if there is a stalled packet in its corresponding input frame in order to cause the router to read stalled packets into the multiqueue regardless of other traffic.

Recall from Section 3.3 that routing is controlled by the *ActionDim* vector which indicates which output frame is currently being considered for routing. *ActionDim* visits each *interesting* output frame in a round robin fashion. If an output frame is *interesting*, it is guaranteed to be a profitable destination for a packet in either the multiqueue or an input frame, unless it is interesting only because the corresponding

[2] When the multiqueue is in derouting mode, all output frames except the ejection frame are *wanted* by the multiqueue.

input frame contains a stalled packet. Thus, when *ActionDim* points to a particular output frame, there is always some action to be taken. Unless the selected output frame is *interesting* only because of a stalled packet, the output frame must select a packet to be routed to it. Since there may be more than one packet in the router which desires the output frame, a combination of priorities and randomization is used to select a single one.

To do this, a combination of a random source and a set of prioritizers is used (see Figure 5.8). Each prioritizer has six ranked inputs and six outputs. The prioritizer simply selects the highest ranked (with '1' being the highest) input as its single, one-hot, output. The random source is used to select which prioritizer will be used to arbitrate between multiple sources. Depending on which prioritizer is selected, different sources have different priorities. The multiqueue always has the highest priority (1), and the processor always has the middle priority (4). The remaining four input frames have evenly distributed priorities of 2, 3, 5, and 6.

The assignment of the middle priority to the processor was originally intended to bias the routing decision slightly against the injection frame in order to prevent overloading the network during busy times. However, it may also exacerbate the problem of starvation. Since the processor may not inject packets into the multiqueue, and packets in the multiqueue already have priority over the processor, the injection frame is already the subject of negative bias. Adding this additional bias only makes the problem worse. Future generations of chaos routers will probably have a different method of dealing with this bias against the processor.

The random source is simply a linear-feedback shift register. For the output frames, a 5-bit shift register which repeats its sequence every 31 cycles is used. Two bits are selected and decoded to provide a 4-bit one-hot selection for the four prioritizers.

## *5.4 MultiQueue*

The multiqueue consists of five queue slots, which are simply FIFO's, and control logic which governs the flow of packets through the multiqueue. There are two main components in the multiqueue control logic: the multiqueue input logic and the multiqueue scoreboard.

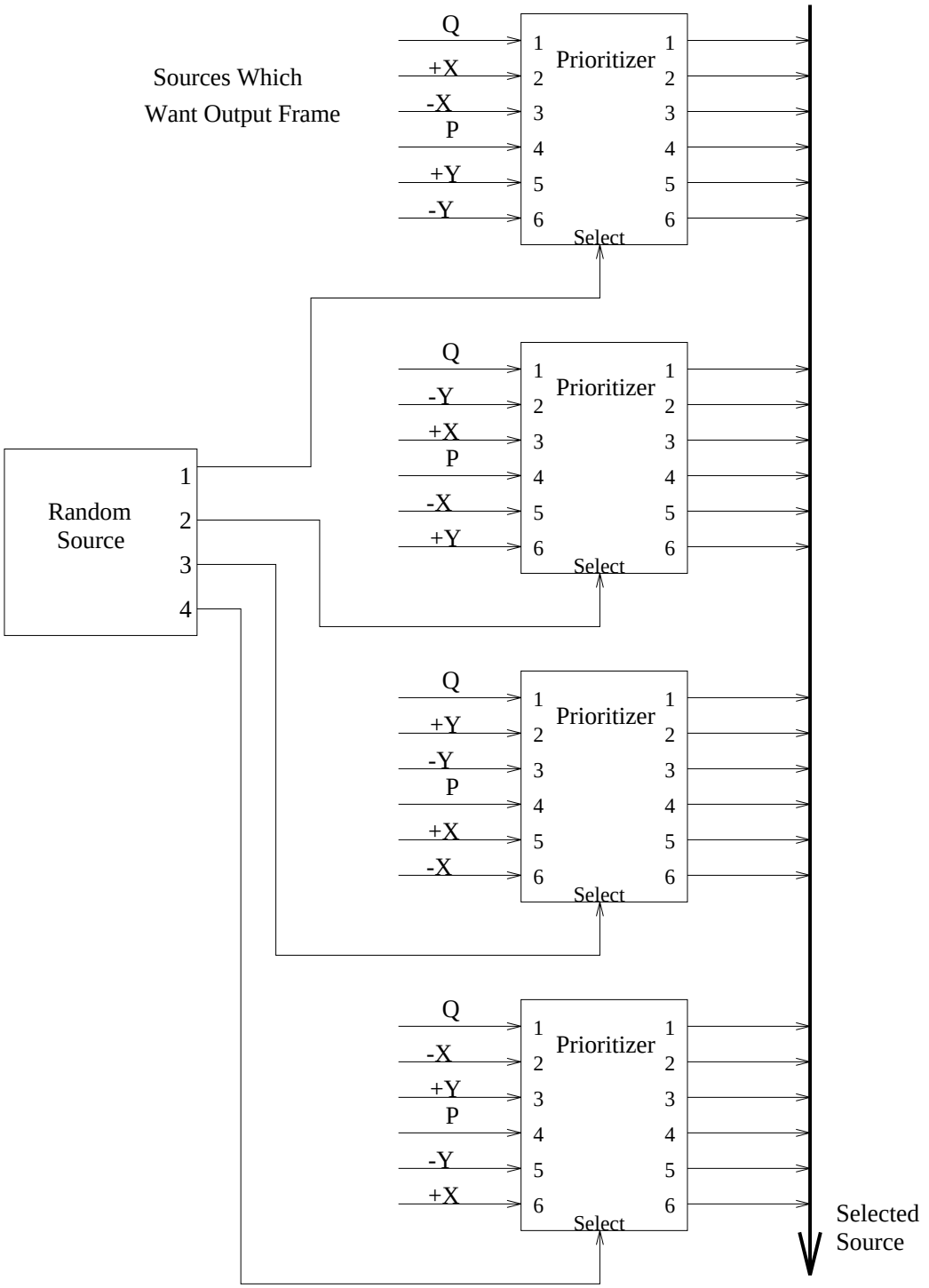



Figure 5.8: Output frame match logic.

### *5.4.1 Multiqueue Input Logic*

The queue input logic controls the flow of messages into the multiqueue. It has two primary functions: selecting free queue slots for packets entering the multiqueue and loading information on packets into the scoreboard at the appropriate times. Because of the wide variety of logic needed for these functions, the multiqueue input logic is implemented in standard cell logic.

When the multiqueue must read a packet from an input frame, the input logic must select a free queue slot for the destination of the queue read. This is done by keeping track of which have room and are not currently in the process of reading in a packet. When multiple queue slots are available, a simple prioritizer is used to choose between them. The logic does not distinguish between completely empty queue slots and slots which still contain the tail of a packet which is on its way out of the queue slot (states *Empty* and *Leaving* in Figure 3.4). An optimization which might benefit later versions of the router is to select first from completely empty slots, and then favor the slots which are the least full. This way, packets will reach the front of the queue slot FIFO's in the quickest time. The logic computes the next available queue slot each cycle, so the destination for a queue read is available immediately when needed. If no slots are available, the input logic signals the main chaos router control logic that it must stall on a queue read until a slot is available.

The second function of the input logic is to load information about packets into the scoreboard. Only when a packet in a queue slot becomes routable is information about it loaded into the scoreboard. The queue input logic decodes the header in the same way as the input frame logic (see Section 5.3.2) and loads this into the scoreboard along with the address of the queue slot. A prioritizer is used to make sure that information on only one new routable packet is loaded into the scoreboard at once. The input logic also asserts the *deroute* signal whenever all the queue slots are full. A slot is full if it has the header of a packet in it (states *Full*, *Filling* and *Sharing* in Figure 3.4), regardless of whether or not the packet is routable. However, all *full* queue slots will become routable eventually, since the header is guaranteed to advance to the head of the FIFO as the tail of the previous packet leaves.

### *5.4.2 Multiqueue Scoreboard*

The multiqueue scoreboard keeps track of the status of packets in the multiqueue, as described in Section 3.4.1. Because of the regularity of the two scoreboard matrices and the time-critical nature of the computations, full-custom design was chosen for the scoreboard. A diagram of the scoreboard components is shown in Figure 5.9. More detailed diagrams of the scoreboard matrices are shown in Figures 3.7 and 3.8.

The destination and storage matrices each have five columns, one for each queue slot, which hold the decoded header (in the destination matrix) and the one-hot encoded address of the queue slot the packet is in (in the storage matrix). Each storage cell contains a register and logic to compute the output functions for the particular matrix.

As new packets enter the multiqueue, information on them is loaded into the leftmost record of the matrices. As packets leave the multiqueue, their records must be removed from the matrices. Since they may not necessarily be at the right edge of the matrices when they are removed, records must be able to be cleared from arbitrary locations in the scoreboard. When a record is removed from the scoreboard, it may leave a *bubble* where it used to be. The destination matrix determines when and where bubbles occur and outputs the signal *bubble* with this information to the shift logic. The shift logic then compacts records to the right whenever new messages are loaded in and there are one or more bubbles in the scoreboard.

The destination matrix computes the *QueueWants* and *HeaderMatch* signals. *QueueWants* indicates which output frames are desired by any packet in the multiqueue. The destination matrix simply does a row-wise OR on all of the headers in the matrix, using pseudo-NMOS logic, to compute this signal. If the multiqueue is in derouting mode, though, this computation is pre-empted and all of the output frames except the ejection frame are considered desired by the multiqueue. This ensures that the main control logic will stop at the next available output frame and allow the multiqueue to deroute a message to it. *HeaderMatch* is a vector which indicates which records in the scoreboard correspond to packets which desire the output frame indicated by *ActionDim*. To compute this signal, *ActionDim* is bit-wise ANDed with each row of the destination matrix. The results are then ORed together column-wise to determine which rows match *ActionDim*. Once again, pseudo-NMOS logic is used for this computation.

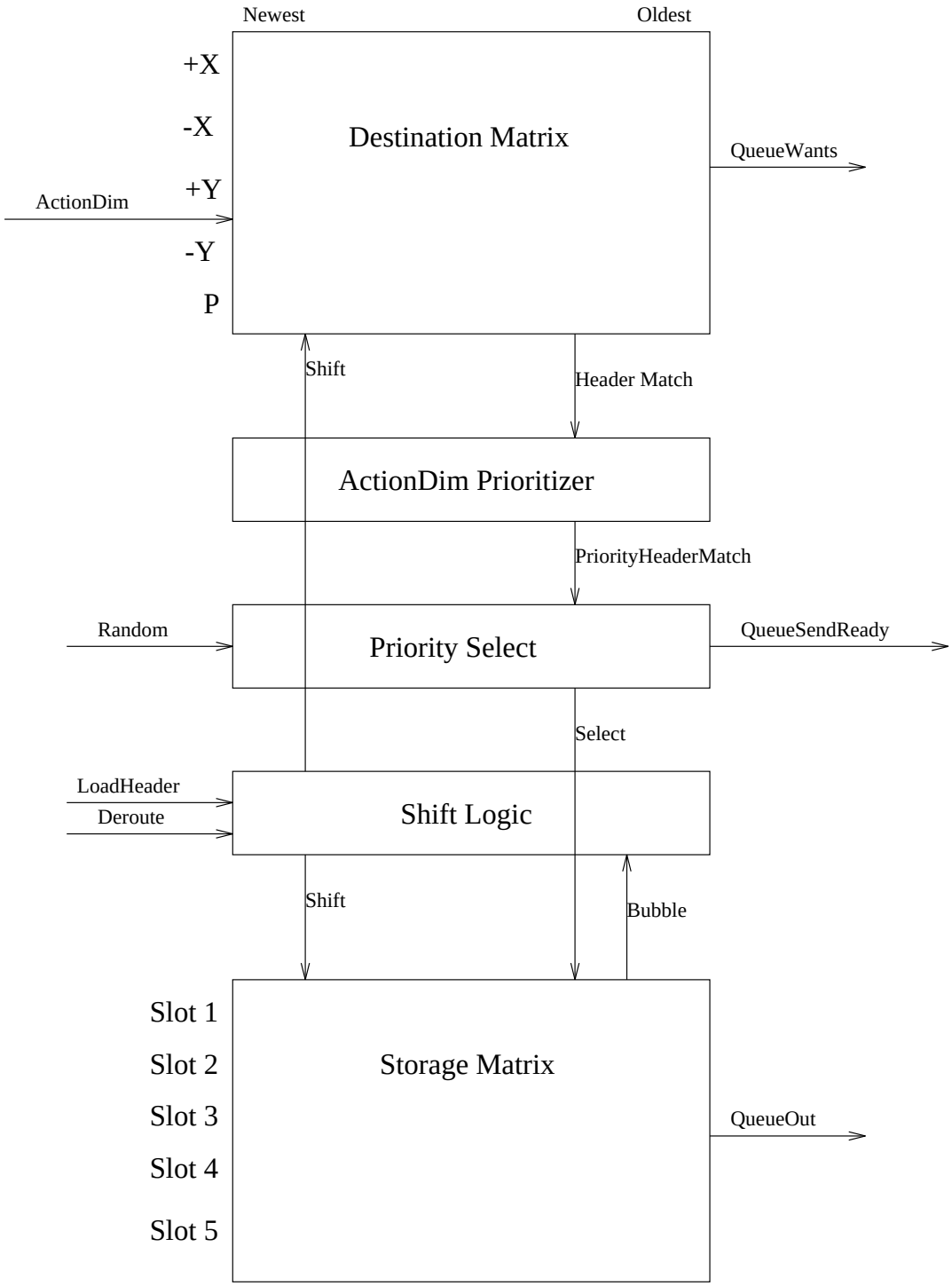



Figure 5.9: Multiqueue Scoreboard diagram.

Once the *HeaderMatch* vector is determined, the matched packet which has been in the multiqueue the longest must be selected. The ActionDim Prioritizer performs this function. Since the matrices order their records in a right-to-left age order, the rightmost matched header corresponds to the oldest packet. Thus, assigning right-to-left priority and selecting the highest priority accomplishes this function. The one-hot *PriorityHeaderMatch* vector is the result of this computation.

In the non-derouting case, *PriorityHeaderMatch* may be input to the storage matrix to compute the *QueueOut* vector immediately. However, if the multiqueue is derouting, then a *random* packet must be selected instead. The priority select logic is used to perform this function.

The selection of a random packet during derouting is done by using a linear feedback shift register as a pseudo-random number source. The shift register has a length of 7 bits and a maximal (127 cycle) period before repeating the pseudo-random sequence. Four bits are decoded into a five bit one-hot vector by decoding circuitry. Since the sixteen patterns can not be equally divided among the five one-hot bits, there is a slight positive bias to one of the five bits. Details on the operation of this logic can be found in Section 3.6.1.

Once the priority select logic has picked a queue slot to route from (either the matched header or a randomly picked slot), it notifies the main control logic that the queue send is ready by asserting the *QueueSendReady* signal. At the same time, the storage matrix transforms the selected record into the actual selected queue slot. This is done by ANDing the selected position with each bit to select the proper slot. The result of this AND is then row-wise ORed to produce the *QueueOut* vector which has the address of the queue slot selected as the source for the queue send operation.

The process of selecting a packet for a queue send operation takes three cycles if the multiqueue is not derouting. If the multiqueue is in derouting mode, it takes five or more cycles, as described in Section 3.6.1.

## *5.5 Channel Controller*

Between the off-chip interface and the input and output frames are the channel controller logic blocks. The channel controller interfaces the two on-chip uni-directional busses coming from and going to an output and input frame to a single bi-directional off-chip bus. The on-chip interface is the standard TD/DV half-cycle interface de-

scribed in Section 5.2, with the exception that once a packet has begun to move, the TD and DV signals are ignored until EOM is present. This results in the requirement that once the header of a packet is routed, the entire packet must follow without interruption. This is guaranteed by the chaos router chip assuming that the processor interface follows the same guidelines. The off-chip interface is a full-cycle interface which follows the more complex protocol described below.

The off-chip interface uses the same bus format for data transfer, with a 16-bit data path, and DV, TD, and EOM control lines. Since the channel is bi-directional, the direction the wires are driven depends on the owner of the channel. At all times one side or the other *owns* the channel and drives all of the data lines and the DV and EOM lines. The *non*-owner of the channel drives the TD line.

The three control lines, DV, TD, and EOM are treated very differently for the off-chip channels than for the on-chip data busses. Since it is assumed that once the transmission of a packet across the channel has begun, it will continue uninterrupted until the transmission is complete, it is not necessary to check the transmission of individual flits in packets. Thus, DV and TD can be ignored during the transmission of a packet. The end of a packet is signaled by the assertion of EOM. After this time, the channel controllers communicate to decide whether the owning side of the channel should retain ownership or yield ownership to the other side.

There are three basic rules which govern channel ownership arbitration:

1. A packet may be transmitted only if the receiving side has room available in its input frame.

2. If only the non-owning side can transmit a packet, ownership is yielded to that side.

3. One side may transmit two consecutive packets only if the other side has no transmittable packets.

The effect of these rules is to cause the two sides of the channel to trade ownership back and forth on a packet basis when both sides can transmit, and to allow either side to use the channel on a first-come-first-served basis otherwise.

In order to implement these rules effectively, information on the states of each side must be communicated. The channel owner uses the DV and EOM signals to

Table 5.1: Channel Arbitration Protocol Signals.

| Signal | Meaning |
|---|---|
| DV $\wedge$ EOM | Owner yields channel |
| DV $\wedge\neg$EOM | Owner is sending a new packet |
| $\neg$DV | Owner retains ownership, but has no packet to send |
| TD, TD | Non-owner's input frame is full, cannot receive |
| TD, $\neg$TD | Non-owner has packet to send, and its input frame is not full |
| $\neg$TD, $\neg$TD | Non-owner has no packet to send, and its input frame is not full |

communicate as shown in Table 5.1. The top portion of the table describes the use of the owner-driven wires. If DV is not asserted, the cycle can be thought of as a no-op, as nothing happens. If DV is asserted, the action depends on the value of the EOM signal. If EOM is high, then the owner is yielding channel ownership and will reverse roles on the next cycle. If EOM is low, the owner is sending a new packet out at this time, and the receiver should receive it. The owner takes into account information provided by the receiver when deciding what course of action to take.

To provide information on the receiver, one reverse-direction control wire, TD, is provided. Since more than one bit of information is required from the receiver, the protocol requires two cycles to transmit complete information. In Table 5.1, the first and second cycles of this transmission are separated by commas. The TD line can be viewed as a *request* line. When TD is asserted, at *any* time, the receiver is requesting ownership of the channel, either because it has a packet to transmit or because its input frame is full and the other side cannot transmit a packet anyway. If the receiver is requesting ownership, but its input frame is not full, it is possible for the sender to retain ownership and send a new packet. To indicate this situation, the TD signal is lowered on the second cycle. If the receiver's input frame is full, it is not possible for the sender to send a packet at all; this is indicated by holding the TD signal high for two consecutive cycles. If TD is not asserted at all, the receiver is signaling that it has no desire to own the channel and the sender may retain ownership.

The effect of this protocol is that if there are no packets being transmitted across the channel and a new packet arrives, it is transmitted without delay. When a packet arrives when the opposing side is transmitting another packet, it must wait

until that transmission is complete before the ownership can be transferred and the packet transmitted.[3] In this case, there is a single "dead" cycle when neither side can transmit data; it is used to exchange information to aid in the arbitration decision. Because of this, if both sides always have packets to transmit, channel utilization will only be 95%. If one side has several consecutive packets to transmit, and the opposite side has none, the packets will be transmitted without any "dead" cycles and channel utilization will be 100%.

## *5.6 Results*

Compiling and laying out all of the parts of the chaos router chip results in a design which fits on a die about 1.1 cm on a side and uses 132 pins. A floorplan is shown in Figure 5.10. The standard cell arrays take up approximately twice the area of the custom components, and, along with the routing, are responsible for most of the area on the chip. The entire chip is shown in Figure 5.11. In this figure, only the first and second-level metal layers are shown. The density of the full custom parts: the FIFO's, the crossbar unit, and the multiqueue scoreboard can be seen to be much greater than the density of the standard cell arrays.

The timing goals for this chip were to run the clock at the fastest speed allowed by the pad driver technology. For the 5V pads used, the worst-case timing will not allow signals to change faster than once every 15ns. Thus, a 15ns clock cycle time was chosen a a goal. Simulations using IRSIM (see Section 5.6.1, below), indicate that the chip should be able to run at this speed.

### *5.6.1 Simulations*

In designing the chaos router chip, many simulations were necessary to debug and verify correct operation. Individual components were simulated, primarily using the simulator IRSIM, with critical components analyzed by HSpice. Chip-level simulation was done using IRSIM. In order to provide a relatively easy "random" test of the chip, a simulation framework was developed. This framework consists of a set of external

[3] Because the implementation of the protocol is pipelined, a packet must arrive at an output frame a few cycles before the transmission is completed in the other direction in order to proceed with minimal delay. Otherwise, the presence of the new packet will not be known before the decision on channel ownership is made and the present owner may retain ownership.

