

# TECHNIQUES FOR HIGH BANDWIDTH, LOW LATENCY INTERCONNECTION NETWORK OPERATION AT HIGH OFFERED LOADS


by

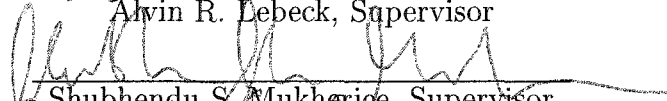
MITHUNA S. THOTTETHODI

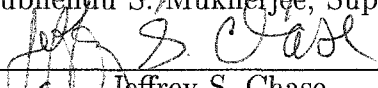
Department of Computer Science  
Duke University

Date: 10-23-02

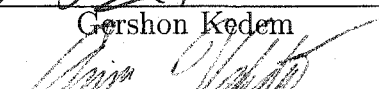
Approved:

  
Alvin R. Debeck, Supervisor

  
Shubhendu S. Mukherjee, Supervisor

  
Jeffrey S. Chase

  
Gershon Kedem

  
Amin M. Vahdat

Dissertation submitted in partial fulfillment of the  
requirements for the degree of Doctor of Philosophy  
in the Department of Computer Science  
in the Graduate School of  
Duke University

2002

UMI Number: 3092889

UMI<sup>®</sup>

---

UMI Microform 3092889

Copyright 2003 by ProQuest Information and Learning Company.

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

ProQuest Information and Learning Company  
300 North Zeeb Road  
P.O. Box 1346  
Ann Arbor, MI 48106-1346

# ABSTRACT

(Computer Architecture)

## TECHNIQUES FOR HIGH BANDWIDTH, LOW LATENCY INTERCONNECTION NETWORK OPERATION AT HIGH OFFERED LOADS


by


MITHUNA S. THOTTETHODI


Department of Computer Science  
Duke University

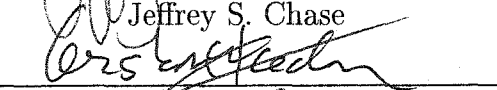
Date: 10-23-02


Approved:

  
Alvin R. Lebeck, Supervisor

  
Shubhendu S. Mukherjee, Supervisor

  
Jeffrey S. Chase

  
Gershon Kedem

  
Amin M. Vahdat

An abstract of a dissertation submitted in partial fulfillment of the  
requirements for the degree of Doctor of Philosophy  
in the Department of Computer Science  
in the Graduate School of  
Duke University

2002

## Abstract

Large-scale, cache coherent, distributed shared memory multiprocessors constitute an important and growing segment of the server market. Overall performance in such systems depends on compute performance, memory performance, and interconnection network performance. Interconnection network performance lies on the critical path of remote misses (which is the key communication mechanism in distributed shared memory multiprocessors) and consequently, poor network performance can significantly reduce overall system performance.

While there are a number of techniques that have enhanced interconnection network performance significantly, known performance problems due to network saturation at high offered loads that result in poor bandwidth and high latencies continue to pose a challenge to system designers. The symptoms of the problems are as follows: the achieved throughput goes up with offered load steadily till a certain point beyond which there is a sudden and significant drop in throughput (and a corresponding increase in latency). This occurs due to imbalances in network resource usage as well as the lack of fast feedback resulting in routers making decisions using only locally available information that may be globally detrimental.

Solutions to this problem can be broadly classified into two categories: (a) congestion control solutions and (b) load balancing solutions. This thesis proposes and evaluates one congestion control technique and two load balancing techniques to overcome the performance problems in k-ary, n-cube networks at high loads.

The first technique is a self-tuned congestion control mechanism—*Tune*—that has two key features: (a) it uses global congestion information to throttle the offered load when saturation is imminent and thus ensures that the network stays in the high-bandwidth, low latency region of operation and (b) it has a global throughput-driven



self-tuning mechanism that enables it to adapt to various communication patterns. Simulations with various communication patterns show that *Tune* is able to prevent the sudden drop in performance that occurs at saturation. *Tune* can be used with both wormhole and virtual cut-through switched networks.

The second technique—*congestion-aware via routing*—attempts to achieve load balancing to prevent saturation. In this technique, packets are directed away from congested network regions using global congestion information by requiring them to go via certain intermediate nodes. I demonstrate that this technique achieves limited improvements for non-uniform traffic patterns. Simulations show that the benefits of *congestion aware via-routing*, even with perfect global knowledge, are not compelling. These results illustrate the inherent limits of load balancing in the minimum rectangle and this key insight leads to the next technique.

Finally, I propose a new non-minimal routing (i.e. packets can be routed on hops that take them farther from the destination) algorithm—*BLAM*—that goes beyond the constraints of minimal routing which I assumed for the first two techniques. Non-minimal routing can deliver better performance because of the additional routing flexibility. However, because packets can go farther from the destination, livelock is a concern in non-minimal routing. Existing non-minimal routing algorithms either require costly and non-scalable implementations of router-wide priorities or they offer only probabilistic guarantees of livelock-freedom. In contrast, *BLAM* achieves the higher performance of non-minimal routing by using *lazy misroutes* without the drawbacks, i.e., it offers deterministic guarantees of livelock freedom by using *limited misroutes*. Simulations show that *BLAM* achieves performance similar to *chaotic routing*, which is a high-performance non-minimal routing algorithm with only probabilistic guarantees of livelock freedom.

# Contents

<b>Abstract</b>	<b>iii</b>
<b>List of Tables</b>	<b>viii</b>
<b>List of Figures</b>	<b>ix</b>
<b>Acknowledgements</b>	<b>xiii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Contributions . . . . .	4
1.2 Thesis Organization . . . . .	8
<b>2 Background and Related Work</b>	<b>9</b>
2.1 Background . . . . .	9
2.1.1 Network Topology . . . . .	10
2.1.2 Base Router Architecture . . . . .	10
2.1.3 Deadlock Handling in Minimal Adaptive Routers with Escape Paths . . . . .	12
2.2 Related Work . . . . .	13
2.2.1 Congestion Control . . . . .	13
2.2.2 Load Balancing in the Minimum Rectangle . . . . .	16
2.2.3 Deadlock, Livelock and Performance Issues in Nonminimal Adaptive Routers . . . . .	18
<b>3 Tune: A Self-tuned Congestion Control Mechanism</b>	<b>21</b>
3.1 Gathering global information . . . . .	22
3.1.1 Implementing Global Information Gather . . . . .	23

3.2	Self-Tuning Mechanism . . . . .	27
3.2.1	Hill Climbing . . . . .	28
3.2.2	Avoiding Local Maxima . . . . .	30
3.2.3	Summary . . . . .	31
3.3	Methodology . . . . .	32
3.4	Simulation Results . . . . .	34
3.4.1	Overall Performance . . . . .	35
3.4.2	Obtaining Global Information . . . . .	41
3.4.3	Varying <i>Tune</i> 's increment/decrement values . . . . .	44
3.4.4	Bursty Traffic . . . . .	44
3.4.5	Distributing Information using Meta Packets . . . . .	50
3.5	Summary . . . . .	54
<b>4</b>	<b>Load Balancing in the Minimum Rectangle</b>	<b>56</b>
4.1	Via Routing: The Mechanism . . . . .	57
4.2	Min-triangle and Random-triangle Via Selection Policies . . . . .	59
4.3	Min-Corridor Via Selection Policy . . . . .	60
4.4	Via-Routing Results . . . . .	62
4.5	Summary . . . . .	64
<b>5</b>	<b>BLAM routing</b>	<b>74</b>
5.1	Deadlocks, Livelocks and Performance in Nonminimal Adaptive Routing	74
5.2	Misroutes . . . . .	77
5.3	Bypass buffers . . . . .	79
5.3.1	Implementing Distributed Bypass Buffers . . . . .	81