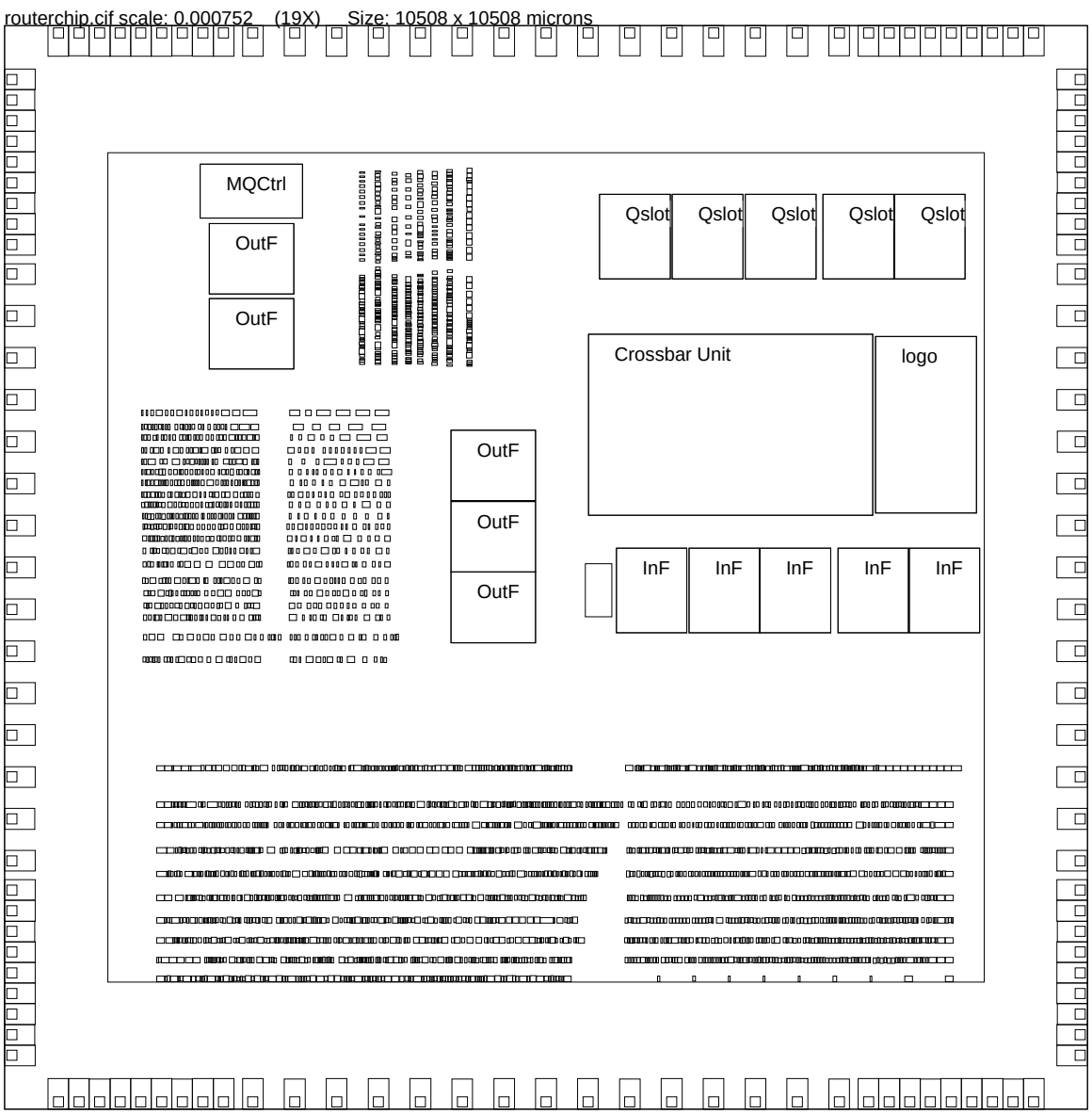



Figure 5.10: Chaos Router Floorplan.

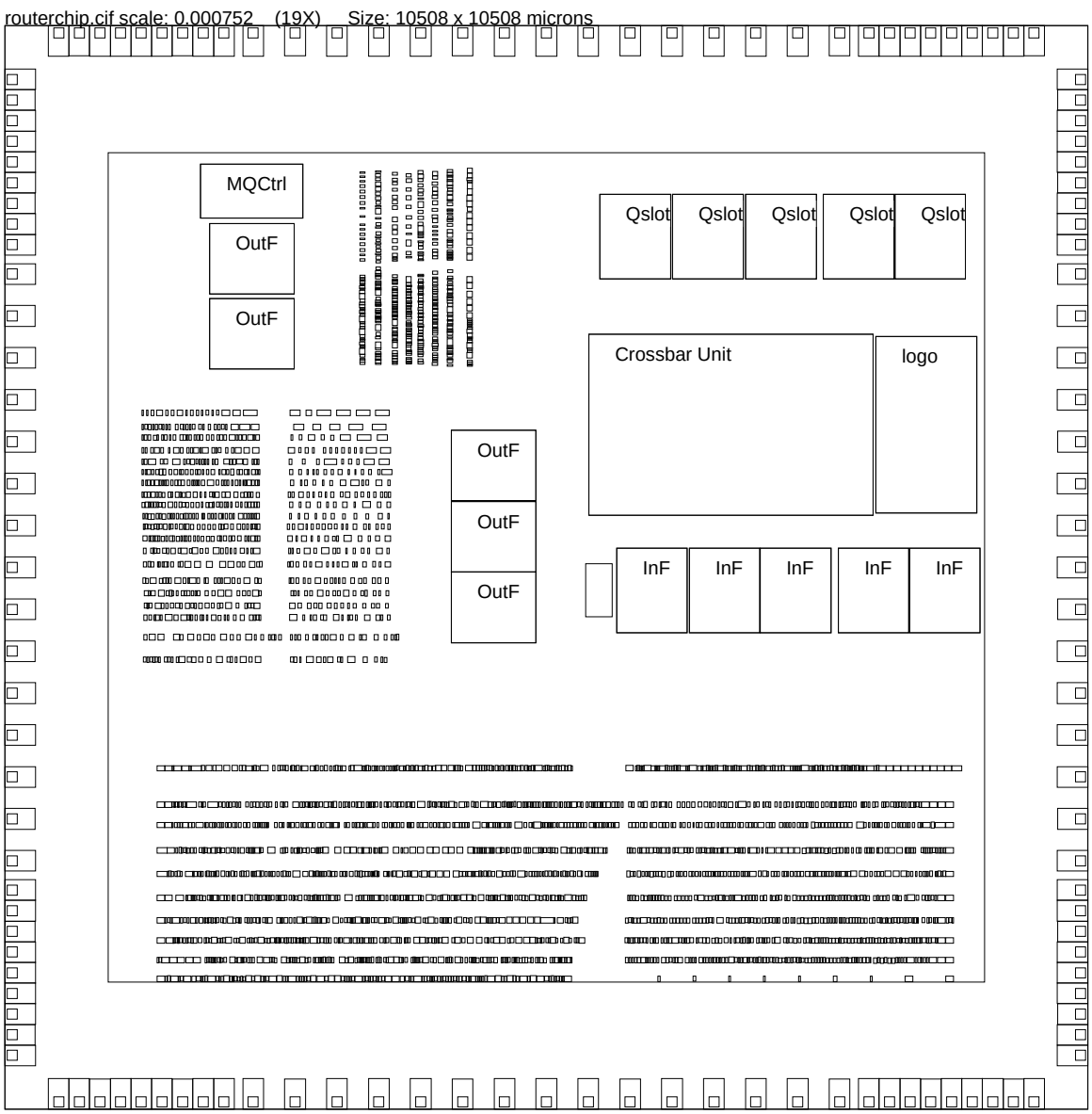



Figure 5.11: Chaos Router Chip Layout.

channel drivers and several small finite state machines to emulate the environment a chaos router might be placed in (see Figure 5.12). For each of the five external channels of the router, the channel controller circuitry that would exist if the chip were connected across a channel to another chaos router was added. Instead of being connected to an input and output frame, though, the channel is connected to a message generator finite state machine and some logic to receive messages.

The message generator has a single input, *send*, which determines when a message should be generated and sent to the router. The generator creates packets which carry information that help in determining correct operation of the router. The packet format is shown in Figure 5.13. The header is chosen pseudo-randomly, with an extra bias so that the X and Y fields each have a 25% chance of being zero. The header is duplicated in the second flit of the packet, so that it will be intact and can be compared at the output with the original header to verify correct routing. The next two flits identify the source of the packet and a sequence number to allow easy mapping of the packets. The remaining flits of the packet simply contain the number representing their order in the packet, except for the last flit, which contains a test pattern. The receive logic simply accepts flits when *receive* is asserted, dumping them as it receives them, and rejects flits when *receive* is not asserted. To emulate the loading of cables for the channels, a 50pF load is added to each of the wires in the channel busses.

To simulate the router, all that is necessary to do is to assert the *send* and *receive* signals appropriately to mimic the desired loading pattern. Correct operation can be verified by examining the signals going across the channels. To help with the verification task, a checker program was designed which verifies that all packets are routed intact and to their proper destinations. It also gathers statistics on the time packets take to cross the router and on the utilization of channels. By using the test framework and the checker program, simulations which send and receive hundreds of packets were successfully run.

## *5.7 Summary*

A prototype chaos router chip was designed and built. The goals for this chip were to be operable at a clock speed limited only by the speed of the off-chip pad drivers and to be implementable within a reasonable die size.

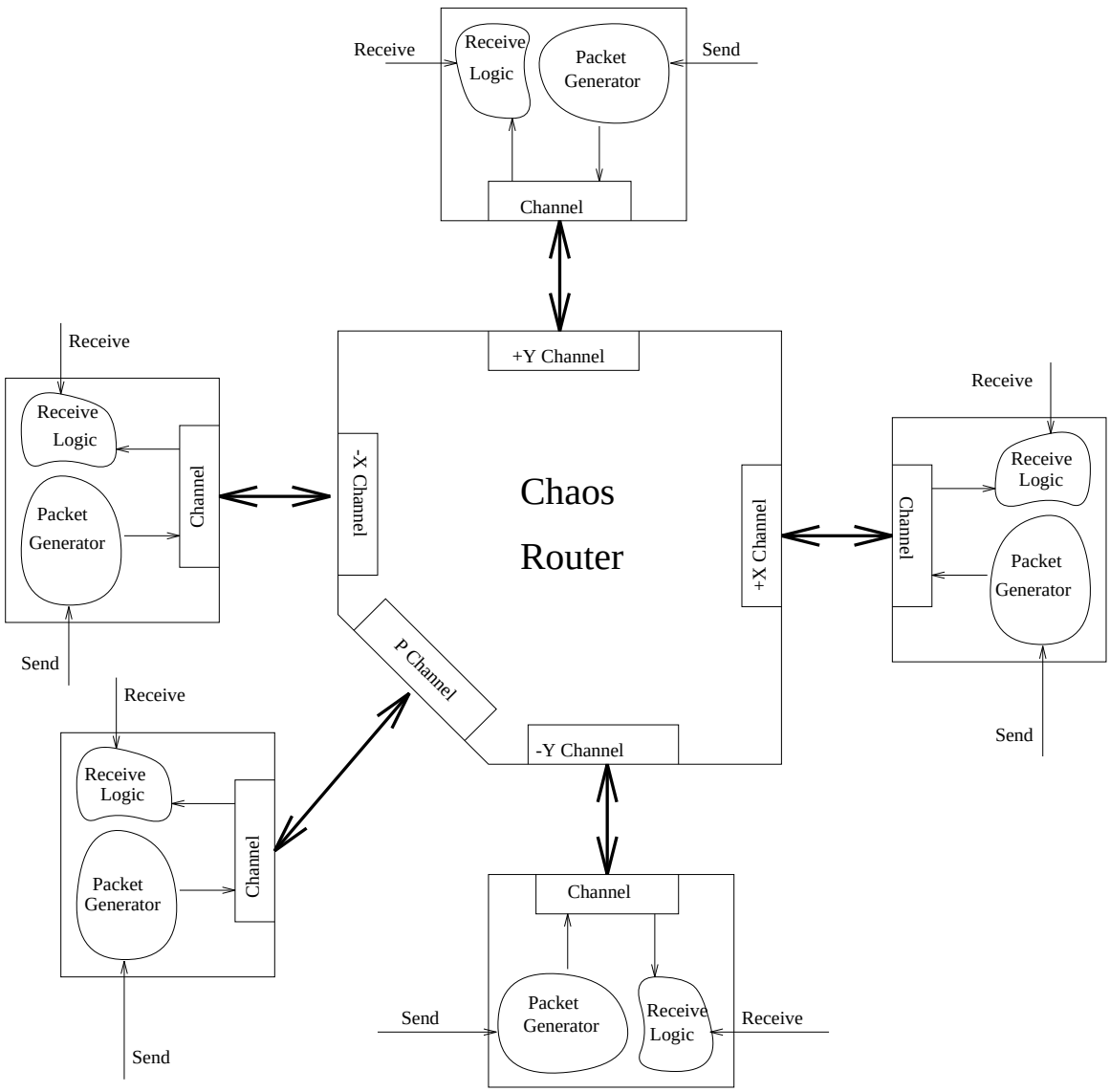



Figure 5.12: Simulator Environment.

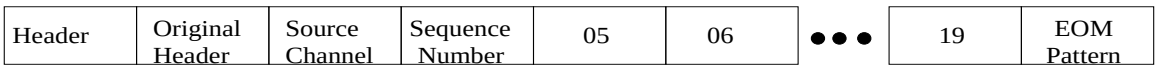



Figure 5.13: Simulator Packet Format.

The design of the chip uses a mixed custom and standard cell design style. The area/time critical components were built using full custom design, while the less critical parts were built using standard cell arrays.

The chip has been simulated and fabricated. Simulations indicate that the chip should be operable at a clock speed of 66MHz, which is as fast as the pads can be safely cycled.

# Chapter 6

# CONCLUSIONS AND REMAINING WORK

This dissertation describes the evolution of chaotic routing from a conceptual routing scheme for a specific network into a a class of routers for a broad class of networks, with a hardware implementation for a specific instance. By working to both generalize the chaotic concept and to improve the performance of chaotic routers, the work presented in this dissertation moves chaotic routing closer to maturity. However, much work remains to be done before the chaotic routing project is complete. Work in progress, such as the testing of the prototype chip and extended simulation studies, must be completed, and areas such as fault tolerance and recovery must be explored.

## *6.1 Conclusions*

The original work of Konstantinidou and Snyder, which proposed a hypercube-based store-and-forward chaotic router, has been expanded in the work reported on in this dissertation. Their proposed hypercube store-and-forward chaos router has been generalized to a scheme which can operate on a large class of network topologies. This has been accomplished by expanding the theoretical underpinnings of chaotic routing to include general networks of any finite size with bounded degree nodes and bi-connected links. This includes all of the members of the popular $k$-ary $d$-cube family.

Since the original chaos router design used a store-and-forward technique, it suffered a large latency penalty when compared to wormhole routing techniques. The router's design has been changed to allow a virtual cut-through design which combines the low-latency benefits of wormhole routing with the high throughput of store-and-forward routing. Not only does this design allow low-latency operation for packets which do not encounter congestion, but it also provides a mechanism to move the complicated multiqueue operations off of the critical timing path of the router. Since packets only enter the multiqueue when congestion is relatively high, a few extra cycles delay due to slower, but more easily implemented, multiqueue operations will

not significantly add to a packet's latency. In order to achieve a workable cut-through design, it is necessary to modify the router's operating protocol in order to ensure deadlock freedom. The resulting cut-through chaos router moves packets into the multiqueue during certain potentially deadlock-producing situations to ensure that deadlock will not occur. Thus, a clean, simple, and fast virtual cut-through router is the result of this design change.

The cut-through chaotic router is then analytically compared with several common routing schemes, from simple oblivious schemes to other complicated non-minimal adaptive schemes. It is shown that chaotic routing should provide better performance at lower cost than minimal adaptive routing or other non-minimal adaptive routing schemes. However, the oblivious router remains competitive due to its simplicity, which gives it very good low-load latency and reasonable high-load throughput.

A detailed design for a practical chaos router is then presented. This design provides several tradeoffs between efficiency and performance. Because any significant complexity of design causes the router to become uncompetitive with state-of-the-art oblivious routers, certain opportunities to maximize performance are sacrificed in order to simplify the design. The first simplification is to limit the router to setting up one new path through the router per cycle by means of the *ActionDim*. By serializing the routing decision this way, the control logic of the router is greatly simplified, although the latency of packets is slightly increased. However, the throughput is shown not to suffer because the multi-flit nature of the packets reduces the demand on the routing logic. The other major simplification is allowing the multiqueue operations to be "slow" compared to the cut-through operations. Because the multiqueue is not on any latency-critical paths, its performance can be sacrificed in order to allow a more easily implemented design.

The detailed design of the chaos router is compared with the design of current oblivious routers. It is shown that the chaos router can achieve the same raw bandwidth as oblivious routers because they are both bound by the same pad-speed limitations. However, the latency of a packet passing through a chaos router will be slightly greater – four cycles instead of three with chaotic routing instead of oblivious routing – when the network is lightly loaded. The effect of this difference depends on the size and topology of the network, as well as the packet length. In general, for smaller networks and larger packet lengths, the difference in latency is less pronounced, while

it is more apparent if packets are short or the network has a large diameter.

To further the chaos design, simulations were done which compared competing design alternatives. A series of simulations on mesh and torus networks showed that a multiqueue which holds five packets is optimally sized for overall throughput and latency needs. If the queue is made smaller, throughput and latency suffer significantly, while if it is made larger, there is very little improvement available. Also, it was shown that bi-directionally shared network channels outperform non-shared uni-directional channels by a large margin. The low-load advantage of shared channels is expected due to the ability of these channels to provide more resources during the low-load case. However, shared channels also provide much better throughput during high loading due primarily to the decreased performance of the multiqueue when longer packets with smaller flits, which are necessary with non-shared channels, are used.

To verify the relative performance of chaotic routing, many simulation studies were conducted to compare the performance of chaotic routing with oblivious and other schemes. The simulations, performed using a variety of traffic patterns, concentrated on mesh, torus, and hypercube networks with either 64, 256, or 1024 nodes. For the mesh network, chaotic routing did not perform significantly better than dimension-order oblivious routing. This is due to two factors: first, the mesh network creates a perpetual "hot spot" in the center of the network that proves to be a formidable barrier to packets which attempt to cross the central portion of the network. Because this hot spot causes many packets to be derouted, the performance of chaotic routing is degraded. Second, the dimension-order oblivious scheme tends to produce traffic which helps to minimize the hot spot in the middle and, thus, performance is improved for oblivious routing. Thus, on mesh networks, oblivious schemes perform as well as chaotic routing because of the detremental effects of the the central network-induced hot spot on chaotic routing.

When the mesh network is changed to a torus, however, the central hot spot disappears and the situation is considerably different. Here, chaotic routing outperforms oblivious routing significantly. Once again, there are two primary reasons for this. First of all, since the torus network is node-symmetric, the uniformity of the network allows the chaos algorithm to work very well, as there are no natural hot spots in the network. When small, local hot spots are added, chaotic routing deals

with them easily, whereas oblivious routing has more difficulties in this case. Secondly, the oblivious scheme requires special deadlock prevention mechanisms which create non-uniform access to the network and lead to degraded performance. Thus, on the torus network, chaotic routing provides significantly better performance than oblivious routing.

On a hypercube network, chaotic routing also achieves much better performance than oblivious routing. However, it still falls quite short of the bisection-bandwidth limited maximum throughput. When hot spots are added to the work load, the performance difference between chaotic and oblivious routing is even greater. Overall, on torus and hypercube network, chaotic routing gives excellent performance compared to oblivious routing. On the mesh network, chaotic routing provides only "competitive" performance, surpassing the performance of oblivious routing only when small, local hot spots are added to the applied traffic model.

With the performance of the detailed design confirmed by simulation, a chaos router chip was built to ensure that such a design can be implemented in hardware. The design of the chip involved using a mixed custom and standard cell approach. By carefully pipelining the functionality of the chip, a 4-cycle deep primary routing pipeline was built. This pipeline allows a packet's header to be routed off of the chip only four cycles after arriving on an input channel. When the network becomes more congested, though, packets may enter the multiqueue along a slower path. The multiqueue was built using a design that is somewhat slower than the primary cut-through data paths. This allows a significant savings in area and complexity at the cost of slowing down packets which enter the multiqueue. Thus, the primary data paths are built to be very fast, with the "secondary" multiqueue paths running more slowly. The result is a chip which, in simulations, is able to run with a 15ns clock cycle, limited in speed only by the technological limits of the pad drivers.

## *6.2 Remaining Work*

The work reported in this dissertation bridges the gap between the theoretical design of chaotic routing and the practical implementation of the chaos router chip. However, in order to bring the project to completion, much work remains. Although the chaos router chip is complete in design and has been fabricated, testing and characterization of the chip remain to be done. Also, several aspects of chaotic routing, such as fault

tolerance and performance under "real" loads, remain to be fully explored. These areas of remaining research are outlined below.

### *6.2.1 Chaos Chip Characterization*

Although a prototype chaos router chip has been completed, it remains to be tested. The testing of the chip will take place in several phases, with each successive phase testing the chip in an environment closer to what would be seen in a multicomputer environment.

The first phase is a functional test of the chip. Correct operation of all of the circuitry will be verified during this test. This test can be accomplished with the UW tester which provides an easily programmed testing interface. Unfortunately, because the tester cannot be operated at the 66MHz clock frequency of the router, speed testing will not be accomplished during this phase. If some components of the chip do not work, this functional test will be used to debug the chip and to help design a corrected version of the chip.

Speed and performance testing will be done in the second phase of chip characterization. Because the test will require hardware which operates faster than the available test equipment, it may be necessary to build custom test equipment which can inject traffic into a chaos router. A possible test set-up might resemble the simulation test circuitry described in Chapter 5.

The third phase of testing will examine the router chip's performance in a network of routers. These tests are meant to provide an idea of the performance of the chaos router in a network environment. As such, they will resemble the simulations presented in Chapter 4. Ideally, this test will be done by building a multicomputer with the chaos router or substituting the router into a currently existing multicomputer. However, this may be unrealistic and it may be necessary to build a network testing assembly. This assembly would contain possibly 16 chaos routers arranged in a mesh network. Each router's processor channel would be attached to injection/receiving logic which would simulate traffic from a processor. The edges of the mesh would also be attached to circuitry which would simulate traffic arriving from other, non-existent, parts of a large network. Designing and building such a test assembly will be difficult, but it is necessary in order to properly gauge the performance of the chaos router.

### *6.2.2 Broadened Simulation Analysis*

The simulations presented in Chapter 4 concentrated on chaotic routing on mesh, torus, and hypercube networks, primarily of 256 nodes. The current design of the prototype chaos router is based upon these results. Although these simulations build a solid foundation for the prototype router, a wider base of simulations to further understand the performance of chaotic routing is needed. A variety of traffic patterns other than the random and hot spot patterns presented in Chapter 4 should be simulated in order to gauge the performance of chaotic routing properly. Also, simulations based on traces from running programs would be useful. In order to efficiently run these simulations, it may be necessary to develop a parallel simulator to run on an existing parallel machine.