5.3.2	Summary . . . . .	84
5.4	Evaluation Methodology . . . . .	85
5.4.1	Simulation Details . . . . .	85
5.4.2	Open vs. Closed Loop . . . . .	86
5.4.3	Network Workload . . . . .	87
5.4.4	Network and Router Architectures . . . . .	88
5.5	Simulation Results . . . . .	89
5.5.1	Overall Performance . . . . .	90
5.5.2	Varying the Misroute Limit . . . . .	98
5.5.3	Effect of Adding Bypass Buffers . . . . .	104
5.5.4	<i>M</i> -misroute, Adaptive router . . . . .	104
5.5.5	Varying Packet Size and Network Size . . . . .	112
5.6	Summary . . . . .	113
<b>6</b>	<b>Conclusion</b>	<b>123</b>
6.1	Summary . . . . .	123
	<b>Bibliography</b>	<b>126</b>
	<b>Biography</b>	<b>133</b>

## List of Tables

3.1	Tuning decision table . . . . .	29
5.1	Design variables for various routing schemes . . . . .	85

## List of Figures

1.1	Distributed Shared Memory Systems with k-ary, n-cube networks . . .	2
1.2	Interconnection network on Critical Path . . . . .	3
1.3	Performance Breakdown at Network Saturation, 16x16 2D network, adaptive routing, deadlock recovery . . . . .	4
1.4	Minimal Adaptive Routing w/deadlock recovery and Chaotic Routing	5
1.5	Routing Options Minimal vs. Non-minimal . . . . .	7
2.1	Base Minimal, Adaptive Router with Multiple Virtual Channels . . .	11
3.1	Dimension-wise global aggregation. . . . .	25
3.2	Estimation of global congestion : Previous Snapshot vs. Linear Ex- trapolation. . . . .	26
3.3	Throughput vs. Full Buffers . . . . .	28
3.4	Overall Performance With Random Traffic . . . . .	37
3.5	Overall Performance With Bit-Reversal Traffic Pattern . . . . .	38
3.6	Overall Performance With Perfect-Shuffle Traffic Pattern . . . . .	39
3.7	Overall Performance With Complement Traffic Pattern . . . . .	40
3.8	Effect of Global Information Gathering Delays for Deadlock Recovery (a & b) and Deadlock Avoidance (c & d). . . . .	46
3.9	Static Threshold vs. Tuning. . . . .	47
3.10	Self-Tuning Operation : An Example . . . . .	47
3.11	Choice of increment/decrement quanta . . . . .	48

3.12	Offered bursty load . . . . .	49
3.13	Performance with Bursty Load : Delivered Throughput . . . . .	49
3.14	Row aggregation using two meta packets . . . . .	51
3.15	Minimum, Maximum and Average gather times using meta packets . . . . .	52
3.16	Effect of Meta-packets and Tuning on Throughput . . . . .	53
4.1	Via Routing : Upper and Lower Triangles . . . . .	60
4.2	Representation of full input buffers . . . . .	62
4.3	Recasting $LCP_{min}$ as SP problem in directed graph . . . . .	62
4.4	Via-routing for Uniform Random pattern . . . . .	66
4.5	Via-routing for Bit Reversal pattern . . . . .	67
4.6	Via-routing for Complement pattern . . . . .	68
4.7	Via-routing for Perfect Shuffle pattern . . . . .	69
4.8	Via-routing with <i>TUNE</i> for Uniform Random pattern . . . . .	70
4.9	Via-routing with <i>TUNE</i> for Bit Reversal pattern . . . . .	71
4.10	Via-routing with <i>TUNE</i> for Complement pattern . . . . .	72
4.11	Via-routing with <i>TUNE</i> for Perfect Shuffle pattern . . . . .	73
5.1	Minimal Adaptive Routing w/deadlock recovery and Chaotic Routing . . . . .	75
5.2	Routing Options Minimal vs. Nonminimal . . . . .	76
5.3	Central Bypass Buffers: Chaos . . . . .	82
5.4	Distributed Bypass Buffers . . . . .	84

5.5	Choice of Base configuration . . . . .	88
5.6	Overall Performance With Random Traffic . . . . .	94
5.7	Overall Performance With Bit-Reversal Traffic Pattern . . . . .	95
5.8	Overall Performance With Perfect-Shuffle Traffic Pattern . . . . .	96
5.9	Overall Performance With Complement Traffic Pattern . . . . .	97
5.10	Effects of Varying the Misroute Limit of BLAM, Uniform Random Traffic	100
5.11	Effects of Varying the Misroute Limit of BLAM, Perfect Shuffle Traffic	101
5.12	Effects of Varying the Misroute Limit of BLAM, Complement Traffic	102
5.13	Effects of Varying the Misroute Limit of BLAM, Bit Reversal Traffic .	103
5.14	<i>M</i> -misroute, Adaptive Router, Uniform Random Traffic . . . . .	106
5.15	<i>M</i> -misroute, Adaptive Router, Perfect Shuffle Traffic . . . . .	107
5.16	<i>M</i> -misroute, Adaptive Router, Complement Traffic . . . . .	108
5.17	<i>M</i> -misroute, Adaptive Router, Bit Reversal Traffic . . . . .	109
5.18	<i>M</i> -misroute, Adaptive Router with <i>lazy</i> Misroutes . . . . .	110
5.19	<i>M</i> -misroute, Adaptive Router with <i>lazy</i> Misroutes . . . . .	111
5.20	Overall Performance For Other Packet Sizes (Uniform Random Traffic, 16-ary, 2-cube Network) . . . . .	115
5.21	Overall Performance For Other Packet Sizes (Bit-Reversal Traffic, 16- ary, 2-cube Network) . . . . .	116
5.22	Overall Performance For Other Packet Sizes (Complement Traffic, 16- ary, 2-cube Network) . . . . .	117
5.23	Overall Performance For Other Packet Sizes (Perfect Shuffle Traffic, 16-ary, 2-cube Network) . . . . .	118



5.24 Overall Performance For Other Network Sizes (Uniform Random Traffic, 16 flit packets) . . . . .	119
5.25 Overall Performance For Other Network Sizes (Bit-Reversal Traffic, 16 flit packets) . . . . .	120
5.26 Overall Performance For Other Network Sizes (Complement Traffic, 16 flit packets) . . . . .	121
5.27 Overall Performance For Other Network Sizes (Perfect Shuffle Traffic, 16 flit packets) . . . . .	122

## Acknowledgements

I consider myself lucky to have had Seema by my side these past four years, and I don't think she will ever know how much her love and support has meant to me. I am also extremely thankful to my parents for their love, encouragement and support.

I wish to thank my advisors, Professor Alvin Lebeck and Shubhendu Mukherjee, whose guidance has been extremely valuable to me. The knowledge and experience I have gained under their guidance, both in the methods of research and in specific research insights will help me throughout my career. I would also like to thank members of my PhD examination committee for their suggestions and comments that helped polish my research and thesis.

Working with Professor Siddhartha Chatterjee, Srikanth Srinivasan, Chia-lin Yang and Dave Raymond on my initial research projects as a graduate student was a rewarding and enriching experience. I have also benefited greatly from the many teachers who have taught me at Duke University and IIT Kharagpur.

Finally, it would have been considerably more difficult for me to adjust to my stay far away from India but for the company of all my family and friends. Thank you, Varsha, Babu, Sri, Gopal, Chia-lin, Karthikeyan, Rama, Om, Krishna, Amit, Pavan, Padmini, Kasturirangan, Vamsee, Rajesh, Jaidev, Sphoorthy, Vikram and Rachana for everything.