A second area of research is in the comparison of chaotic routing with other proposed routing schemes. The results of Chapter 4 compared chaotic routing with buffered oblivious routing and with deflection routing. The simulator should be extended to implement priority adaptive routing, various minimal adaptive schemes, and wormhole oblivious routing in order to compare the performance of these schemes with chaotic routing.

### *6.2.3 Fault Detection*

Although chaotic routing is much more fault-tolerant than oblivious or minimal adaptive schemes, it is still very beneficial to be able to detect the presence of faulty components in a network automatically. An outline for fault detection has been presented [Bolding & Snyder 91a]. The proposed mechanism uses the message re-ordering hardware, which is necessary anyway, to detect potentially "lost" packets. When a packet is missing, a quick diagnostic test is made which will detect any permanent (non-transient) faults. The hardware needed to do this is minimal and should not adversely affect performance.

This proposed design must be expanded to include more detail and then simulated and implemented to properly gauge its ability to detect faulty circuitry without penalizing performance when there are no faults present.

### *6.2.4 Fault Recovery*

There are two aspects of fault recovery which need to be explored in chaotic routing. The first is the "immediate" response to faults, before they have been detected by the fault-detection circuitry. Since a "dead" link will appear to a chaos router as an infinitely congested link, packets will be derouted around the faulty path automatically. However, performance may be significantly degraded since derouting is required for packets to be properly delivered. To gauge the performance of chaotic routing in the presence of faults, it is necessary to model and simulate a network with faults in it.

The second fault recovery issue is performance after fault detection. Once a fault has been detected simple mechanisms can be employed which keep packets from using faulty paths and immediately deroute packets which are waiting for a known faulty link. Thus, derouting can be made "immediately," without having to wait for a full queue and random selection when a fault is encountered. Once again, however, performance necessarily will be affected. Simulations are needed to gauge the impact of known faults on performance.

Finally, these fault recovery mechanisms should be implemented in a fault-tolerant version of the chaos router. As before, the goals for this implementation include correct and efficient delivery of all packets, while ensuring that the fault detection and recovery circuitry does not cause any degradation of performance when there are no faults present.

## *6.3 Summary*

A design for a practical and efficient non-minimal adaptive router for multicomputer networks has been presented in this dissertation. The design has been shown to have performance competitive with state-of-the-art routers, and, in general, to have higher throughput capabilities than other routers. The practical nature of this design has been confirmed by the design and implementation of a prototype chaos router chip which, in simulations, performs very well.

To complete the chaos router project, the prototype chip must be tested in several areas to ensure its correct function. Also, speed and network performance must be tested and characterized. To bring the project to maturity, more extensive simulations must be done to fine-tune the chaos design so that the design parameters can be chosen

based on the largest amount of information. Finally, aspects of fault-tolerance and recovery must be added to the design and simulated, resulting in a very efficient and reliable multicomputer router design.

# Bibliography

[Abraham & Padmanabhan 91] S. Abraham and K. Padmanabhan. Performance of multicomputer networks under pin-out constraints. *Journal of Parallel and Distributed Computing*, 12(3):237–248, July 1991.

[Abraham 90] S. Abraham. *Issues in the Architecture of Direct Interconnection Schemes for Multiprocessors.* PhD dissertation, University of Illinois, March 1990.

[Adve & Vernon 93] V. S. Adve and M. K. Vernon. Performance analysis of mesh interconnection networks with deterministic routing. *IEEE Transactions on Parallel and Distributed Systems*, 1993.

[Agarwal 91] A. Agarwal. Limits on interconnection network performance. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):398–412, October 1991.

[Bakoglu 90] H. Bakoglu. *Circuits, Interconnections, and Packaging for VLSI.* Addison-Wesley, 1990.

[Bhuyan 85] L. Bhuyan. An analysis of processor-memory interconnection networks. *IEEE Trans. on Computers*, C-34(3):279–283, March 1985.

[Bolding & Konstantinidou 92] K. Bolding and S. Konstantinidou. On the comparison of hypercube and torus networks. In *Proceedings of the 21st International Conference on Parallel Processing*, pages 62–66, August 1992.

[Bolding & Snyder 91a] K. Bolding and L. Snyder. Overview of fault handling for the chaos router. In *Proceedings of the 1991 IEEE International Workshop on Defect and Fault Tolerance in VLSI Systems*, pages 124–127. IEEE, November 1991.

[Bolding & Snyder 91b] K. Bolding and L. Snyder. Performance study of two-dimensional chaotic routers. Technical Report CSE-91-04-04, University of Washington, Seattle, WA, April 1991.

[Bolding & Snyder 92] K. Bolding and L. Snyder. Mesh and torus chaotic routing. In *Advanced Research in VLSI and Parallel Systems: Proceedings of the 1992 Brown/MIT Conference*, pages 333–347, March 1992.

[Bolding 92] K. Bolding. Non-uniformities introduced by virtual channel deadlock prevention. Technical Report CSE-92-07-07, University of Washington, Seattle, WA, July 1992.

[Bolding et al. 93] K. Bolding, S.-C. Cheung, S.-E. Choi, C. Ebeling, S. Hassoun, T. A. Ngo, and R. Wille. The chaos router chip: Design and implementation of an adaptive router. In *Proceedings of VLSI '93*, September 1993.

[Borgonovo & Cadorin 90] F. Borgonovo and E. Cadorin. Locally-optimal deflection routing in the bidirectional Manhattan network. In *Proceedings of IEEE INFOCOM '90*, pages 458–464. IEEE, June 1990.

[Borodin & Hopcroft 85] A. Borodin and J. E. Hopcroft. Routing, merging and sorting on parallel models of computation. *Journal of Computer and System Sciences*, 30:130–145, 1985.

[Cascade Design Automation Corporation 91] Cascade Design Automation Corporation. *ChipCrafter Designer's Handbook*, 1991.

[Chang & Simon 86] Y. Chang and J. Simon. Continuous routing and batch routing on the hypercube. In *Proceedings of Principles of Distributed Computing*, pages 272–281. ACM, 1986.

[Chen & Shin 90] M.-S. Chen and K. G. Shin. Adaptive fault-tolerant routing in hypercube multicomputers. *IEEE Transactions on Computers*, 39(12):1406–1416, December 1990.

[Chen 82] P. Y. Chen. *Multiprocessor Systems: Interconnection Networks, Memory Hierarchy, Modeling and Simulations*. PhD dissertation, University of Illinois, 1982.

[Chen et al. 81] P. Chen, D. Lawrie, and P. Yew. Interconnection networks using shuffles. *IEEE Computer*, pages 55–64, December 1981.

[Chien & Kim 92] A. A. Chien and J. H. Kim. Planar-adaptive routing: Low-cost adaptive networks for multiprocessors. In *Proc. 19th Intl. Symposium on Computer Architecture*, pages 268–277, May 1992.

[Chien 93] A. A. Chien. A cost and speed model for k-ary n-cube wormhole routers. In *Proceedings of Hot Interconnects '93*, August 1993.

[Chittor & Enbody 90] S. Chittor and R. Enbody. Hypercubes vs. 2D meshes. In *Proceedings SIAM Fourth Annual Conference on Parallel Processing for Scientific Computing*, pages 313–318, 1990.

[Coates et al. 93] B. Coates, A. Davis, and K. Stevens. The post office experience: Designing a large asynchronous chip. In *Proceedings of the HICSS*, 1993.

[Cypher & Gravano 92] R. Cypher and L. Gravano. Adaptive, deadlock-free packet routing in torus networks with minimal storage. In *Proc. 1992 Intl. Conf. on Parallel Processing*, pages 204–211, 1992.

[Dally & Seitz 86] W. Dally and C. Seitz. The torus routing chip. *Journal of Distributed Computing*, 1(3), 1986.

[Dally & Seitz 87] W. Dally and C. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Transactions on Computers*, C-36(5):547–553, May 1987.

[Dally 87] W. Dally. Wire-efficient VLSI multiprocessor communication networks. In P. Losleben, editor, *Proceedings of the Stanford Conference on Advanced Research in VLSI*, pages 391–415. MIT Press, March 1987.

[Dally 90] W. Dally. Network and processor architecture for message-driven computers. In Suaya and Birtwhistle, editors, *VLSI and Parallel Computation*. Morgan Kaufmann, 1990.

[Dally 92] W. Dally. Virtual-channel flow control. *IEEE Trans. on Parallel and Distributed Systems*, 3(2):194–205, March 1992.

[Dias & Jump 81a] D. M. Dias and J. R. Jump. Analysis and simulation of buffered delta networks. *IEEE Trans. on Computers*, C-30(4):273–282, April 1981.

[Dias & Jump 81b] D. M. Dias and J. R. Jump. Packet switching interconnection networks for modular systems. *Computer*, pages 43–53, December 1981.

[Duato 91] J. Duato. Deadlock-free adaptive routing algorithms for multicomputers. *Technique et Science Informatiques*, 10(4):275–285, 1991.

[Fang & Szymanski 91] C. Fang and T. Szymanski. An analysis of deflection routing in multi-dimensional regular mesh networks. In *Proceedings of IEEE INFOCOM '91*, pages 859–868. IEEE, April 1991.

[Flaig 87] C. Flaig. VLSI mesh routing systems. Master's thesis, California Institute of Technology, May 1987.

[Fulgham & Snyder 93] M. L. Fulgham and L. Snyder. A study of chaotic routing with nonuniform traffic. Technical Report CSE-93-06-01, University of Washington, Seattle, WA, June 1993.

[Gerlenter 81] D. Gerlenter. A DAG-based algorithm for prevention of store-and-forward deadlocks in packet networks. *IEEE Transactions on Computers*, C-30:709–715, October 1981.

[Gottlieb et al. 83] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir. The NYU ultracomputer – designing an MIMD shared memory parallel computer. *IEEE Trans. on Computers*, C-32(2):175–189, February 1983.

[Günther 81] K. Günther. Prevention of deadlocks in packet-switched data transport systems. *IEEE Transactions on Communication*, COM-29:512–524, April 1981.

[Hillis 85] W. D. Hillis. *The Connection Machine*. MIT Press, 1985.

[Hillis 86] W. D. Hillis. Method and apparatus for routing message packets. United States Patent 4,598,400, July 1986.

[H.Sullivan & Brashkow 77] H.Sullivan and T. Brashkow. A large scale homogeneous machine. In *Proceedings of the 4th Annual Symposium on Computer Architecture*, pages 105–124, 1977.

[Kaklamanis et al. 90] C. Kaklamanis, D. Krizanc, and T. Tsantilas. Tight bounds for oblivious routing in the hypercube. In *Proceedings of the 2nd Symposium on Parallel Algorithms and Architectures*, pages 31–35, 1990.

[Karol et al. 87] M. Karol, M. Hluchyj, and S. Morgan. Input versus output queueing on a space-division packet switch. *IEEE Transactions on Communication*, COM-35(12):1347–1356, December 1987.

[Kermani & Kleinrock 79] P. Kermani and L. Kleinrock. Virtual cut-through: A new computer communication switching technique. *Computer Networks*, 3:267–286, 1979.

[Konstantinidou & Snyder 90] S. Konstantinidou and L. Snyder. The chaos router: A practical application of randomization in network routing. In *Proceedings of the 2nd Symposium on Parallel Algorithms and Architectures*, pages 21–30. ACM, 1990.

[Konstantinidou & Snyder 91] S. Konstantinidou and L. Snyder. Chaos router: Architecture and performance. In *Proceedings of the 18th International Symposium on Computer Architecture*, pages 212–221. IEEE, May 1991.

[Konstantinidou 90] S. Konstantinidou. Adaptive, minimal routing in hypercubes. In *Proc. 6th MIT Conf. On Advanced Research in VLSI*, pages 139–153, 1990.

[Konstantinidou 91] S. Konstantinidou. *Deterministic and Chaotic Adaptive Routing in Multicomputers.* PhD dissertation, University of Washington, Seattle, WA, May 1991.

[Kruskal & Snir 83] C. Kruskal and M. Snir. The performance of multistage interconnection networks for multiprocessors. *IEEE Trans. on Computers*, C-32(12):1091–1098, December 1983.

[Ladner et al. 92] R. E. Ladner, A. LaMarca, and E. Tempero. Counting protocols for reliable end-to-end transmission. Technical Report CSE-92-10-06, University of Washington, Seattle, WA, October 1992.

[Linder & Hardin 91] D. H. Linder and J. C. Hardin. An adaptive and fault tolerant wormhole routing strategy for $k$-ary $n$-cubes. *IEEE Transactions on Computers*, C-40(1):2–12, January 1991.

[Maxemchuk 89] N. F. Maxemchuk. Comparison of deflection and store-and-forward techniques in the manhattan street and shuffle-exchange networks. In *Proceedings of IEEE INFOCOM '89*, pages 800–809. IEEE, 1989.

[Merlin & Schweitzer 80] P. Merlin and P. Schweitzer. Deadlock avoidance in store-and-forward networks – I: Store-and-forward deadlock. *IEEE Transactions on Communications*, COM-28:345–354, March 1980.

[Ngai & Seitz 89] J. Y. Ngai and C. L. Seitz. A framework for adaptive routing in multicomputer networks. In *Proceedings of the Symposium of Parallel Algorithms and Architectures*, pages 1–9. ACM, 1989.

[Ngai 89] J. Y. Ngai. *A Framework for Adaptive Routing in Multicomputer Networks.* PhD dissertation, California Institute of Technology, Pasadena, CA, May 1989.

[Nuth & Dally 92] P. R. Nuth and W. J. Dally. The J-machine network. In *Proceedings of 1992 IEEE Conference on Computer Design: VLSI in Computers and Processors*, pages 420–423, October 1992.

[Peleg & Upfal 90] D. Peleg and E. Upfal. A time-randomness trade-off for oblivious routing. *SIAM Journal of Computing*, 19(2):256–266, April 1990.

[Pertel 92a] M. J. Pertel. A critique of adaptive routing. Technical Report CS-TR-92-06, California Institute of Technology, Pasadena, CA, June 1992.

[Pertel 92b] M. J. Pertel. A simple simulator for multicomputer routing networks. Technical Report CS-TR-92-04, California Institute of Technology, Pasadena, CA, March 1992.

[Pfister & Norton 85] G. F. Pfister and V. A. Norton. "Hot spot" contention and combining in multistage interconnection networks. *IEEE Trans. on Computers*, C-34(10), October 1985.

[Pifarré et al. 91] G. D. Pifarré, L. Gravano, S. A. Felperin, and J. L. C. Sanz. Fully-adaptive minimal deadlock-free packet routing in hypercubes, meshes and other networks. In *Proc. 3rd ACM Symposium On Parallel Algorithms and Architectures*, pages 278–290, 1991.

[Seitz & Su 93] C. L. Seitz and W.-K. Su. A family of routing and communication chips based on the Mosaic. In *Symp. on Integrated Systems: Proc. of the 1993 Washington Conf.*, 1993.

[Smith 81] B. J. Smith. Architecture and applications of the HEP multiprocessor computer system. In *Proceedings of SPIE*, pages 241–248, 1981.

[Smitley 89] D. Smitley. Design tradeoffs for a high speed network node. Technical Report SRC-TR-89-007, Supercomputing Research Center Institute for Defense Analysis, Bowie, Maryland, July 1989.

[Szymanski 90] T. Szymanski. An analysis of "hot-potato" routing in a fiber optic packet switched hypercube. In *Proceedings of IEEE INFOCOM '90*, pages 918–925. IEEE, 1990.

[Thompson 79] C. Thompson. Area-time complexity for VLSI. In *Annual Symposium on Theory of Computing*, pages 81–88, May 1979.

[Toueg & Ullman 79] S. Toueg and J. Ullman. Deadlock-free packet-switching networks. In *Proceedings of the 11th ACM Symposium on Theory of Computation*, pages 88–98. ACM, 1979.

[Toueg 80] S. Toueg. Deadlock- and livelock-free packet switching networks. In *Proceedings of the 12th ACM Symposium on Theory of Computation*, pages 94–99, 1980.

[Upfal 84] E. Upfal. Efficient schemes for parallel communication. *Journal of the ACM*, 31(3):507–517, July 1984.

[Valiant & Brebner 81] L. G. Valiant and G. Brebner. Universal schemes for parallel communication. In *Proceedings of the 13th ACM Symposium on Theory of Computing*, pages 263–277, 1981.

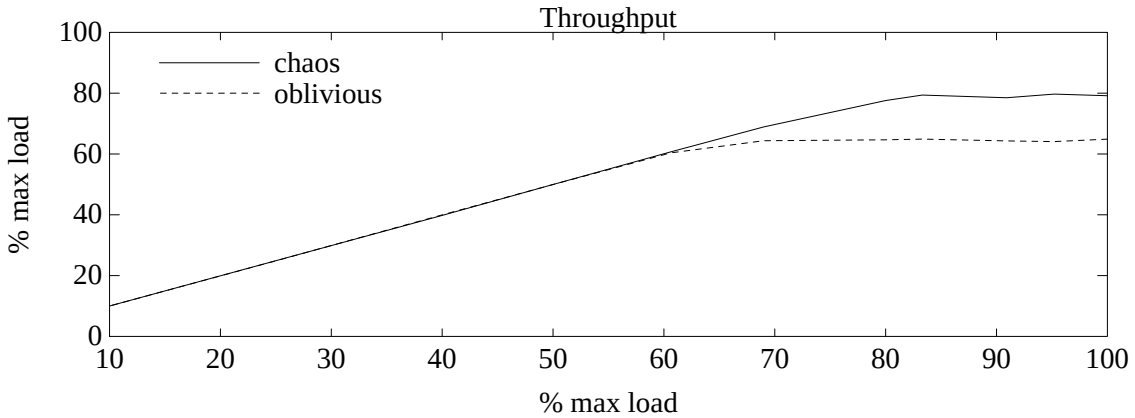
[Valiant 83] L. G. Valiant. Optimality of a two-phase strategy for routing in interconnection networks. *IEEE Trans. on Computers*, C-32(9):861–863, September 1983.

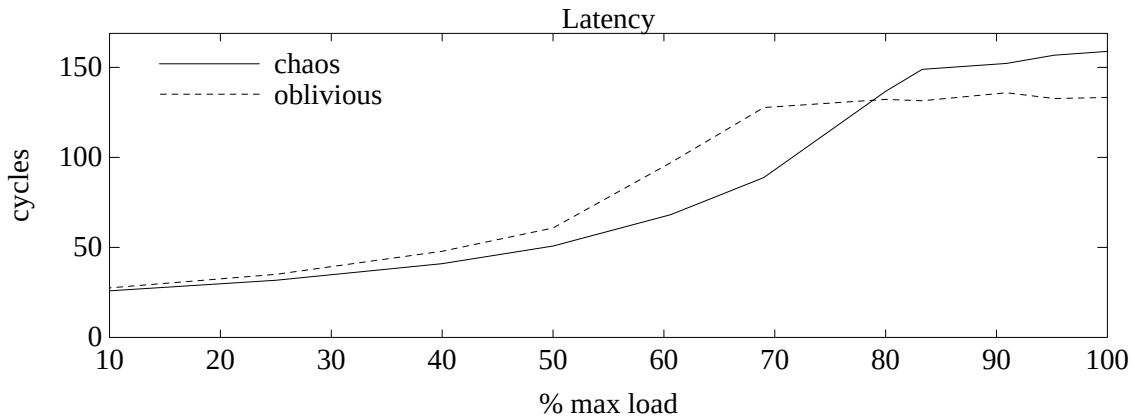
[Yew 81] P. C. Yew. *On the Design of Interconnection Networks for Parallel and Multiprocessor Systems*. PhD dissertation, University of Illinois, 1981.

Appendix A

# EXTENDED SIMULATION RESULTS – 64-NODE NETWORKS

This appendix contains simulation results for 64-node networks. Discussion of the results, as well as results for 256-node networks, can be found in Chapter 4.



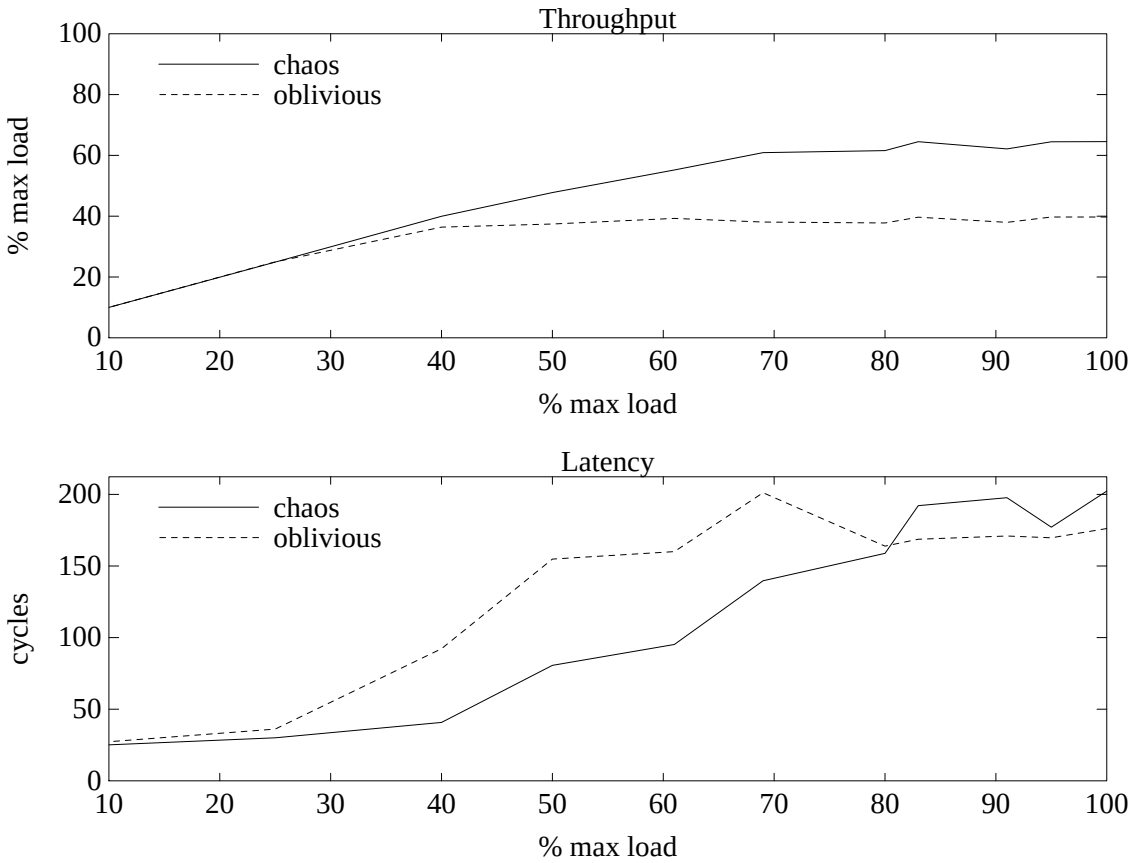

Chaos Routing

| % load | 10 | 25 | 40 | 50 | 61 | 69 | 80 | 83 | 91 | 95 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| mean xpt | 10.00 | 24.90 | 39.80 | 49.96 | 60.68 | 68.90 | 77.58 | 79.38 | 78.50 | 79.70 | 79.16 |
| std xpt | 0.00 | 0.00 | 0.18 | 0.14 | 0.10 | 0.46 | 1.09 | 0.79 | 0.64 | 1.03 | 0.95 |
| mean lat | 25.88 | 31.75 | 40.97 | 50.76 | 68.16 | 88.76 | 136.69 | 148.90 | 152.19 | 156.76 | 158.87 |
| std lat | 0.05 | 0.27 | 0.49 | 0.69 | 2.72 | 2.34 | 4.25 | 8.00 | 3.21 | 1.78 | 4.25 |

Oblivious Routing

| % load | 10 | 25 | 40 | 50 | 61 | 69 | 80 | 83 | 91 | 95 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| mean xpt | 10.00 | 24.86 | 39.96 | 49.96 | 60.36 | 64.36 | 64.66 | 64.90 | 64.30 | 64.08 | 64.88 |
| std xpt | 0.00 | 0.05 | 0.24 | 0.10 | 0.46 | 1.42 | 0.54 | 1.50 | 1.71 | 1.04 | 1.13 |
| mean lat | 27.52 | 35.04 | 47.86 | 60.78 | 97.01 | 127.69 | 132.21 | 131.46 | 135.90 | 132.70 | 133.28 |
| std lat | 0.10 | 0.30 | 0.50 | 1.23 | 5.14 | 6.29 | 2.18 | 2.78 | 2.33 | 1.96 | 1.23 |

Figure A.1: 64-node shared channel hypercube: chaos vs oblivious (Uniform traffic)



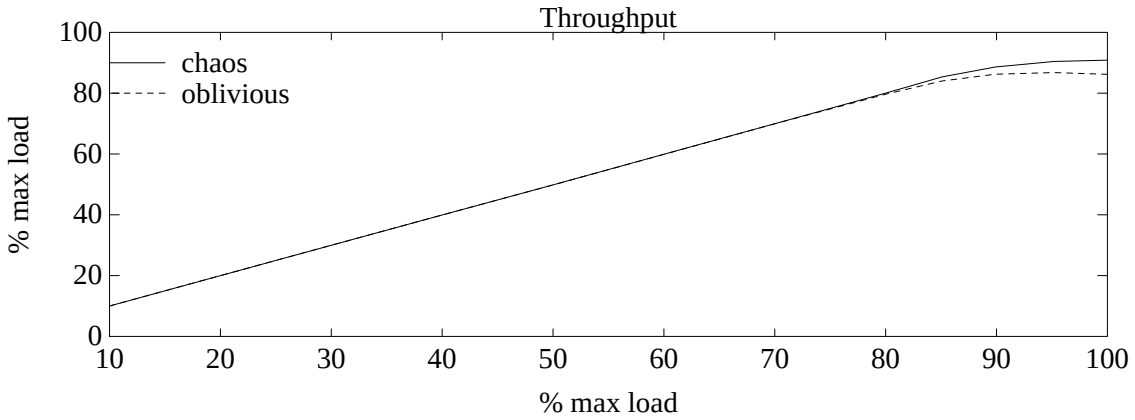

Chaos Routing (4X delivery rate)

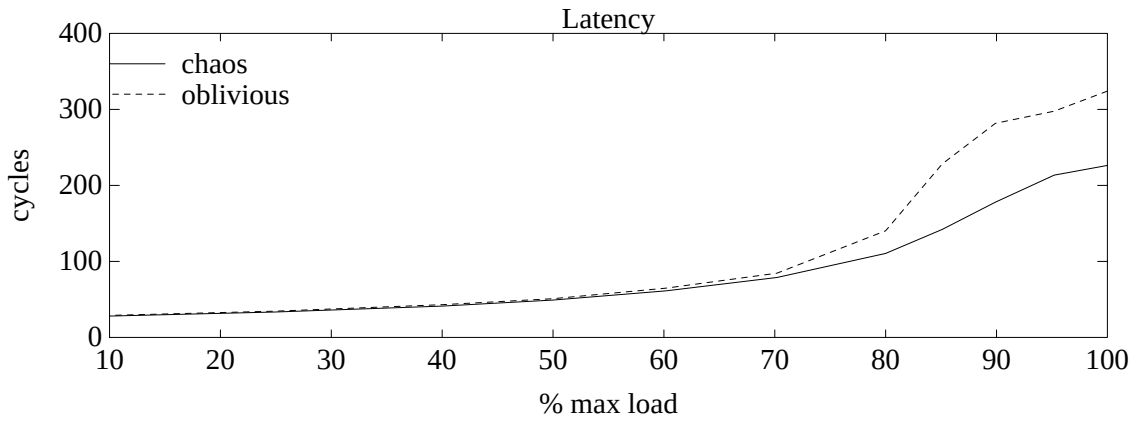
| % load | 10 | 25 | 40 | 50 | 61 | 69 | 80 | 83 | 91 | 95 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| mean xpt | 10.00 | 24.90 | 39.97 | 47.77 | 55.20 | 60.90 | 61.57 | 64.50 | 62.13 | 64.47 | 64.53 |
| std xpt | 0.00 | 0.00 | 0.34 | 2.95 | 7.28 | 9.33 | 10.61 | 11.82 | 14.38 | 12.00 | 12.83 |
| mean lat | 25.11 | 30.03 | 40.77 | 80.62 | 95.20 | 139.69 | 158.82 | 192.13 | 197.68 | 177.10 | 202.29 |
| std lat | 0.07 | 0.16 | 3.70 | 48.11 | 45.48 | 76.52 | 64.17 | 94.54 | 102.26 | 69.46 | 101.91 |

Oblivious Routing (4X delivery rate)

| % load | 10 | 25 | 40 | 50 | 61 | 69 | 80 | 83 | 91 | 95 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| mean xpt | 10.00 | 24.97 | 36.40 | 37.43 | 39.27 | 38.07 | 37.77 | 39.67 | 37.97 | 39.73 | 39.73 |
| std xpt | 0.00 | 0.05 | 2.98 | 4.72 | 6.58 | 5.54 | 5.60 | 5.62 | 6.28 | 3.66 | 4.74 |
| mean lat | 27.29 | 36.12 | 92.25 | 154.82 | 160.05 | 201.16 | 163.88 | 168.68 | 170.94 | 169.69 | 176.15 |
| std lat | 0.06 | 1.30 | 24.48 | 10.87 | 8.46 | 49.27 | 10.61 | 16.46 | 18.20 | 10.17 | 15.16 |

Figure A.2: 64-node shared channel hypercube: chaos vs oblivious (4X hot spot traffic)



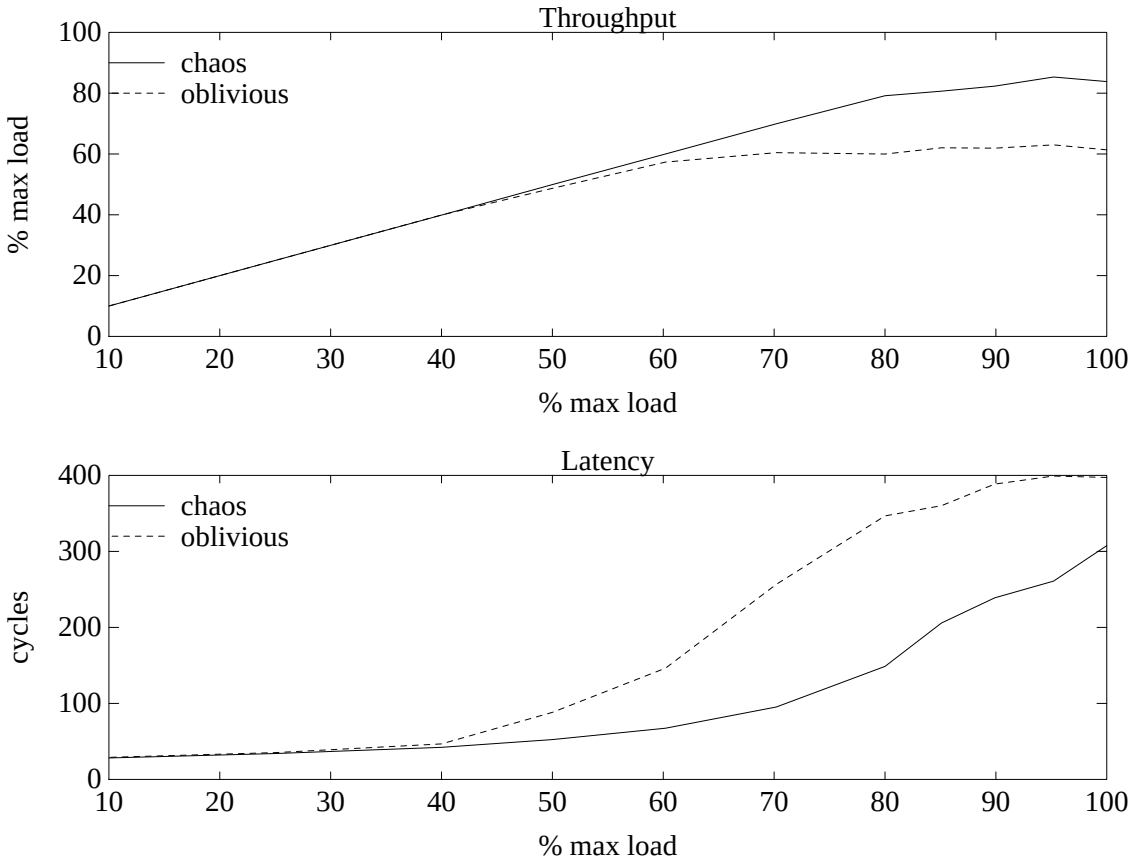

Chaos Routing

| % load | 10 | 25 | 40 | 50 | 60 | 70 | 80 | 85 | 90 | 95 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| mean xpt | 10.00 | 24.98 | 39.94 | 49.84 | 60.12 | 70.16 | 80.02 | 85.36 | 88.62 | 90.44 | 90.86 |
| std xpt | 0.00 | 0.04 | 0.08 | 0.05 | 0.13 | 0.14 | 0.21 | 0.44 | 1.56 | 0.61 | 0.61 |
| mean lat | 28.10 | 33.46 | 41.21 | 49.17 | 61.29 | 78.87 | 110.52 | 142.07 | 177.97 | 213.52 | 226.28 |
| std lat | 0.13 | 0.31 | 0.27 | 0.42 | 0.99 | 3.04 | 6.75 | 6.02 | 8.77 | 3.84 | 4.01 |

Oblivious Routing

| % load | 10 | 25 | 40 | 50 | 60 | 70 | 80 | 85 | 90 | 95 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| mean xpt | 10.00 | 25.00 | 39.94 | 49.82 | 60.08 | 70.14 | 79.66 | 84.02 | 86.24 | 86.78 | 86.20 |
| std xpt | 0.00 | 0.00 | 0.08 | 0.07 | 0.16 | 0.15 | 0.36 | 0.60 | 1.45 | 1.60 | 0.70 |
| mean lat | 29.00 | 34.58 | 43.08 | 50.97 | 64.85 | 84.53 | 140.30 | 228.21 | 281.70 | 297.65 | 324.07 |
| std lat | 0.06 | 0.25 | 0.36 | 0.65 | 1.62 | 5.03 | 15.64 | 18.37 | 31.15 | 11.47 | 12.95 |

Figure A.3: 64-node shared channel mesh: chaos vs oblivious (Uniform traffic)



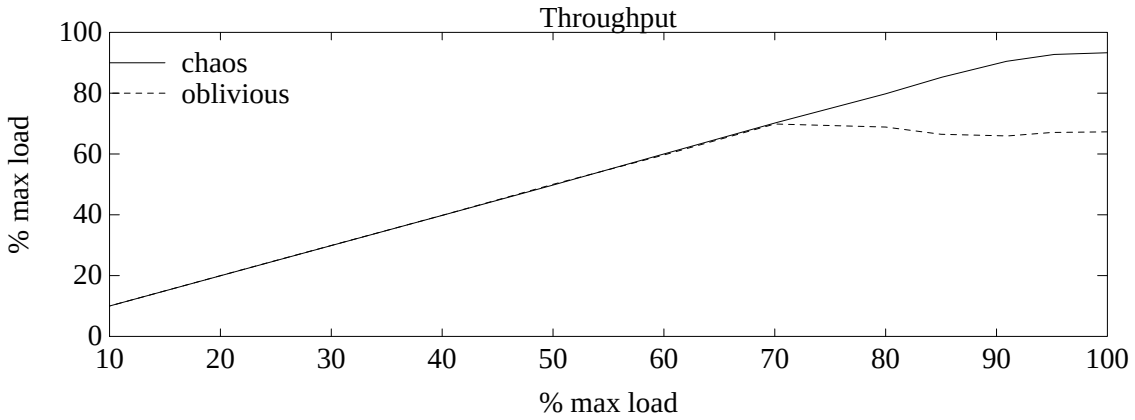

Chaos Routing

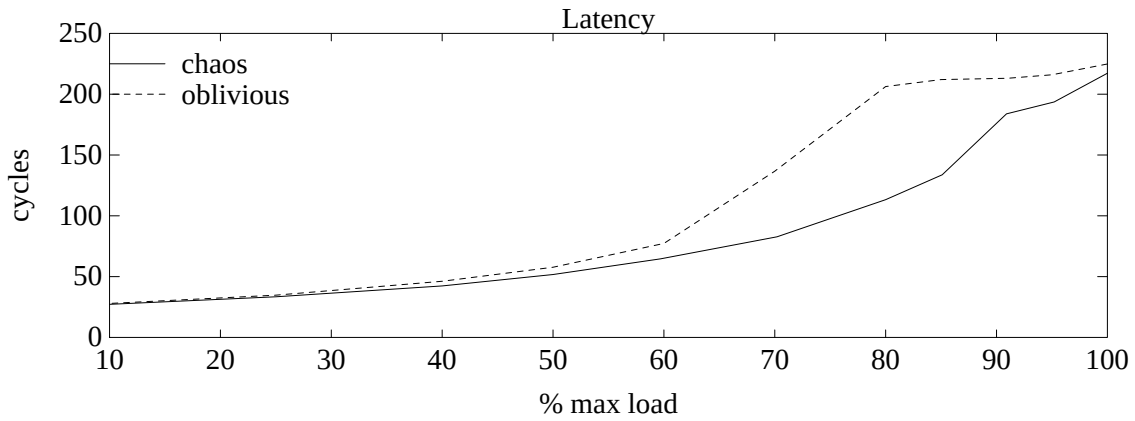
| % load | 10 | 25 | 40 | 50 | 60 | 70 | 80 | 85 | 90 | 95 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| mean xpt | 10.00 | 24.98 | 39.90 | 49.94 | 60.00 | 69.94 | 79.16 | 80.66 | 82.32 | 85.32 | 83.82 |
| std xpt | 0.00 | 0.04 | 0.00 | 0.08 | 0.21 | 0.54 | 2.41 | 3.53 | 4.16 | 4.78 | 5.89 |
| mean lat | 28.24 | 34.06 | 42.23 | 52.41 | 67.38 | 95.31 | 148.70 | 205.85 | 238.93 | 260.89 | 307.58 |
| std lat | 0.20 | 0.20 | 0.65 | 0.88 | 5.27 | 15.63 | 30.15 | 56.41 | 56.55 | 42.80 | 67.44 |

Oblivious Routing

| % load | 10 | 25 | 40 | 50 | 60 | 70 | 80 | 85 | 90 | 95 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| mean xpt | 10.00 | 24.98 | 39.94 | 48.70 | 57.34 | 60.44 | 59.98 | 62.04 | 61.92 | 63.00 | 61.36 |
| std xpt | 0.00 | 0.04 | 0.05 | 2.60 | 5.98 | 5.59 | 7.42 | 6.63 | 6.97 | 6.47 | 6.85 |
| mean lat | 29.04 | 35.27 | 46.74 | 88.24 | 146.34 | 256.72 | 346.68 | 360.21 | 388.57 | 399.17 | 397.32 |
| std lat | 0.17 | 0.43 | 3.08 | 64.80 | 87.60 | 56.14 | 70.48 | 25.55 | 39.23 | 18.99 | 20.95 |

Figure A.4: 64-node shared channel mesh: chaos vs oblivious (4X hot spot traffic)



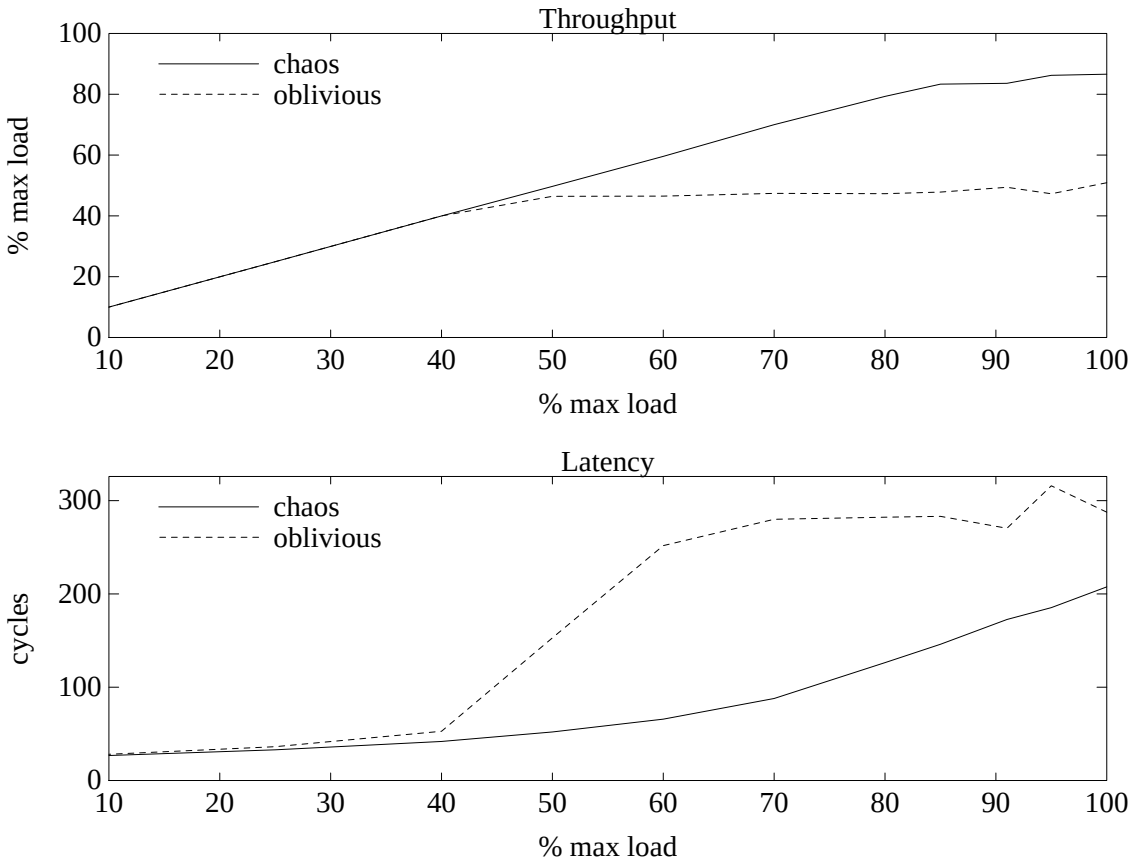

Chaos Routing

| % load | 10 | 25 | 40 | 50 | 60 | 70 | 80 | 85 | 91 | 95 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| mean xpt | 10.00 | 24.92 | 39.80 | 49.78 | 59.70 | 70.38 | 79.80 | 85.28 | 90.48 | 92.74 | 93.30 |
| std xpt | 0.00 | 0.04 | 0.00 | 0.15 | 0.17 | 0.16 | 0.30 | 0.50 | 1.10 | 1.29 | 1.57 |
| mean lat | 27.22 | 33.42 | 42.34 | 51.73 | 64.62 | 82.74 | 113.27 | 133.73 | 183.85 | 193.62 | 217.24 |
| std lat | 0.10 | 0.36 | 0.60 | 1.09 | 1.07 | 3.69 | 5.40 | 3.44 | 9.65 | 4.56 | 8.05 |

Oblivious Routing

| % load | 10 | 25 | 40 | 50 | 60 | 70 | 80 | 85 | 91 | 95 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| mean xpt | 10.00 | 24.96 | 39.82 | 50.02 | 59.68 | 69.86 | 68.86 | 66.48 | 65.94 | 67.08 | 67.28 |
| std xpt | 0.00 | 0.05 | 0.07 | 0.07 | 0.07 | 0.64 | 1.09 | 0.88 | 1.30 | 1.41 | 0.69 |
| mean lat | 27.94 | 34.75 | 46.18 | 57.72 | 77.29 | 136.69 | 206.29 | 212.02 | 213.10 | 215.97 | 224.85 |
| std lat | 0.06 | 0.31 | 0.64 | 1.59 | 3.23 | 10.57 | 5.77 | 3.49 | 5.61 | 1.61 | 6.59 |

Figure A.5: 64-node shared channel torus: chaos vs oblivious (Uniform traffic)