# Chapter 1:

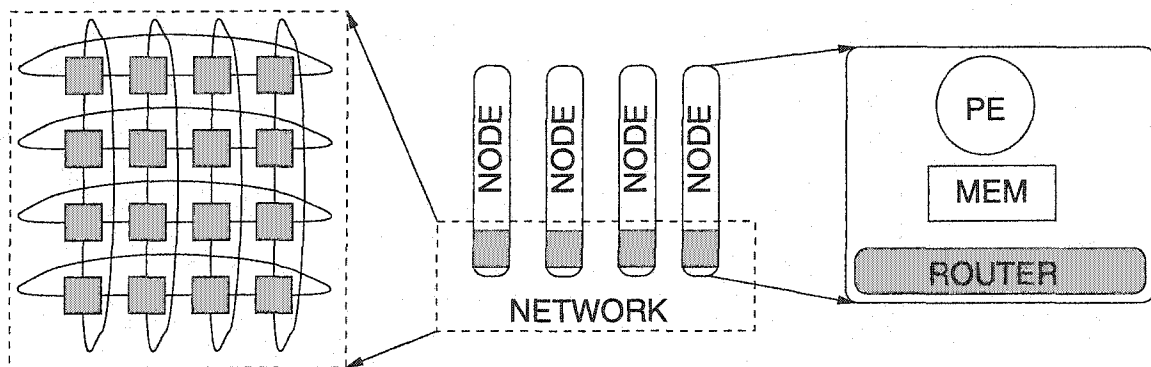
## Introduction

The market for large-scale cache-coherent distributed shared-memory multiprocessor machines (DSMPs) with 16 or more processors has tripled in the past four years [9]. In 2001 this server market resulted in an annual revenue of \$9 billion. Roughly half of this revenue resulted from large-scale machines with 32 or more processors. Today most major vendors, such as IBM [12, 22], HP [29, 43, 32], SGI [54], and Sun Microsystems [9], offer machines that scale up to a large number of processors (usually between 24 and 512).

These systems (Figure 1.1) typically consist of a set of *nodes* that can communicate over an *interconnection network*. As shown in Figure 1.1, each node consists of a processing element (PE), memory (including caches) and a router. The interconnection network connects all the routers and allows communication between various nodes.

The *memory wall* problem in uniprocessor design is useful to illustrate the criticality of interconnection network performance in DSMP performance. In the uniprocessor system context, the growing gap between processor and memory speeds poses a challenge. Memory hierarchies and processor techniques to “tolerate” cache-miss latencies mitigate this problem to an extent, but the gap is too wide to be completely tolerated. This results in processor stalls and reduces overall performance. Consequently, there is a need for high bandwidth and low latency miss servicing to ensure that the processor does not stall waiting for data.

The same problem of miss-service latency and bandwidth is exacerbated in DSMP systems because a cache miss may potentially suffer the interconnection network delay in addition to the memory latency (Figure 1.2). High latencies and/or poor



**Figure 1.1:** Distributed Shared Memory Systems with  $k$ -ary,  $n$ -cube networks

bandwidth can increase the time required to service remote cache misses and has the potential to increase processor stall time. As such, high bandwidth, low latency interconnection network performance is desirable in order to achieve high system performance in demanding server applications, such as databases [51]. Further, the advent of multiprocessor systems built with highly aggressive, out-of-order, and speculative microprocessors, simultaneous multithreaded processors [21], and chip multiprocessors [16], promises to dramatically increase the offered load on such multiprocessor networks.

Unfortunately, network packets are often delayed due to transient network congestion. Consequently, many interconnection networks employ virtual cut-through and adaptive routing algorithms. Virtual cut-through pipelines a packet among multiple routers and buffers it entirely at a router when the packet header is blocked due to congestion. This reduces congestion around a router by removing packets from the network links. Adaptive routing routes packets around congested spots in a network (by adapting to the network state) to achieve higher throughput from the network. Virtual channels also help reduce congestion by reducing head-of-line waiting. However, in spite of significant benefits from the use of techniques mentioned above, there remain known performance bottlenecks [48] at high loads due to network saturation.