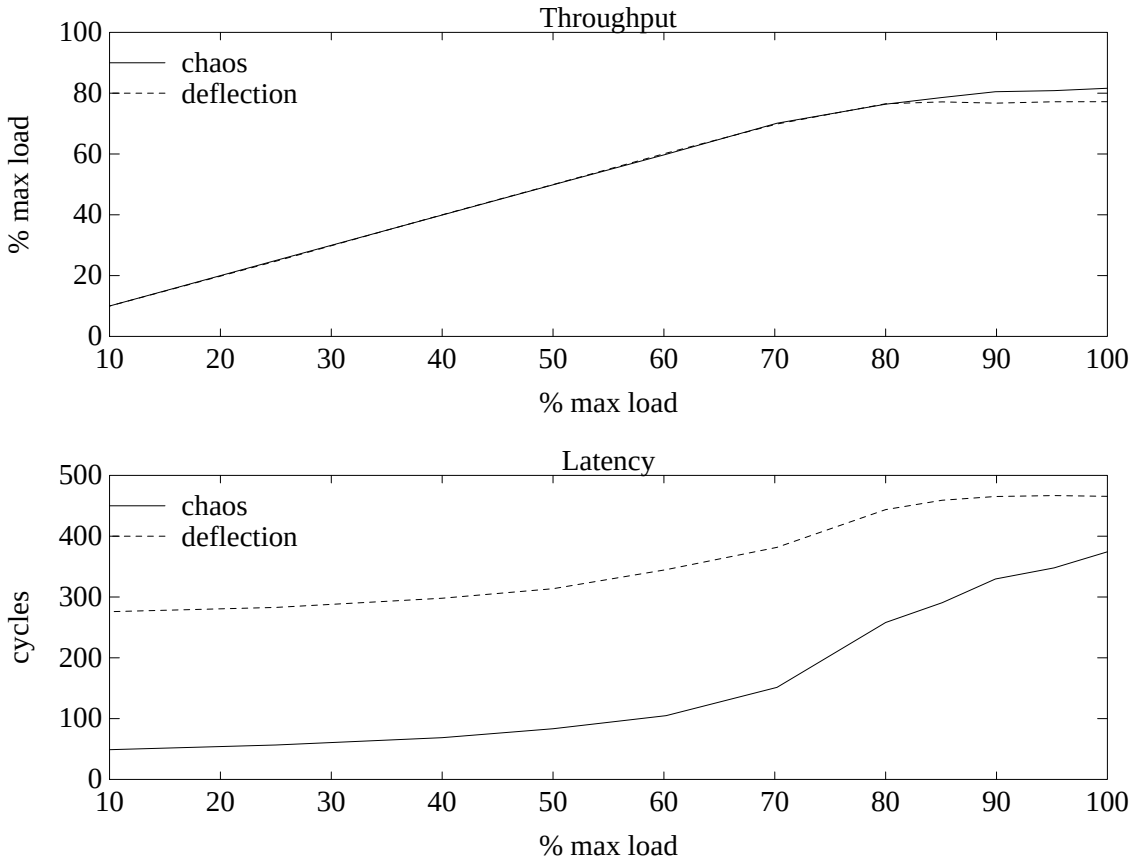

Chaos Routing (4X delivery rate)

| % load | 10 | 25 | 40 | 50 | 60 | 70 | 80 | 85 | 91 | 95 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| mean xpt | 10.00 | 24.93 | 39.97 | 49.70 | 59.57 | 70.00 | 79.30 | 83.33 | 83.60 | 86.23 | 86.60 |
| std xpt | 0.00 | 0.05 | 0.12 | 0.16 | 0.25 | 0.45 | 1.30 | 2.09 | 2.61 | 3.71 | 2.13 |
| mean lat | 26.78 | 32.90 | 41.86 | 52.04 | 65.77 | 87.91 | 126.36 | 146.04 | 172.60 | 185.32 | 207.55 |
| std lat | 0.05 | 0.10 | 0.22 | 1.20 | 2.73 | 4.46 | 7.55 | 9.70 | 12.09 | 11.23 | 6.20 |

Oblivious Routing (4X delivery rate)

| % load | 10 | 25 | 40 | 50 | 60 | 70 | 80 | 85 | 91 | 95 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| mean xpt | 10.00 | 24.93 | 40.00 | 46.43 | 46.50 | 47.40 | 47.30 | 47.83 | 49.40 | 47.27 | 50.90 |
| std xpt | 0.00 | 0.05 | 0.08 | 3.07 | 4.81 | 4.62 | 5.53 | 2.01 | 2.84 | 1.60 | 3.21 |
| mean lat | 28.10 | 36.25 | 52.77 | 152.88 | 251.78 | 280.01 | 282.25 | 283.16 | 270.38 | 315.90 | 287.63 |
| std lat | 0.04 | 0.22 | 1.86 | 53.41 | 16.69 | 27.64 | 26.12 | 7.70 | 27.60 | 14.48 | 18.87 |

Figure A.6: 64-node shared channel torus: chaos vs oblivious (4X hot spot traffic)



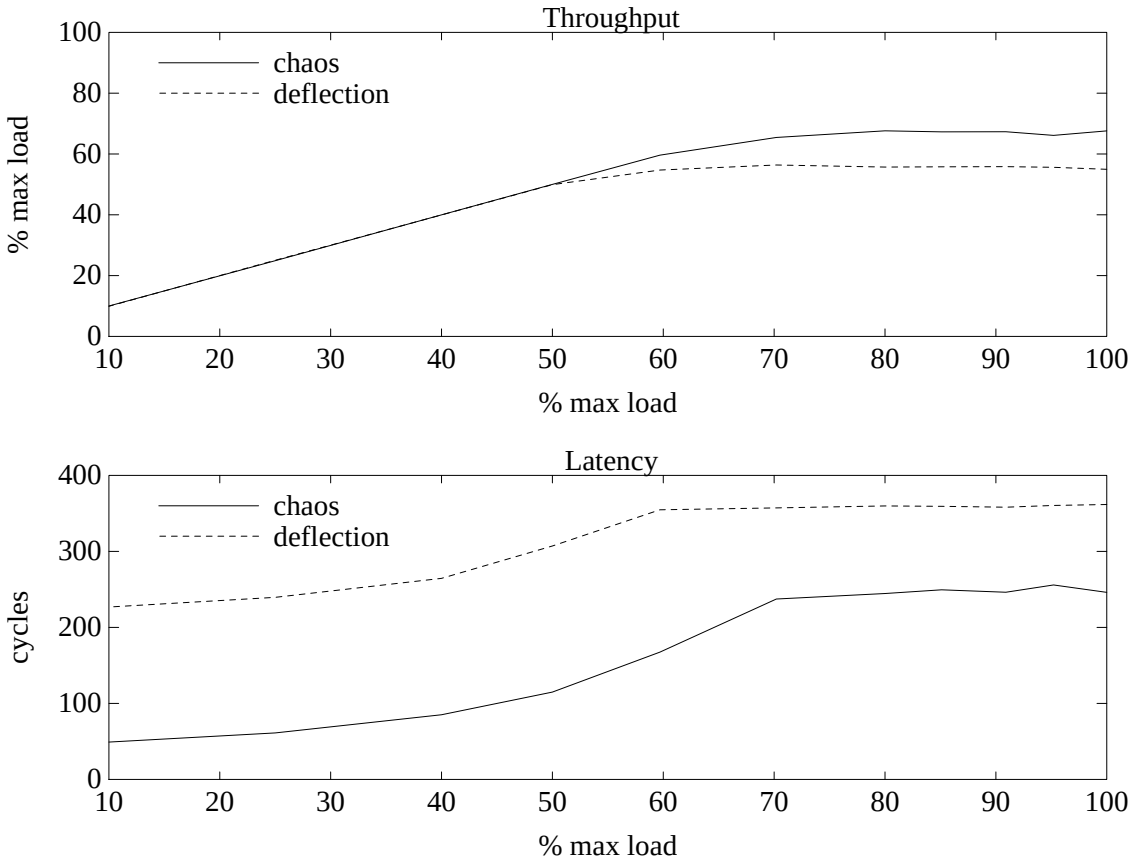

Chaos Routing

| % load | 10 | 25 | 40 | 50 | 60 | 70 | 80 | 85 | 90 | 95 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| mean xpt | 10.00 | 24.98 | 39.92 | 49.86 | 59.90 | 70.12 | 76.38 | 78.58 | 80.48 | 80.84 | 81.62 |
| std xpt | 0.00 | 0.04 | 0.04 | 0.14 | 0.33 | 0.17 | 1.33 | 1.16 | 0.97 | 1.24 | 1.17 |
| mean lat | 48.91 | 56.61 | 68.67 | 83.35 | 104.89 | 151.41 | 258.08 | 290.65 | 329.62 | 347.92 | 374.34 |
| std lat | 0.13 | 0.49 | 0.80 | 1.54 | 1.90 | 3.58 | 18.38 | 14.46 | 9.21 | 12.24 | 7.72 |

Deflection Routing

| % load | 10 | 25 | 40 | 50 | 60 | 70 | 80 | 85 | 90 | 95 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| mean xpt | 10.00 | 24.74 | 40.00 | 49.94 | 60.26 | 69.86 | 76.52 | 77.14 | 76.72 | 77.18 | 77.20 |
| std xpt | 0.00 | 0.05 | 0.09 | 0.15 | 0.30 | 0.22 | 1.09 | 1.16 | 0.91 | 1.28 | 0.97 |
| mean lat | 275.8 | 282.8 | 297.9 | 313.6 | 344.9 | 381.6 | 443.7 | 459.1 | 465.2 | 466.8 | 465.5 |
| std lat | 1.39 | 0.45 | 0.72 | 2.31 | 3.76 | 9.40 | 6.82 | 4.14 | 2.55 | 2.35 | 0.77 |

Figure A.7: 64-node nonshared channel mesh: chaos vs deflection (Uniform traffic)



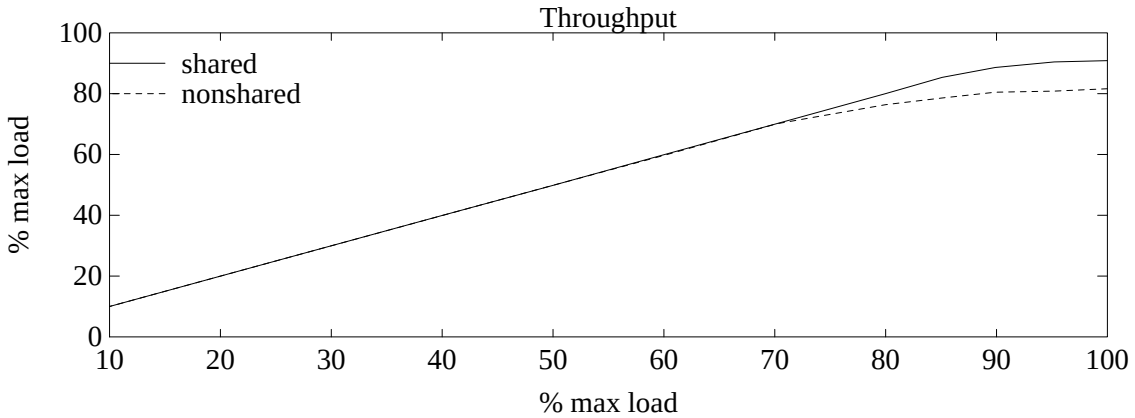

Chaos Routing

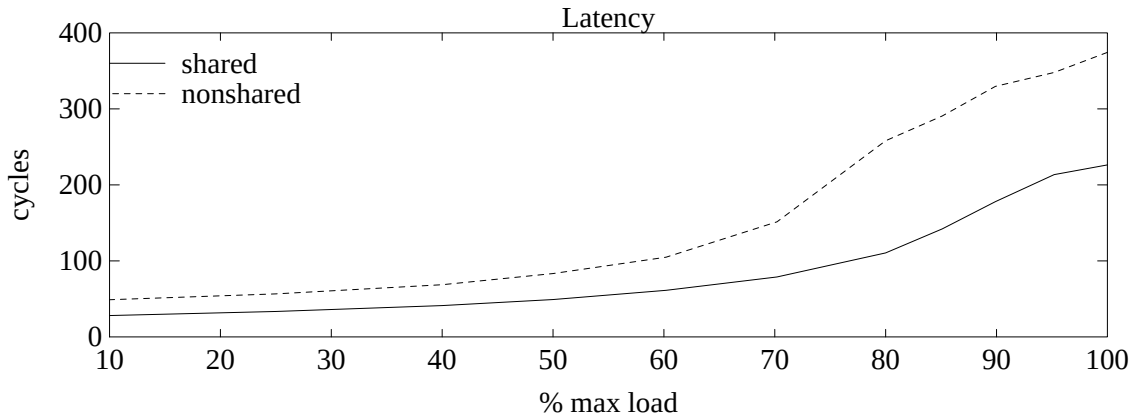
| % load | 10 | 25 | 40 | 50 | 60 | 70 | 80 | 85 | 91 | 95 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| mean xpt | 10.00 | 24.90 | 39.98 | 50.00 | 59.62 | 65.44 | 67.62 | 67.28 | 67.32 | 66.12 | 67.60 |
| std xpt | 0.00 | 0.00 | 0.20 | 0.13 | 0.13 | 0.23 | 0.89 | 1.25 | 1.20 | 0.77 | 0.67 |
| mean lat | 49.15 | 61.19 | 85.08 | 114.91 | 167.50 | 237.35 | 244.60 | 249.46 | 246.26 | 255.84 | 246.11 |
| std lat | 0.13 | 0.75 | 1.34 | 2.03 | 6.51 | 9.02 | 5.46 | 3.28 | 2.80 | 3.55 | 3.57 |

Deflection Routing

| % load | 10 | 25 | 40 | 50 | 60 | 70 | 80 | 85 | 91 | 95 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| mean xpt | 9.92 | 25.04 | 39.96 | 49.92 | 54.72 | 56.38 | 55.70 | 55.78 | 55.84 | 55.62 | 54.96 |
| std xpt | 0.04 | 0.10 | 0.05 | 0.19 | 0.56 | 1.30 | 0.58 | 1.12 | 0.94 | 1.29 | 0.98 |
| mean lat | 226.8 | 239.5 | 264.5 | 307.1 | 354.7 | 357.2 | 359.8 | 359.3 | 358.1 | 360.4 | 361.7 |
| std lat | 0.80 | 1.09 | 0.83 | 7.45 | 5.17 | 2.20 | 3.19 | 0.82 | 1.09 | 3.15 | 3.21 |

Figure A.8: 64-node nonshared channel torus: chaos vs deflection (Uniform traffic)



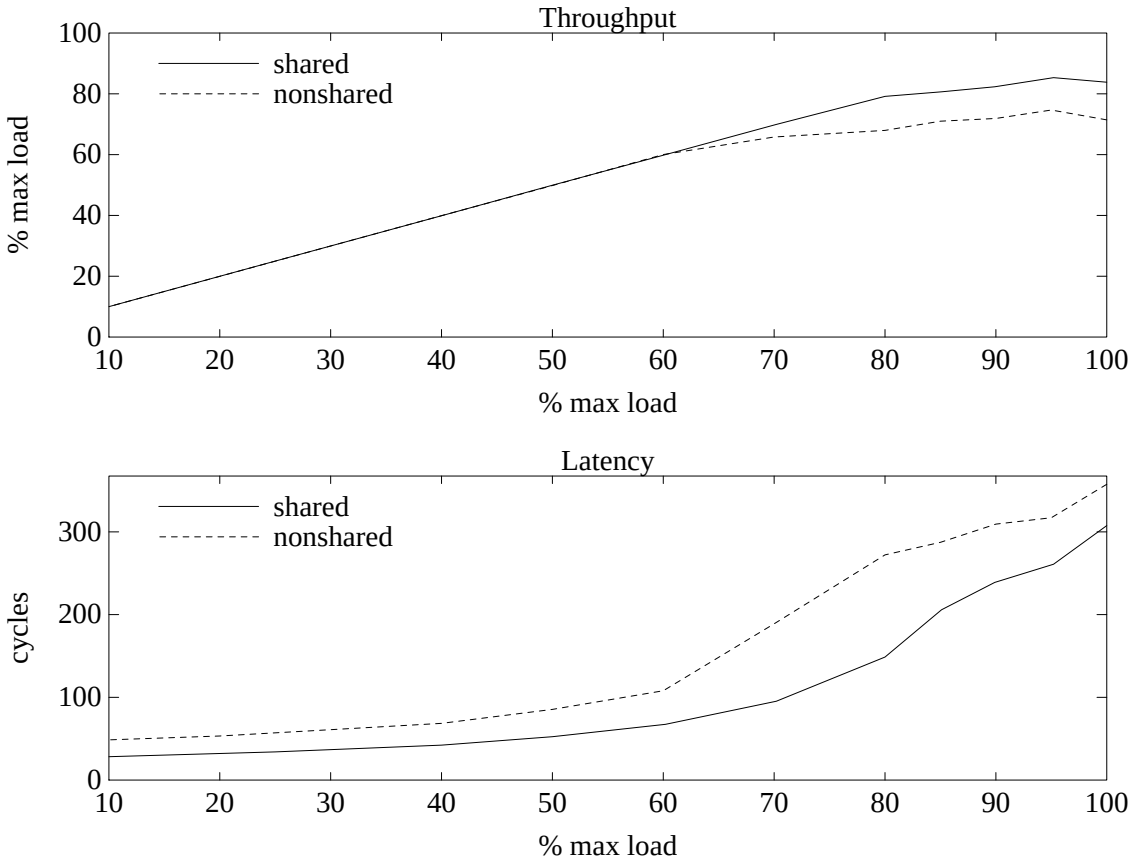

Shared channel

| % load | 10 | 25 | 40 | 50 | 60 | 70 | 80 | 85 | 90 | 95 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| mean xpt | 10.00 | 24.98 | 39.94 | 49.84 | 60.12 | 70.16 | 80.02 | 85.36 | 88.62 | 90.44 | 90.86 |
| std xpt | 0.00 | 0.04 | 0.08 | 0.05 | 0.13 | 0.14 | 0.21 | 0.44 | 1.56 | 0.61 | 0.61 |
| mean lat | 28.10 | 33.46 | 41.21 | 49.17 | 61.29 | 78.87 | 110.52 | 142.07 | 177.97 | 213.52 | 226.28 |
| std lat | 0.13 | 0.31 | 0.27 | 0.42 | 0.99 | 3.04 | 6.75 | 6.02 | 8.77 | 3.84 | 4.01 |

Nonshared channel

| % load | 10 | 25 | 40 | 50 | 60 | 70 | 80 | 85 | 90 | 95 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| mean xpt | 10.00 | 24.98 | 39.92 | 49.86 | 59.90 | 70.12 | 76.38 | 78.58 | 80.48 | 80.84 | 81.62 |
| std xpt | 0.00 | 0.04 | 0.04 | 0.14 | 0.33 | 0.17 | 1.33 | 1.16 | 0.97 | 1.24 | 1.17 |
| mean lat | 48.91 | 56.61 | 68.67 | 83.35 | 104.89 | 151.41 | 258.08 | 290.65 | 329.62 | 347.92 | 374.34 |
| std lat | 0.13 | 0.49 | 0.80 | 1.54 | 1.90 | 3.58 | 18.38 | 14.46 | 9.21 | 12.24 | 7.72 |

Figure A.9: 64-node chaos mesh: shared vs nonshared (Uniform traffic)



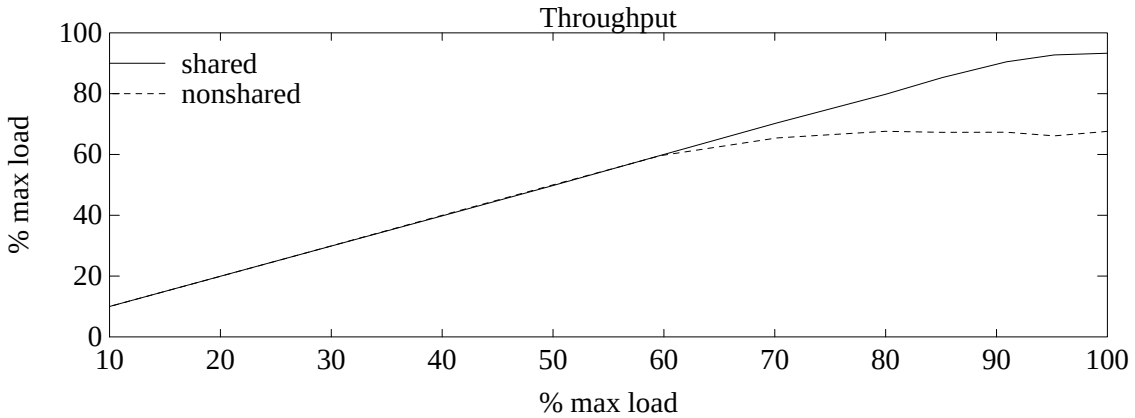

Shared channel

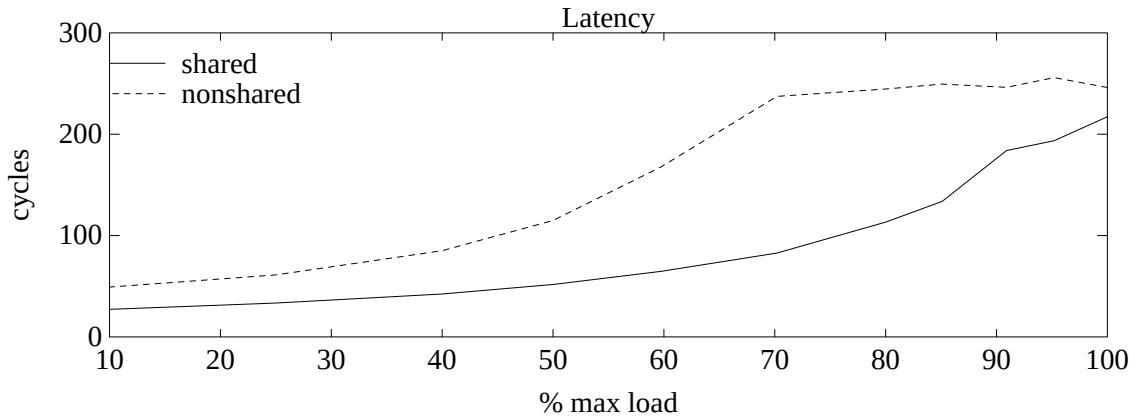
| % load | 10 | 25 | 40 | 50 | 60 | 70 | 80 | 85 | 90 | 95 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| mean xpt | 10.00 | 24.98 | 39.90 | 49.94 | 60.00 | 69.94 | 79.16 | 80.66 | 82.32 | 85.32 | 83.82 |
| std xpt | 0.00 | 0.04 | 0.00 | 0.08 | 0.21 | 0.54 | 2.41 | 3.53 | 4.16 | 4.78 | 5.89 |
| mean lat | 28.24 | 34.06 | 42.23 | 52.41 | 67.38 | 95.31 | 148.70 | 205.85 | 238.93 | 260.89 | 307.58 |
| std lat | 0.20 | 0.20 | 0.65 | 0.88 | 5.27 | 15.63 | 30.15 | 56.41 | 56.55 | 42.80 | 67.44 |

Nonshared channel (4X delivery rate)

| % load | 10 | 20 | 40 | 50 | 60 | 70 | 80 | 85 | 90 | 95 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| mean xpt | 10.00 | 19.97 | 39.93 | 49.83 | 60.03 | 65.80 | 67.97 | 71.00 | 71.90 | 74.67 | 71.43 |
| std xpt | 0.00 | 0.05 | 0.05 | 0.12 | 0.45 | 3.99 | 8.75 | 6.03 | 5.17 | 8.40 | 8.51 |
| mean lat | 48.59 | 53.27 | 68.55 | 85.41 | 107.96 | 188.89 | 272.19 | 287.39 | 309.38 | 317.10 | 357.50 |
| std lat | 0.20 | 0.25 | 1.15 | 1.70 | 5.75 | 46.31 | 98.60 | 59.78 | 67.69 | 71.62 | 91.28 |

Figure A.10: 64-node chaos mesh: shared vs nonshared (4X hot spot traffic)



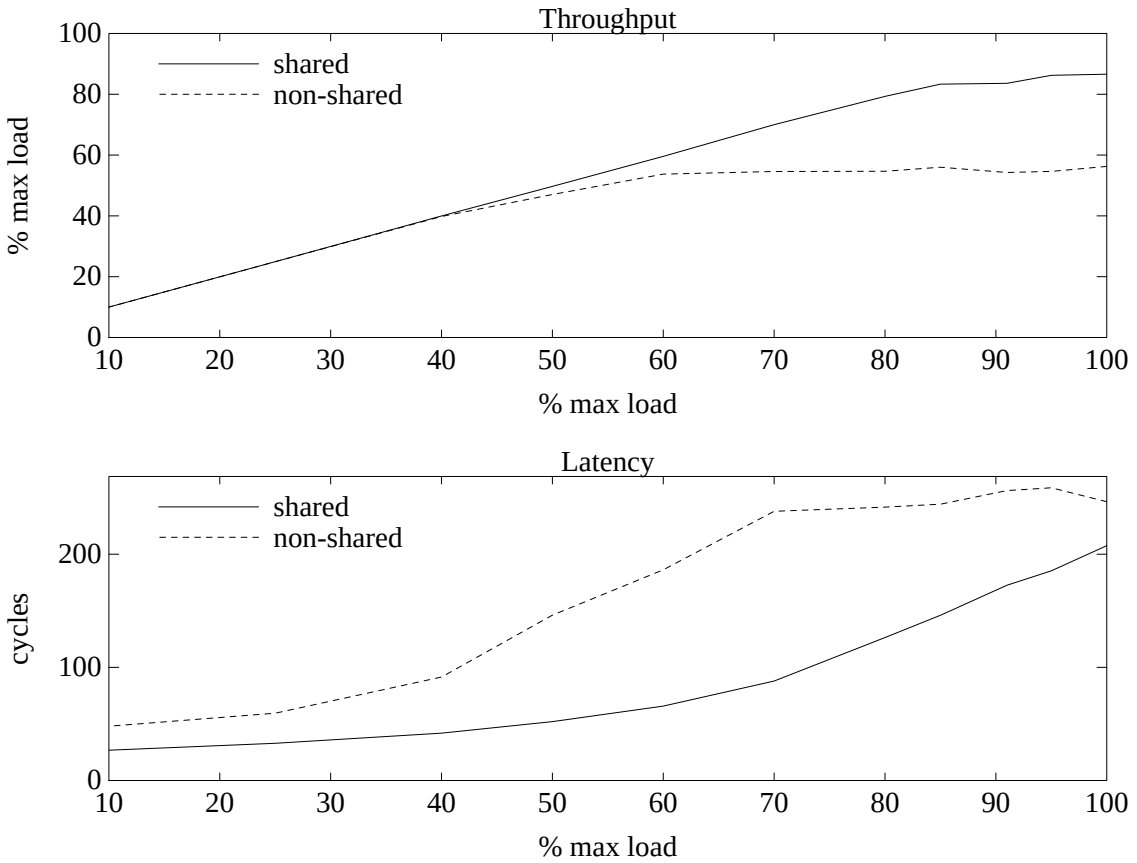

Shared channel

| % load | 10 | 25 | 40 | 50 | 60 | 70 | 80 | 85 | 91 | 95 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| mean xpt | 10.00 | 24.92 | 39.80 | 49.78 | 59.70 | 70.38 | 79.80 | 85.28 | 90.48 | 92.74 | 93.30 |
| std xpt | 0.00 | 0.04 | 0.00 | 0.15 | 0.17 | 0.16 | 0.30 | 0.50 | 1.10 | 1.29 | 1.57 |
| mean lat | 27.22 | 33.42 | 42.34 | 51.73 | 64.62 | 82.74 | 113.27 | 133.73 | 183.85 | 193.62 | 217.24 |
| std lat | 0.10 | 0.36 | 0.60 | 1.09 | 1.07 | 3.69 | 5.40 | 3.44 | 9.65 | 4.56 | 8.05 |

Nonshared channel

| % load | 10 | 25 | 40 | 50 | 60 | 70 | 80 | 85 | 91 | 95 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| mean xpt | 10.00 | 24.90 | 39.98 | 50.00 | 59.62 | 65.44 | 67.62 | 67.28 | 67.32 | 66.12 | 67.60 |
| std xpt | 0.00 | 0.00 | 0.20 | 0.13 | 0.13 | 0.23 | 0.89 | 1.25 | 1.20 | 0.77 | 0.67 |
| mean lat | 49.15 | 61.19 | 85.08 | 114.91 | 167.50 | 237.35 | 244.60 | 249.46 | 246.26 | 255.84 | 246.11 |
| std lat | 0.13 | 0.75 | 1.34 | 2.03 | 6.51 | 9.02 | 5.46 | 3.28 | 2.80 | 3.55 | 3.57 |

Figure A.11: 64-node chaos torus: shared vs nonshared (Uniform traffic)




Shared channel (4X delivery rate)

| % load | 10 | 25 | 40 | 50 | 60 | 70 | 80 | 85 | 91 | 95 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| mean xpt | 10.00 | 24.93 | 39.97 | 49.70 | 59.57 | 70.00 | 79.30 | 83.33 | 83.60 | 86.23 | 86.60 |
| std xpt | 0.00 | 0.05 | 0.12 | 0.16 | 0.25 | 0.45 | 1.30 | 2.09 | 2.61 | 3.71 | 2.13 |
| mean lat | 26.78 | 32.90 | 41.86 | 52.04 | 65.77 | 87.91 | 126.36 | 146.04 | 172.60 | 185.32 | 207.55 |
| std lat | 0.05 | 0.10 | 0.22 | 1.20 | 2.73 | 4.46 | 7.55 | 9.70 | 12.09 | 11.23 | 6.20 |

Nonshared channel (4X delivery rate)

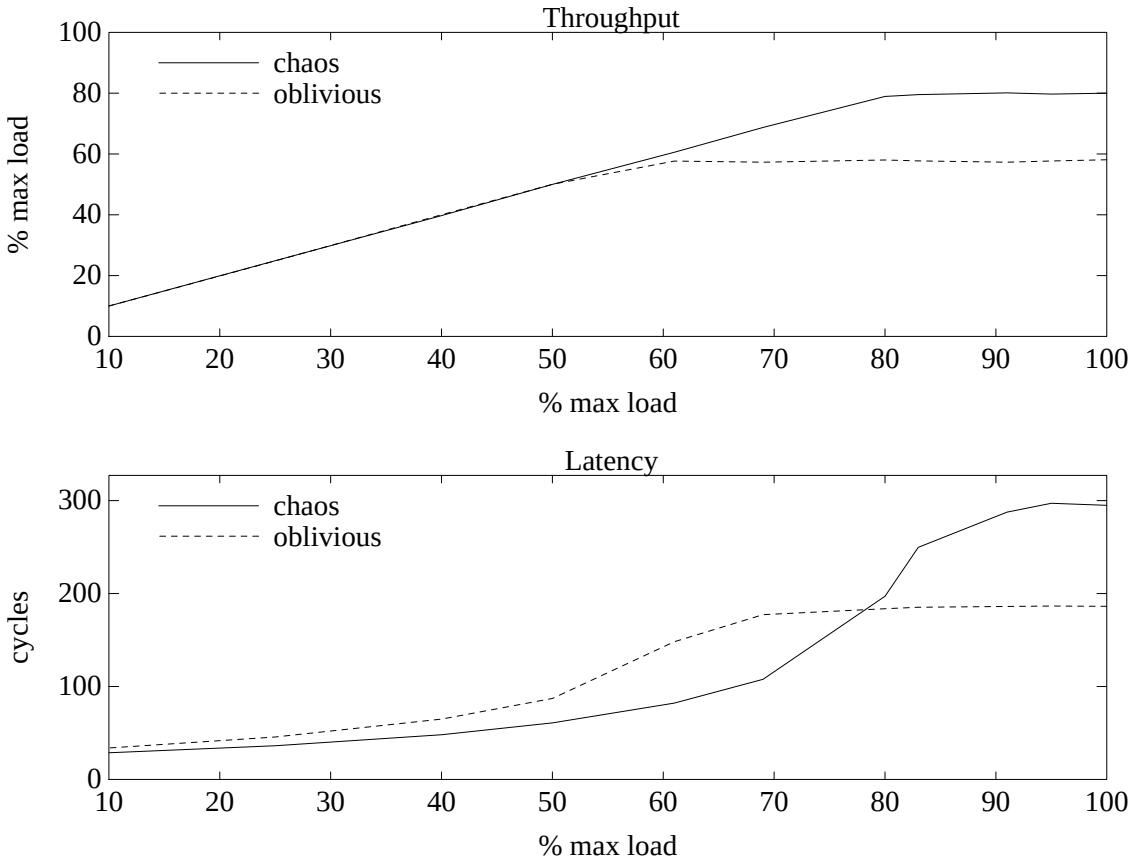
| % load | 10 | 25 | 40 | 50 | 60 | 70 | 80 | 85 | 91 | 95 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| mean xpt | 10.00 | 24.93 | 39.80 | 47.03 | 53.73 | 54.60 | 54.67 | 56.00 | 54.27 | 54.63 | 56.27 |
| std xpt | 0.00 | 0.05 | 0.42 | 4.20 | 6.18 | 7.29 | 7.63 | 7.99 | 9.11 | 8.39 | 6.84 |
| mean lat | 47.99 | 59.49 | 91.45 | 146.15 | 186.22 | 237.94 | 241.64 | 244.23 | 256.25 | 258.72 | 246.41 |
| std lat | 0.18 | 1.81 | 18.05 | 50.63 | 38.06 | 63.55 | 60.16 | 66.18 | 88.55 | 68.98 | 56.13 |

Figure A.12: 64-node chaos torus: shared vs nonshared (4X hot spot traffic)

Appendix B

# EXTENDED SIMULATION RESULTS – 1024-NODE NETWORKS

This appendix contains simulation results for 1024-node networks. Discussion of the results, as well as results for 256-node networks, can be found in Chapter 4.



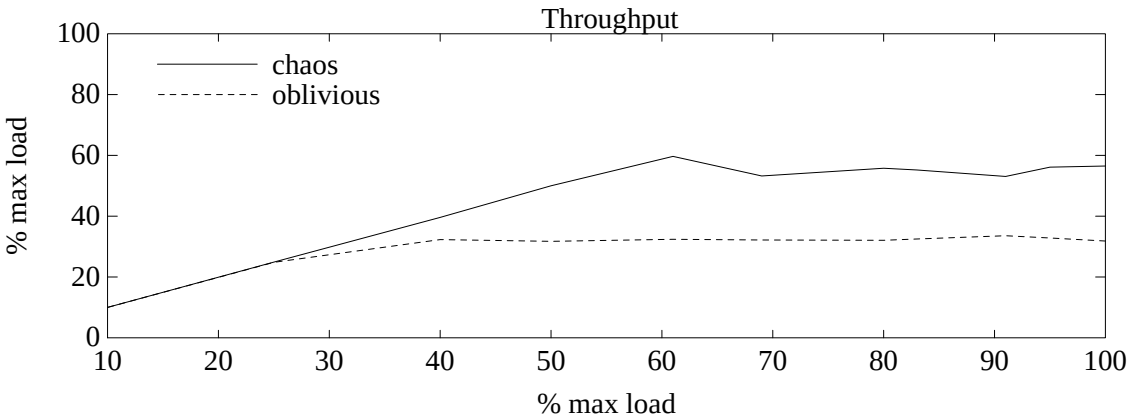

Chaos Routing

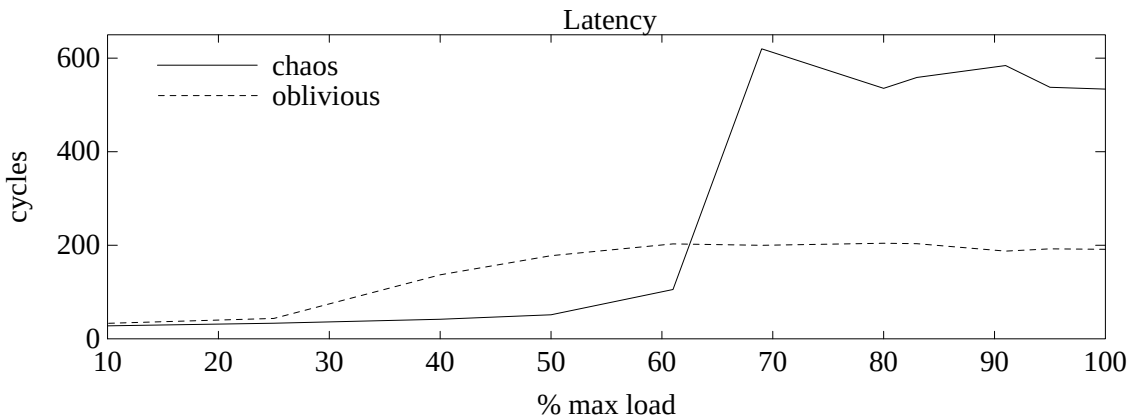
| % load | 10 | 25 | 40 | 50 | 61 | 69 | 80 | 83 | 91 | 95 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| mean xpt | 10.00 | 24.90 | 39.73 | 50.00 | 60.57 | 68.73 | 78.93 | 79.53 | 80.10 | 79.70 | 80.00 |
| std xpt | 0.00 | 0.00 | 0.19 | 0.00 | 0.05 | 0.09 | 0.12 | 0.48 | 0.08 | 0.28 | 0.28 |
| mean lat | 28.68 | 36.23 | 48.08 | 60.84 | 82.15 | 107.74 | 197.09 | 249.71 | 287.60 | 297.14 | 294.93 |
| std lat | 0.00 | 0.05 | 0.09 | 0.15 | 0.09 | 0.62 | 2.86 | 9.96 | 0.53 | 0.61 | 2.97 |

Oblivious Routing

| % load | 10 | 25 | 40 | 50 | 61 | 69 | 80 | 83 | 91 | 95 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| mean xpt | 10.00 | 24.83 | 40.00 | 50.03 | 57.67 | 57.30 | 58.03 | 57.73 | 57.30 | 57.70 | 58.10 |
| std xpt | 0.00 | 0.05 | 0.00 | 0.05 | 0.24 | 1.13 | 1.17 | 0.66 | 0.29 | 0.14 | 0.42 |
| mean lat | 33.83 | 45.66 | 64.92 | 87.11 | 148.18 | 177.29 | 183.67 | 185.26 | 186.11 | 186.58 | 186.26 |
| std lat | 0.02 | 0.01 | 0.07 | 0.23 | 4.78 | 3.32 | 1.97 | 1.05 | 1.65 | 0.68 | 2.12 |

Figure B.1: 1024-node shared channel hypercube: chaos vs oblivious (Uniform Traffic)



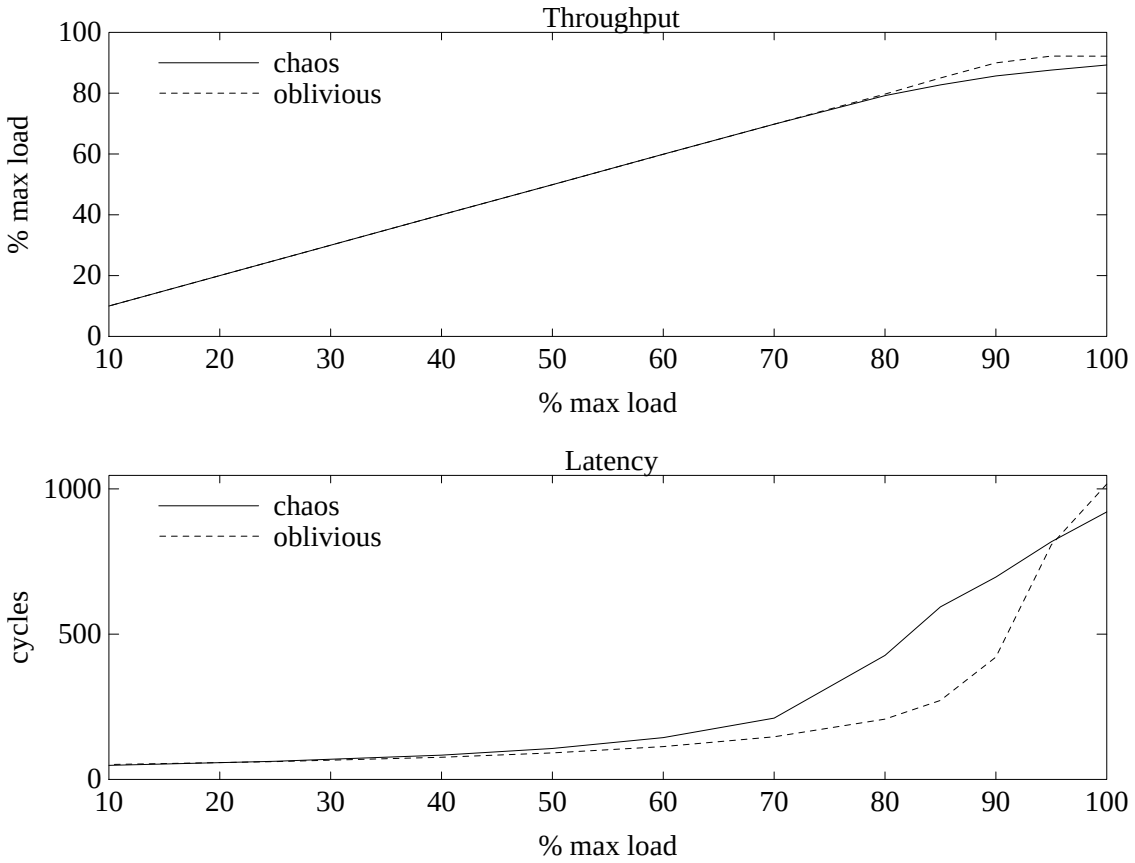

Chaos Routing (4X delivery rate)

| % load | 10 | 25 | 40 | 50 | 61 | 69 | 80 | 83 | 91 | 95 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| mean xpt | 10.00 | 24.90 | 39.60 | 50.00 | 59.67 | 53.23 | 55.77 | 55.20 | 53.07 | 56.13 | 56.50 |
| std xpt | 0.00 | 0.00 | 0.00 | 0.00 | 0.05 | 1.61 | 1.27 | 1.22 | 2.44 | 3.05 | 2.15 |
| mean lat | 27.73 | 33.32 | 41.80 | 51.39 | 105.41 | 619.95 | 535.46 | 558.73 | 584.20 | 537.73 | 533.83 |
| std lat | 0.01 | 0.05 | 0.07 | 0.34 | 22.03 | 29.38 | 27.46 | 27.91 | 48.94 | 56.67 | 48.09 |

Oblivious Routing (4X delivery rate)

| % load | 10 | 25 | 40 | 50 | 61 | 69 | 80 | 83 | 91 | 95 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| mean xpt | 10.00 | 24.83 | 32.30 | 31.73 | 32.40 | 32.17 | 32.07 | 32.50 | 33.57 | 32.83 | 31.83 |
| std xpt | 0.00 | 0.05 | 0.79 | 0.49 | 0.70 | 0.69 | 0.73 | 0.29 | 0.48 | 0.74 | 1.39 |
| mean lat | 33.12 | 43.59 | 136.81 | 177.73 | 202.91 | 199.90 | 204.29 | 203.37 | 187.48 | 192.32 | 191.35 |
| std lat | 0.04 | 0.02 | 4.05 | 3.83 | 12.85 | 6.60 | 13.09 | 3.24 | 4.01 | 5.14 | 5.51 |

Figure B.2: 1024-node shared channel hypercube: chaos vs oblivious (4X hot spot traffic)