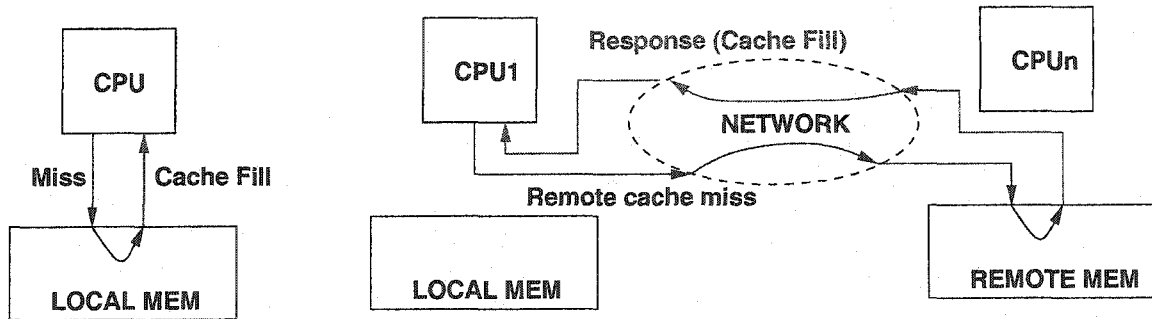


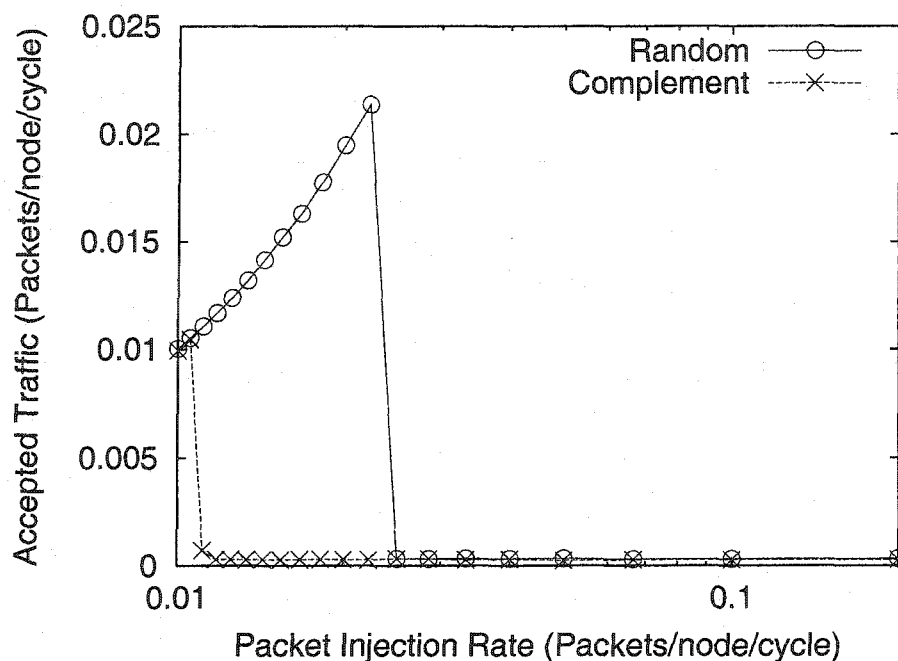
Figure 1.2: Interconnection network on Critical Path

Tree saturation occurs when multiple packets contend for a single resource (e.g., a link between nodes) creating a hot-spot. Since only one packet can use the resource, other packets must wait. These waiting packets occupy buffers and thus delay other packets, even though they may be destined for a completely different node and share only one link on their paths to their respective destinations. This process continues, waiting packets delay other packets producing a tree of waiting packets that fans out from the original hot-spot.

The performance degradation caused by network saturation can be severe, as illustrated in Figure 1.3. The y-axis corresponds to delivered bandwidth (flits/node/cycle) while the x-axis shows offered load in terms of packet injection rate (packets/node/cycle). The two lines correspond to different communication patterns: randomly selecting a destination node (*random*), and using the node number with its bits inverted as the destination (*complement*). (I explain communication patterns in greater detail in Section 5.4.3.)

From Figure 1.3 we can make two important observations. First, both communication patterns incur dramatic reductions in throughput when the network reaches saturation. The second observation is that the network saturates at different points for the different communication patterns.