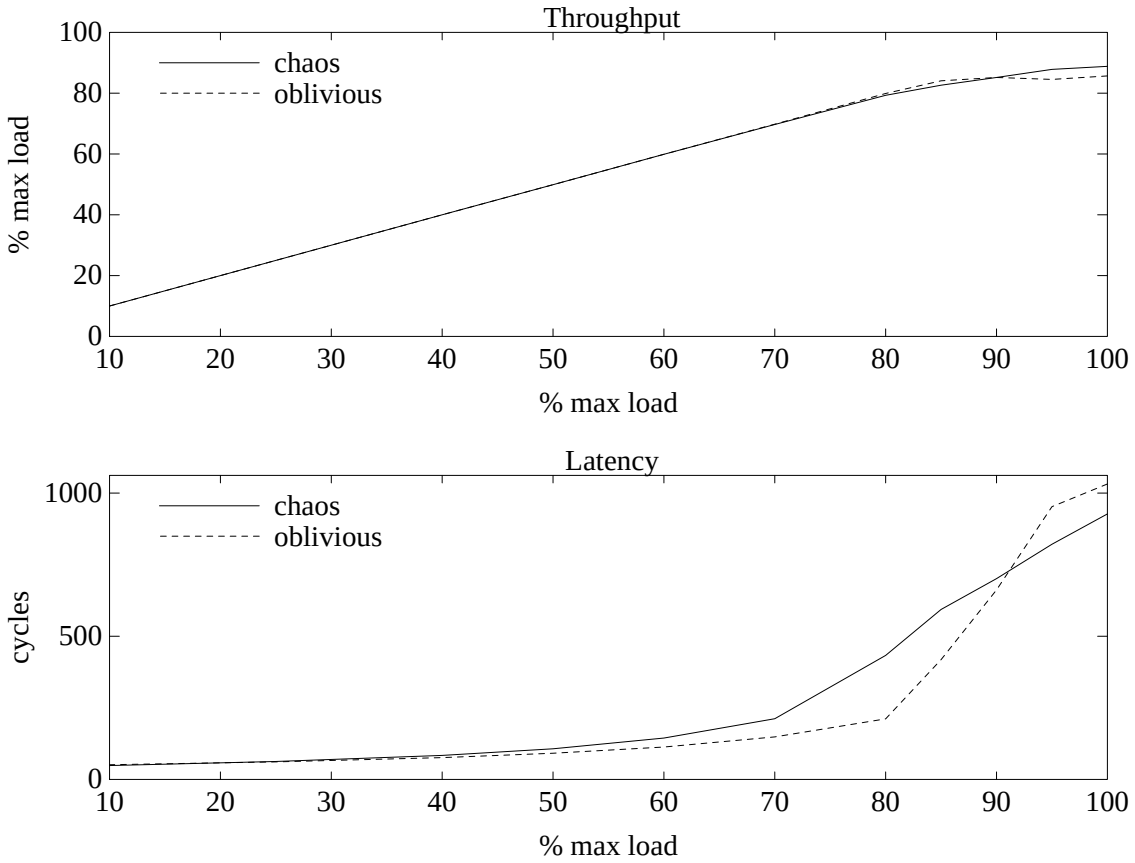

Chaos Routing

| % load | 10 | 25 | 40 | 50 | 60 | 70 | 80 | 85 | 90 | 95 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| mean xpt | 10.00 | 25.00 | 40.00 | 49.90 | 59.93 | 69.80 | 79.20 | 82.73 | 85.67 | 87.60 | 89.27 |
| std xpt | 0.00 | 0.00 | 0.00 | 0.00 | 0.05 | 0.14 | 0.08 | 0.09 | 0.33 | 0.16 | 0.29 |
| mean lat | 48.39 | 62.44 | 83.58 | 106.55 | 143.91 | 211.04 | 426.91 | 593.96 | 696.56 | 818.04 | 920.75 |
| std lat | 0.04 | 0.04 | 0.21 | 0.89 | 1.03 | 2.25 | 15.36 | 0.73 | 3.27 | 1.46 | 16.89 |

Oblivious Routing

| % load | 10 | 25 | 40 | 50 | 60 | 70 | 80 | 85 | 90 | 95 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| mean xpt | 10.00 | 25.00 | 40.00 | 49.90 | 59.90 | 69.80 | 79.70 | 85.00 | 89.97 | 92.20 | 92.17 |
| std xpt | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.22 | 0.00 | 0.17 | 0.37 | 0.25 |
| mean lat | 51.32 | 61.30 | 76.29 | 91.31 | 113.02 | 146.76 | 207.53 | 271.99 | 421.14 | 807.24 | 1016.45 |
| std lat | 0.07 | 0.04 | 0.33 | 0.25 | 0.71 | 1.02 | 2.20 | 3.57 | 18.02 | 11.44 | 21.61 |

Figure B.3: 1024-node shared channel mesh: chaos vs oblivious (Uniform traffic)



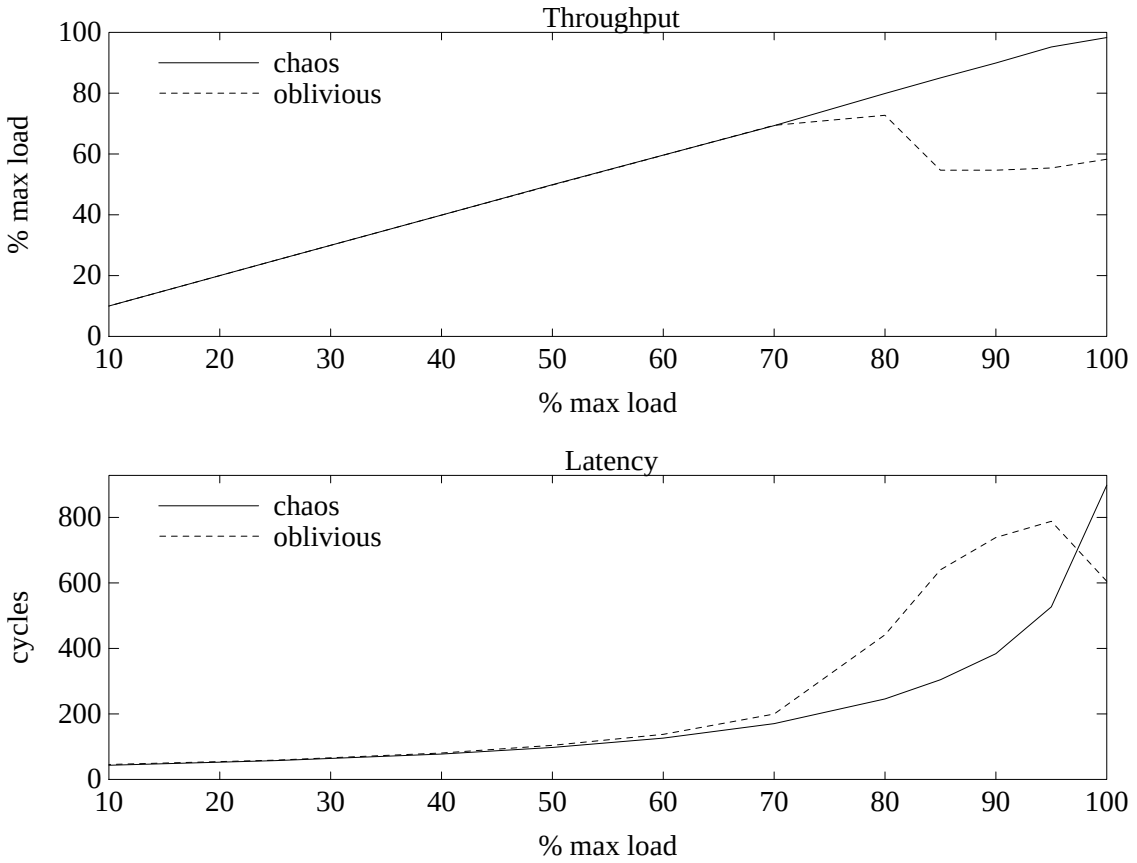

Chaos Routing

| % load | 10 | 25 | 40 | 50 | 60 | 70 | 80 | 85 | 90 | 95 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| mean xpt | 10.00 | 25.00 | 40.00 | 49.90 | 59.90 | 69.70 | 79.30 | 82.63 | 85.17 | 87.83 | 88.83 |
| std xpt | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.16 | 0.19 | 0.33 | 0.25 | 0.34 |
| mean lat | 48.46 | 62.68 | 83.90 | 106.97 | 144.48 | 212.02 | 433.12 | 593.56 | 701.25 | 821.30 | 927.16 |
| std lat | 0.07 | 0.04 | 0.17 | 0.65 | 0.45 | 3.85 | 14.46 | 5.52 | 5.35 | 6.58 | 11.48 |

Oblivious Routing

| % load | 10 | 25 | 40 | 50 | 60 | 70 | 80 | 85 | 90 | 95 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| mean xpt | 10.00 | 25.00 | 40.00 | 49.90 | 59.90 | 69.80 | 79.90 | 84.07 | 85.17 | 84.53 | 85.67 |
| std xpt | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.22 | 0.86 | 2.37 | 1.97 | 0.78 |
| mean lat | 51.31 | 61.30 | 76.18 | 91.44 | 113.14 | 148.51 | 211.49 | 418.36 | 661.83 | 952.56 | 1031.76 |
| std lat | 0.11 | 0.19 | 0.31 | 0.23 | 0.55 | 0.61 | 1.28 | 95.93 | 113.84 | 5.30 | 8.34 |

Figure B.4: 1024-node shared channel mesh: chaos vs oblivious (4X hot spot traffic)



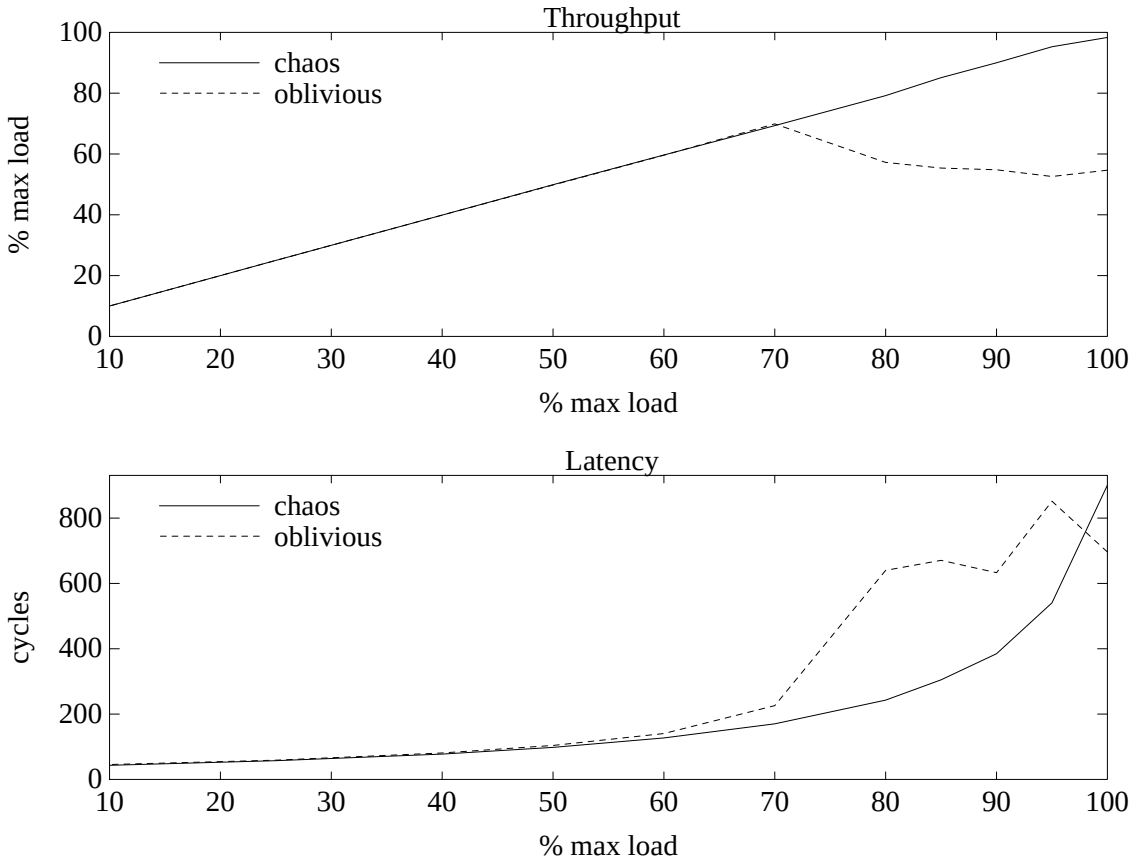

Chaos Routing

| % load | 10 | 25 | 40 | 50 | 60 | 70 | 80 | 85 | 90 | 95 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| mean xpt | 10.00 | 25.00 | 39.90 | 49.90 | 59.63 | 69.33 | 79.90 | 85.03 | 89.93 | 95.20 | 98.30 |
| std xpt | 0.00 | 0.00 | 0.00 | 0.00 | 0.05 | 0.05 | 0.08 | 0.05 | 0.12 | 0.29 | 0.50 |
| mean lat | 43.23 | 57.36 | 77.50 | 97.64 | 126.13 | 170.39 | 245.77 | 304.39 | 384.02 | 526.87 | 898.34 |
| std lat | 0.01 | 0.10 | 0.16 | 0.43 | 1.27 | 0.74 | 0.44 | 1.37 | 3.34 | 13.05 | 39.15 |

Oblivious Routing

| % load | 10 | 25 | 40 | 50 | 60 | 70 | 80 | 85 | 90 | 95 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| mean xpt | 10.00 | 25.00 | 39.90 | 49.80 | 59.60 | 69.43 | 72.73 | 54.67 | 54.67 | 55.40 | 58.27 |
| std xpt | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.33 | 1.18 | 3.63 | 2.25 | 2.77 | 5.78 |
| mean lat | 45.64 | 58.80 | 80.65 | 104.09 | 137.68 | 199.66 | 441.70 | 640.12 | 738.73 | 788.16 | 605.01 |
| std lat | 0.03 | 0.15 | 0.20 | 0.48 | 0.92 | 1.36 | 9.97 | 28.51 | 131.94 | 146.23 | 117.55 |

Figure B.5: 1024-node shared channel torus: chaos vs oblivious (Uniform traffic)



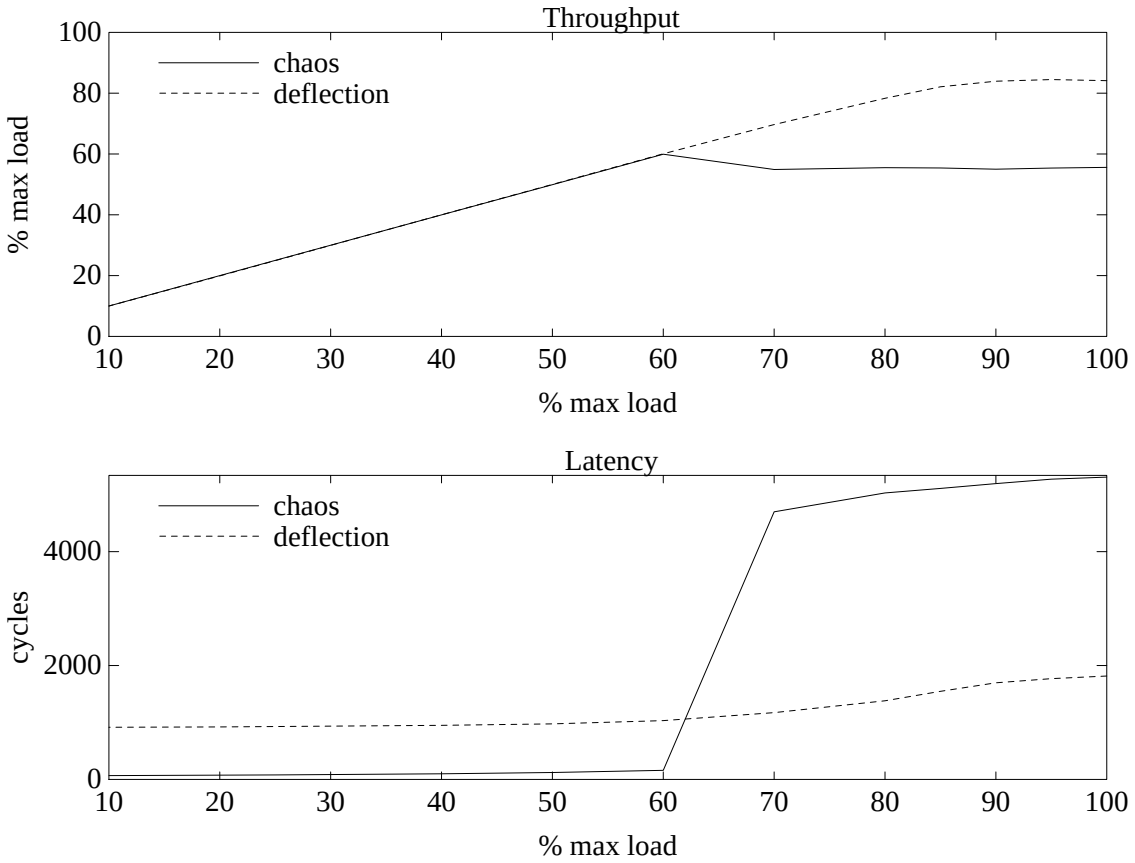

Chaos Routing

| % load | 10 | 25 | 40 | 50 | 60 | 70 | 80 | 85 | 90 | 95 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| mean xpt | 10.00 | 25.00 | 39.90 | 49.90 | 59.67 | 69.33 | 79.20 | 85.10 | 90.00 | 95.27 | 98.33 |
| std xpt | 0.00 | 0.00 | 0.00 | 0.00 | 0.05 | 0.05 | 0.36 | 0.08 | 0.08 | 0.21 | 0.31 |
| mean lat | 43.21 | 57.35 | 77.55 | 98.07 | 127.03 | 170.21 | 242.79 | 304.91 | 384.79 | 540.51 | 900.73 |
| std lat | 0.04 | 0.11 | 0.29 | 0.36 | 0.88 | 0.44 | 1.22 | 1.99 | 1.30 | 11.49 | 29.34 |

Oblivious Routing

| % load | 10 | 25 | 40 | 50 | 60 | 70 | 80 | 85 | 90 | 95 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| mean xpt | 10.00 | 25.00 | 39.90 | 49.80 | 59.60 | 69.90 | 57.23 | 55.37 | 54.80 | 52.60 | 54.67 |
| std xpt | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.08 | 3.93 | 3.90 | 2.64 | 1.77 | 0.83 |
| mean lat | 45.64 | 58.94 | 81.03 | 104.04 | 140.51 | 225.90 | 640.02 | 670.64 | 632.84 | 851.68 | 696.35 |
| std lat | 0.05 | 0.17 | 0.30 | 0.63 | 1.36 | 12.71 | 107.53 | 114.69 | 78.92 | 111.51 | 117.68 |

Figure B.6: 1024-node shared channel torus: chaos vs oblivious (4X hot spot traffic)



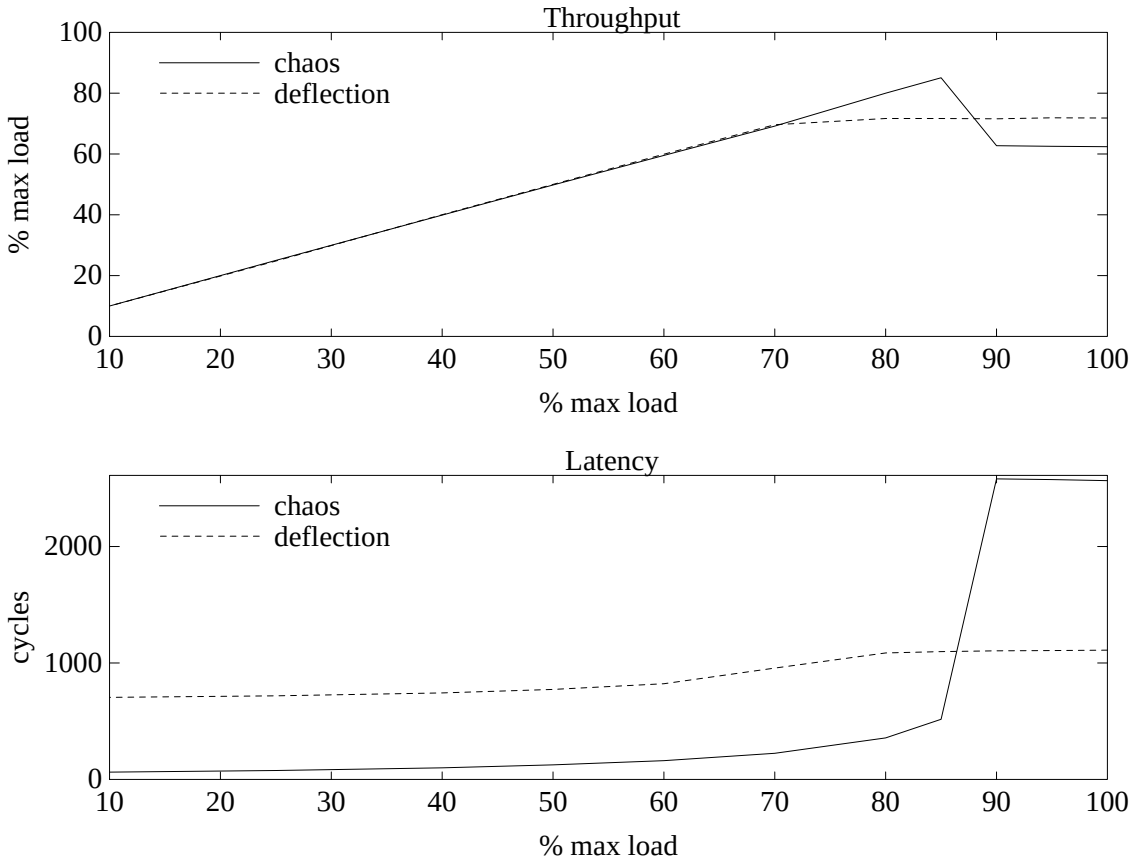

Chaos Routing

| % load | 10 | 25 | 40 | 50 | 60 | 70 | 80 | 85 | 90 | 95 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| mean xpt | 10.00 | 25.00 | 39.97 | 49.90 | 59.93 | 54.90 | 55.50 | 55.40 | 55.00 | 55.37 | 55.60 |
| std xpt | 0.00 | 0.00 | 0.05 | 0.00 | 0.05 | 0.37 | 0.51 | 0.29 | 0.24 | 0.54 | 0.29 |
| mean lat | 67.01 | 79.13 | 98.88 | 121.4 | 159.6 | 4699.2 | 5030.9 | 5110.2 | 5194.9 | 5272.7 | 5309.0 |
| std lat | 0.10 | 0.08 | 0.16 | 0.59 | 1.35 | 80.72 | 59.74 | 47.73 | 77.51 | 78.29 | 152.25 |