The space of solutions to the problem of performance degradation at network

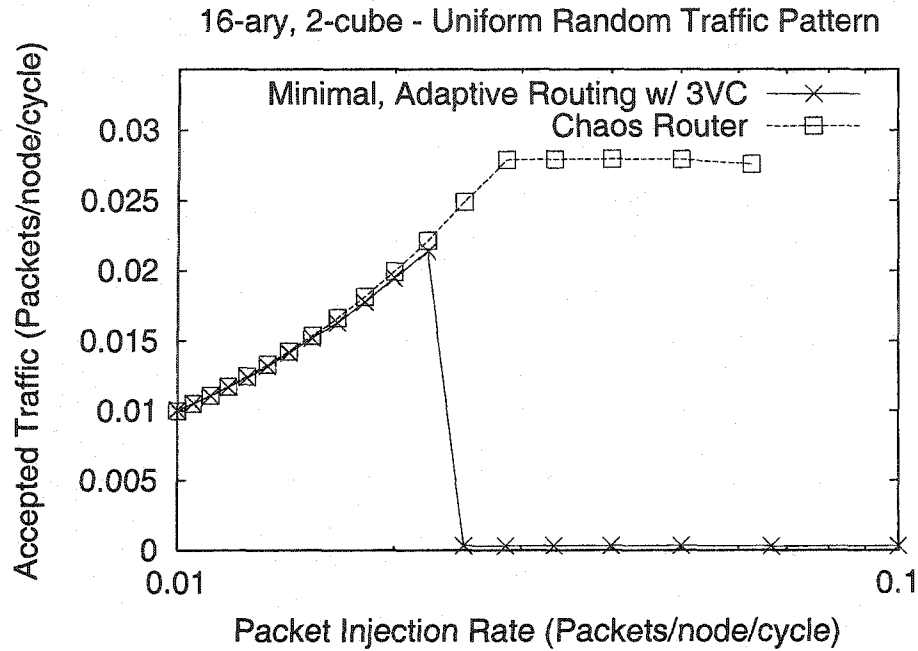


**Figure 1.3:** Performance Breakdown at Network Saturation, 16x16 2D network, adaptive routing, deadlock recovery

saturation can be divided into two broad sub-spaces: congestion control mechanisms and load-balancing schemes. The congestion control approach throttles injection of new packets into the network so as to keep the network operating in the high bandwidth, low latency region of operation. This region of operation corresponds to the peak in Figure 1.3 at loads slightly lower than saturation load. The load balancing approach tries to prevent imbalances in load in different regions of the network. The rationale behind this approach is that load imbalances can cause network delays in a small region of the network that can propagate quickly because of the tree-saturation phenomenon.

## 1.1 Contributions

The contributions of this thesis are the design and evaluation of the following three techniques to achieve high bandwidth, low latency interconnection network operation



**Figure 1.4:** Minimal Adaptive Routing w/deadlock recovery and Chaotic Routing at high loads. The first technique falls under the congestion control category and the second two techniques fall under the load balancing categories.

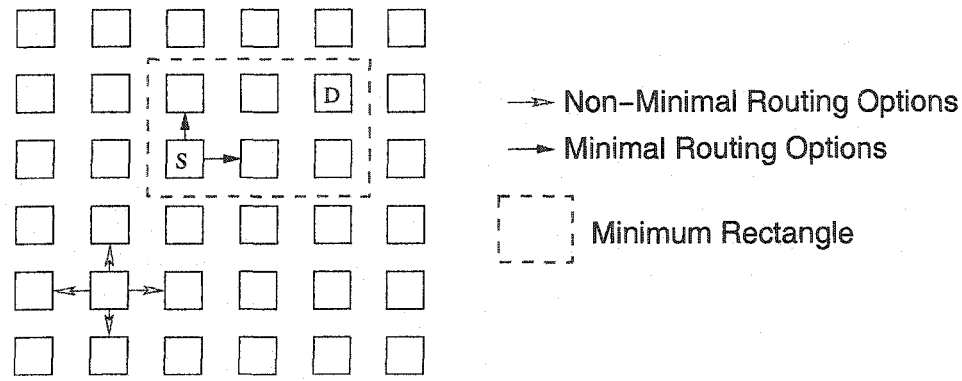
- *Self-tuned, global knowledge based, congestion control:* One way to prevent network saturation is to use source throttling, which prevents source node packet injection when congestion is detected. An oracle could achieve this by knowing both the communication pattern and the packet injection rate that maximizes performance. The challenge is to develop a realistic implementation that can prevent performance degradation at network saturation and adapt to variations in communication patterns.

I propose a self-tuned, global knowledge based congestion control mechanism that achieves both objectives. The two key innovations of this technique are: (a) faster feedback via the use of global congestion information and (b) a throughput driven self-tuned congestion control mechanism that adapts to changes in

communication patterns.