Deflection Routing

| % load | 10 | 20 | 40 | 50 | 60 | 70 | 80 | 85 | 90 | 95 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| mean xpt | 10.00 | 19.90 | 40.00 | 49.98 | 60.06 | 69.68 | 78.32 | 82.10 | 83.94 | 84.48 | 84.12 |
| std xpt | 0.00 | 0.00 | 0.00 | 0.04 | 0.05 | 0.07 | 0.12 | 0.41 | 0.29 | 0.20 | 0.10 |
| mean lat | 915.9 | 921.9 | 949.8 | 975.4 | 1032.7 | 1173.0 | 1381.1 | 1546.9 | 1696.8 | 1769.7 | 1815.8 |
| std lat | 0.85 | 0.71 | 1.41 | 0.85 | 2.35 | 7.26 | 4.49 | 29.23 | 23.93 | 7.43 | 3.34 |

Figure B.7: 1024-node nonshared channel mesh: chaos vs deflection (Uniform traffic)



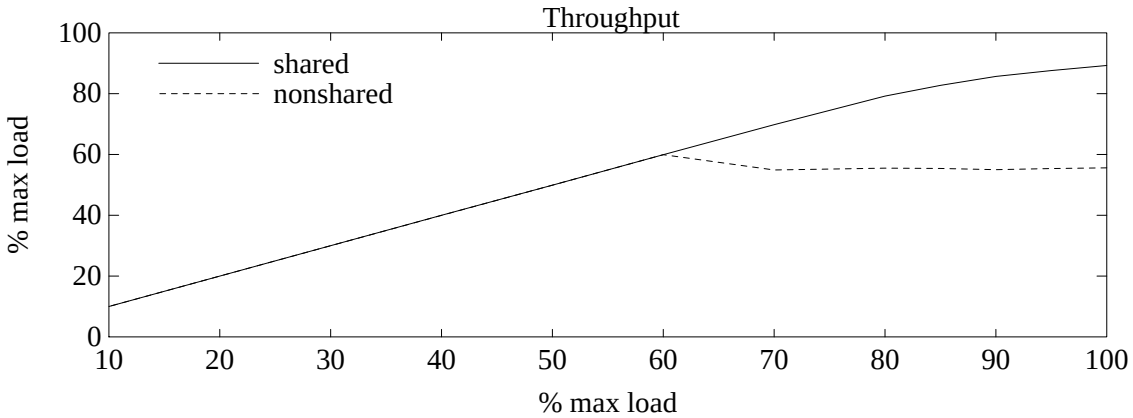

Chaos Routing

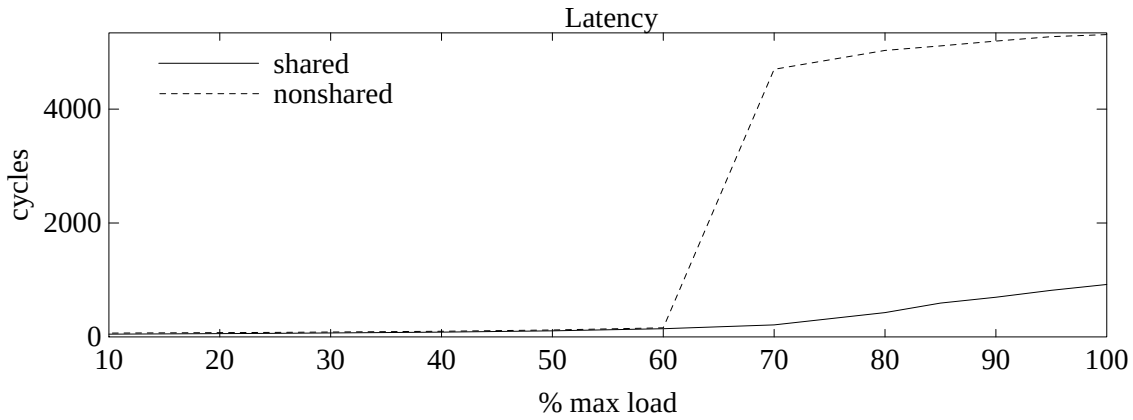
| % load | 10 | 25 | 40 | 50 | 60 | 70 | 80 | 85 | 90 | 95 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| mean xpt | 10.00 | 25.00 | 39.90 | 49.80 | 59.53 | 69.17 | 80.00 | 85.07 | 62.73 | 62.53 | 62.40 |
| std xpt | 0.00 | 0.00 | 0.00 | 0.00 | 0.05 | 0.05 | 0.00 | 0.17 | 0.21 | 0.41 | 0.28 |
| mean lat | 62.60 | 76.58 | 99.61 | 125.0 | 161.1 | 224.3 | 356.5 | 516.4 | 2580.8 | 2574.9 | 2565.7 |
| std lat | 0.00 | 0.06 | 0.13 | 0.47 | 1.03 | 1.69 | 2.97 | 13.25 | 8.52 | 22.32 | 22.11 |

Deflection Routing

| % load | 10 | 25 | 40 | 50 | 60 | 70 | 80 | 85 | 90 | 95 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| mean xpt | 10.00 | 24.80 | 40.03 | 50.00 | 59.93 | 69.63 | 71.67 | 71.67 | 71.57 | 71.90 | 71.83 |
| std xpt | 0.00 | 0.14 | 0.05 | 0.00 | 0.05 | 0.12 | 0.09 | 0.09 | 0.05 | 0.16 | 0.17 |
| mean lat | 704.6 | 717.8 | 742.8 | 772.8 | 821.8 | 956.1 | 1086.0 | 1097.5 | 1104.3 | 1106.6 | 1110.1 |
| std lat | 0.15 | 1.38 | 0.50 | 1.26 | 0.49 | 5.61 | 1.65 | 1.21 | 0.67 | 1.27 | 0.77 |

Figure B.8: 1024-node nonshared channel torus: chaos vs deflection (Uniform traffic)



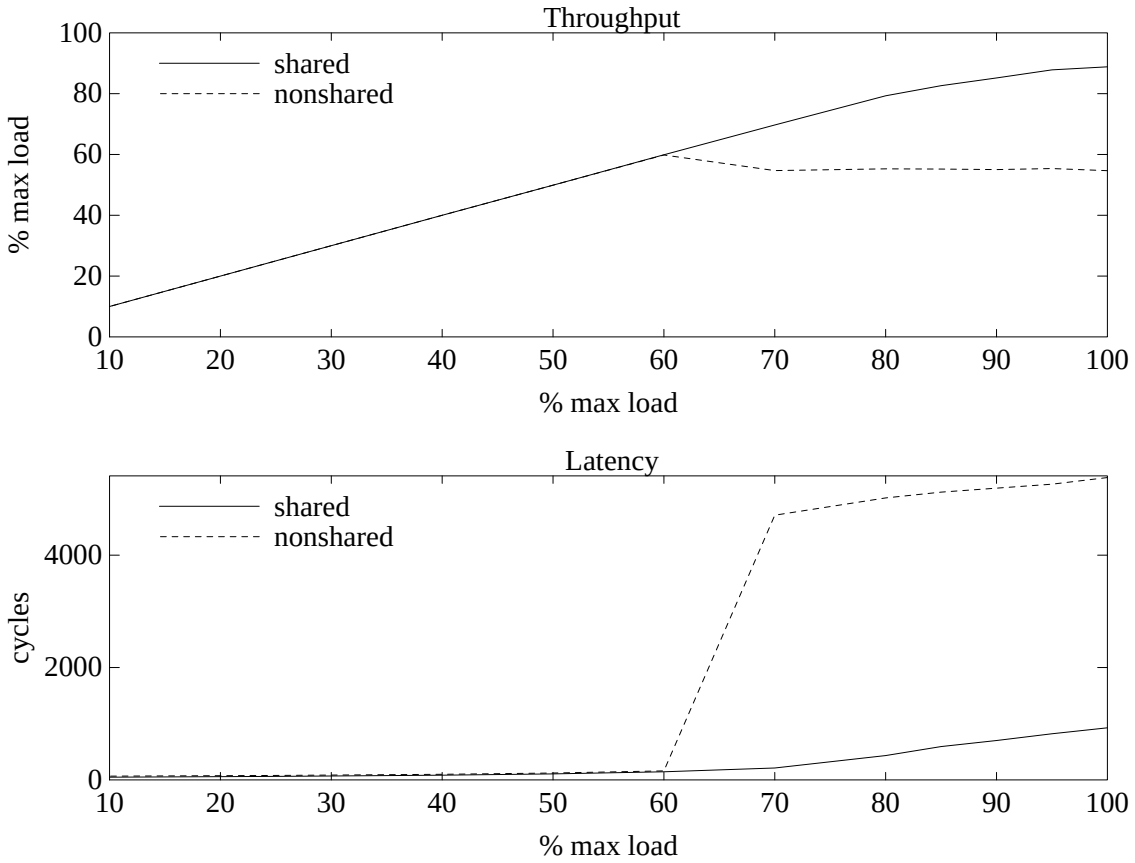

Shared channels

| % load | 10 | 25 | 40 | 50 | 60 | 70 | 80 | 85 | 90 | 95 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| mean xpt | 10.00 | 25.00 | 40.00 | 49.90 | 59.93 | 69.80 | 79.20 | 82.73 | 85.67 | 87.60 | 89.27 |
| std xpt | 0.00 | 0.00 | 0.00 | 0.00 | 0.05 | 0.14 | 0.08 | 0.09 | 0.33 | 0.16 | 0.29 |
| mean lat | 48.39 | 62.44 | 83.58 | 106.5 | 143.9 | 211.0 | 426.9 | 593.9 | 696.5 | 818.0 | 920.7 |
| std lat | 0.04 | 0.04 | 0.21 | 0.89 | 1.03 | 2.25 | 15.36 | 0.73 | 3.27 | 1.46 | 16.89 |

Nonshared channel

| % load | 25 | 10 | 40 | 50 | 60 | 70 | 80 | 85 | 90 | 95 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| mean xpt | 15.00 | 20.00 | 39.97 | 49.90 | 59.93 | 54.90 | 55.50 | 55.40 | 55.00 | 55.37 | 55.60 |
| std xpt | 7.07 | 7.07 | 0.05 | 0.00 | 0.05 | 0.37 | 0.51 | 0.29 | 0.24 | 0.54 | 0.29 |
| mean lat | 71.07 | 75.06 | 98.88 | 121.4 | 159.6 | 4699.2 | 5030.9 | 5110.2 | 5194.9 | 5272.7 | 5309.0 |
| std lat | 5.77 | 5.68 | 0.16 | 0.59 | 1.35 | 80.72 | 59.74 | 47.73 | 77.51 | 78.29 | 152.25 |

Figure B.9: 1024-node chaos mesh: shared vs nonshared (Uniform traffic)



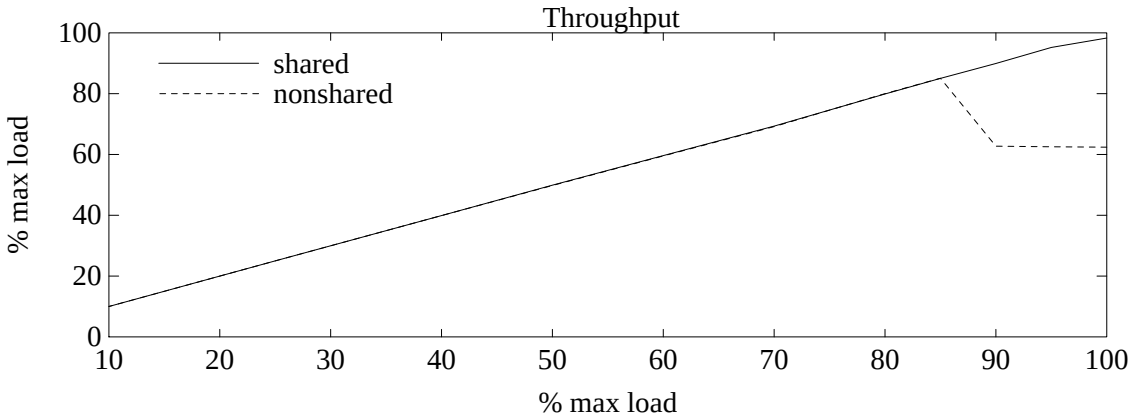

Shared channel

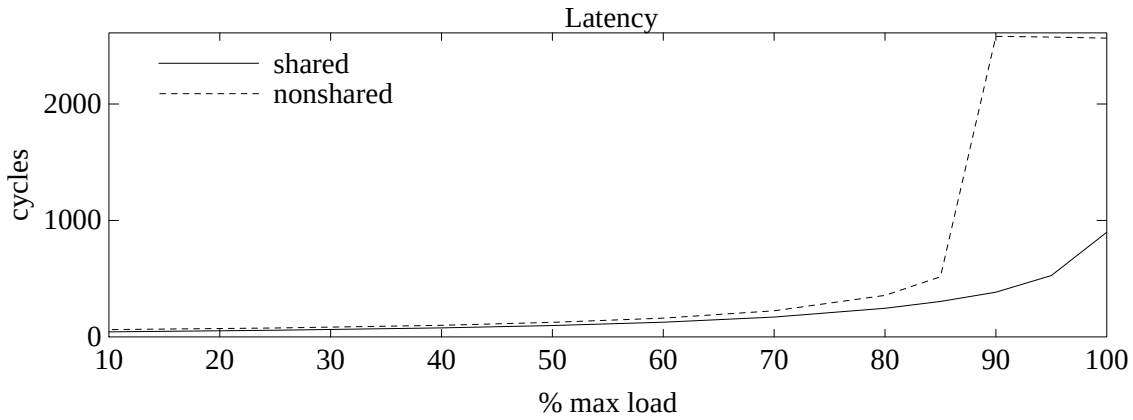
| % load | 10 | 25 | 40 | 50 | 60 | 70 | 80 | 85 | 90 | 95 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| mean xpt | 10.00 | 25.00 | 40.00 | 49.90 | 59.90 | 69.70 | 79.30 | 82.63 | 85.17 | 87.83 | 88.83 |
| std xpt | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.16 | 0.19 | 0.33 | 0.25 | 0.34 |
| mean lat | 48.46 | 62.68 | 83.90 | 106.97 | 144.48 | 212.02 | 433.12 | 593.56 | 701.25 | 821.30 | 927.16 |
| std lat | 0.07 | 0.04 | 0.17 | 0.65 | 0.45 | 3.85 | 14.46 | 5.52 | 5.35 | 6.58 | 11.48 |

Nonshared channel

| % load | 10 | 25 | 40 | 50 | 60 | 70 | 80 | 85 | 90 | 95 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| mean xpt | 10.00 | 25.00 | 40.00 | 49.90 | 59.83 | 54.70 | 55.27 | 55.20 | 55.03 | 55.37 | 54.67 |
| std xpt | 0.00 | 0.00 | 0.00 | 0.00 | 0.05 | 0.29 | 0.34 | 0.37 | 0.34 | 0.38 | 0.31 |
| mean lat | 67.03 | 79.26 | 99.65 | 122.4 | 160.5 | 4711.2 | 5018.7 | 5121.8 | 5192.7 | 5263.4 | 5381.2 |
| std lat | 0.04 | 0.15 | 0.25 | 0.68 | 0.76 | 22.41 | 49.61 | 49.24 | 68.30 | 19.29 | 62.97 |

Figure B.10: 1024-node chaos mesh:shared vs nonshared (4X hot spot traffic)



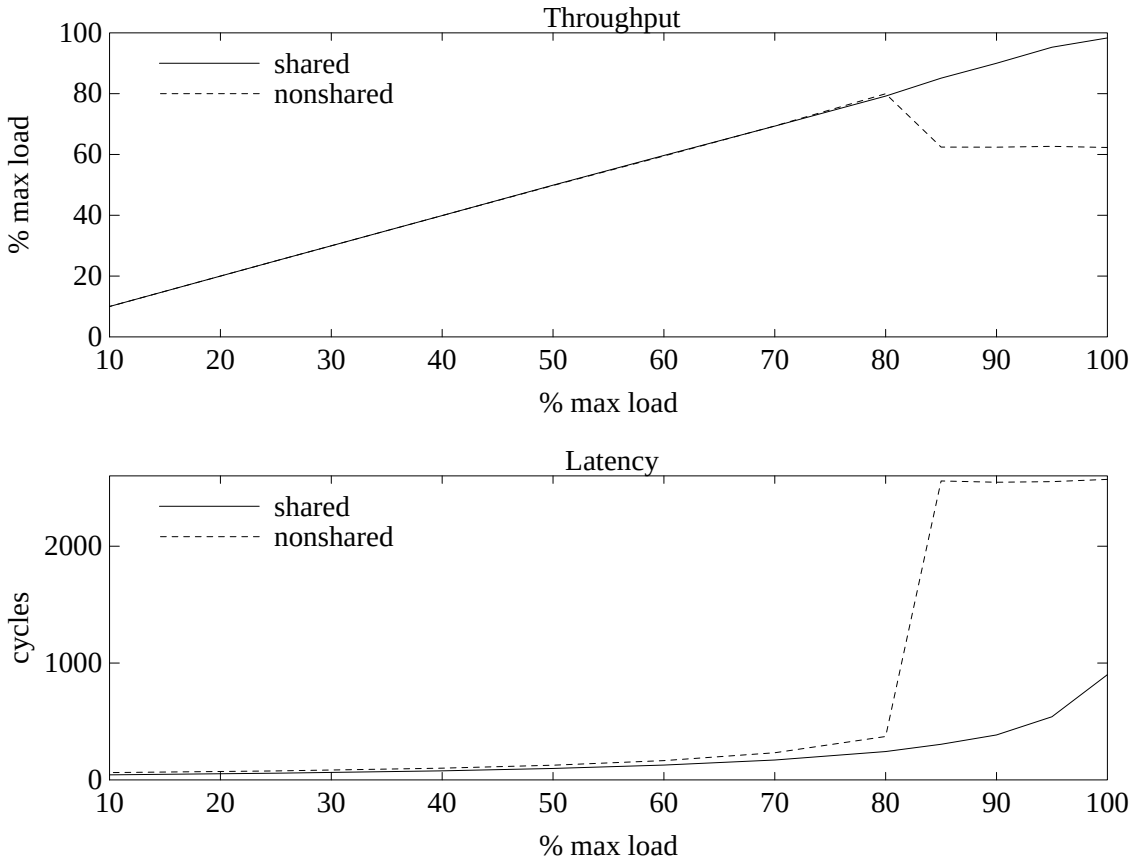

Shared channel

| % load | 10 | 25 | 40 | 50 | 60 | 70 | 80 | 85 | 90 | 95 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| mean xpt | 10.00 | 25.00 | 39.90 | 49.90 | 59.63 | 69.33 | 79.90 | 85.03 | 89.93 | 95.20 | 98.30 |
| std xpt | 0.00 | 0.00 | 0.00 | 0.00 | 0.05 | 0.05 | 0.08 | 0.05 | 0.12 | 0.29 | 0.50 |
| mean lat | 43.23 | 57.36 | 77.50 | 97.64 | 126.1 | 170.3 | 245.7 | 304.3 | 384.0 | 526.8 | 898.3 |
| std lat | 0.01 | 0.10 | 0.16 | 0.43 | 1.27 | 0.74 | 0.44 | 1.37 | 3.34 | 13.05 | 39.15 |

Nonshared channel

| % load | 10 | 25 | 40 | 50 | 60 | 70 | 80 | 85 | 90 | 95 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| mean xpt | 10.00 | 25.00 | 39.90 | 49.80 | 59.53 | 69.17 | 80.00 | 85.07 | 62.73 | 62.53 | 62.40 |
| std xpt | 0.00 | 0.00 | 0.00 | 0.00 | 0.05 | 0.05 | 0.00 | 0.17 | 0.21 | 0.41 | 0.28 |
| mean lat | 62.60 | 76.58 | 99.61 | 125.0 | 161.1 | 224.3 | 356.5 | 516.4 | 2580.8 | 2574.9 | 2565.7 |
| std lat | 0.00 | 0.06 | 0.13 | 0.47 | 1.03 | 1.69 | 2.97 | 13.25 | 8.52 | 22.32 | 22.11 |

Figure B.11: 1024-node chaos torus: shared vs nonshared (Uniform traffic)




Shared channel

| % load | 10 | 25 | 40 | 50 | 60 | 70 | 80 | 85 | 90 | 95 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| mean xpt | 10.00 | 25.00 | 39.90 | 49.90 | 59.67 | 69.33 | 79.20 | 85.10 | 90.00 | 95.27 | 98.33 |
| std xpt | 0.00 | 0.00 | 0.00 | 0.00 | 0.05 | 0.05 | 0.36 | 0.08 | 0.08 | 0.21 | 0.31 |
| mean lat | 43.21 | 57.35 | 77.55 | 98.07 | 127.03 | 170.21 | 242.79 | 304.91 | 384.79 | 540.51 | 900.73 |
| std lat | 0.04 | 0.11 | 0.29 | 0.36 | 0.88 | 0.44 | 1.22 | 1.99 | 1.30 | 11.49 | 29.34 |

Nonshared channel

| % load | 10 | 25 | 40 | 50 | 60 | 70 | 80 | 85 | 90 | 95 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| mean xpt | 10.00 | 25.00 | 39.90 | 49.80 | 59.50 | 69.37 | 79.97 | 62.43 | 62.40 | 62.67 | 62.30 |
| std xpt | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.31 | 0.05 | 0.12 | 0.33 | 0.21 | 0.14 |
| mean lat | 62.55 | 76.86 | 100.4 | 125.9 | 165.1 | 232.4 | 371.4 | 2558.5 | 2547.4 | 2552.7 | 2572.0 |
| std lat | 0.05 | 0.02 | 0.07 | 0.31 | 1.71 | 1.92 | 3.46 | 23.21 | 16.55 | 12.21 | 29.43 |

Figure B.12: 1024-node chaos torus: shared vs nonshared (4X hot spot traffic)

# Vita

Kevin Bolding graduated from Friendswood High School, Friendswood, Texas in June of 1984. He attended Rice University in Houston, Texas starting in 1984 and graduating *magna cum laude* with a Bachelor of Arts degree in Computer Science and Mathematical Science in May of 1988. In September 1988, he joined the Computer Science and Engineering Department at the University of Washington. There, he worked as a Research Assistant through July 1993 and held an Intel Foundation Graduate Fellowship from September 1990 through September 1991. He received a Master of Science degree in Computer Science and Engineering and was advanced to Ph.D. candidacy in June 1991.