- *Congestion-aware via routing*: An alternate way would be achieve load balancing by making sure that packets do not flood congested regions of the network. This prevents congestion from building up in regions of the network and thus prevents saturation. I use a mechanism called via-routing that routes packets towards certain intermediate nodes (*via* nodes) en route to the destination node. The key innovation of my technique is the use of of global knowledge to direct packets to congestion free regions of the network by using global-congestion-aware via-node selection policies.
- *BLAM routing*: Earlier, we had seen the performance bottleneck of minimal adaptive routers, i.e., routers that ensure that packets get closer to the destination with each hop, at high loads due to network saturation. Interestingly, however, non-minimal routing algorithms, which allow packets to take hops that take them farther from the destination, may perform significantly better than a minimal adaptive routing algorithm (Figure 1.4) at high offered loads. This is because non-minimal routing algorithms offer greater routing freedom compared to a minimal adaptive routing algorithms (Figure 1.5.) Figure 1.4 compares the performance of a minimal adaptive router with the *Chaos* router, which is a non-minimal adaptive router. The minimal adaptive routing algorithm saturates and, thereby, causes the performance to degrade rapidly beyond a certain offered load. Chaos' performance is better on two counts: (a) saturation occurs at higher loads and (b) there is no degradation in performance at saturation. Unfortunately, the better performance of *Chaos* is achieved at the cost of weaker livelock-freedom guarantees. Chaotic routing offers no deterministic guarantee of livelock-freedom—only a probabilistic guarantee that





**Figure 1.5:** Routing Options Minimal vs. Non-minimal

the chance of a packet continuously circulating in the network diminishes with each hop.

Unfortunately, in spite of its high performance, to the best of my knowledge no commercially available interconnection network uses the Chaos routing algorithm, even though it has been over a decade since the design was proposed. The presence of livelocks—however low its probability may be—causes network designers to shy away from using such algorithms in real products. The challenge is to develop a solution that has the benefits of each of the two routing algorithms (minimal adaptive and Chaos) without either technique's pitfalls. I propose a new non-minimal routing algorithm—*BLAM*—that has the best features of minimal (deterministic guarantees of deadlock- and livelock-freedom) and non-minimal routing (better performance due to improved flexibility) without the drawbacks of either. The key innovations of this technique are the use of lazy misrouting (to prevent spurious and wasteful misrouting) with bypassing (to eliminate head-of-line waiting) and the use of limited misroutes (to ensure livelock-freedom).

## 1.2 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 provides background information and discusses related work. The next three chapters describe and evaluate three techniques to achieve high bandwidth low latency network operation in interconnection networks. Chapter 3 describes a congestion control mechanism that relies on global information for faster feedback and self-tuning for adapting to various communication patterns. Chapter 4 proposes a mechanism for *congestion aware via-routing* that attempts load balancing to overcome the performance problems at high loads. Chapter 5 describes a new high performance non-minimal routing algorithm—*BLAM*—that is guaranteed to be livelock- and deadlock-free. Chapter 6 summarizes and concludes this thesis.

## Chapter 2: Background and Related Work

### 2.1 Background

High performance interconnection networks in tightly coupled multiprocessors can be achieved by using wormhole [14, 15] or virtual cut-through switching [35], adaptive routing [28], and multiple virtual channels [13]. Many commercial machines [43, 53, 39] use a combination of these techniques for their interconnection networks. In these systems communication occurs by sending packets of information that are routed independently through the network. Each packet is composed of flits (flow control units) that are transferred between network nodes.<sup>1</sup> Usually, the header flit(s) contains information necessary to choose a route to the destination.

In wormhole switching, when a node receives the header flit (which typically contains the routing information), it immediately selects a route and forwards the flit to the next node. This can provide very low latency compared to store-and-forward routing where the entire packet is received by a node before forwarding it. However, when a packet blocks in a wormhole network, its flits occupy buffer space across several network nodes. In contrast, virtual cut-through routers can buffer the entire blocked packet within a single node while also providing low latency at light loads, since they forward packet headers without waiting for the whole packet to arrive.

In the rest of this section, I describe the network topology and the router architecture I consider in this thesis.

---

<sup>1</sup>For ease of exposition I assume each network node contains a processor, memory and a network router.

### 2.1.1 Network Topology

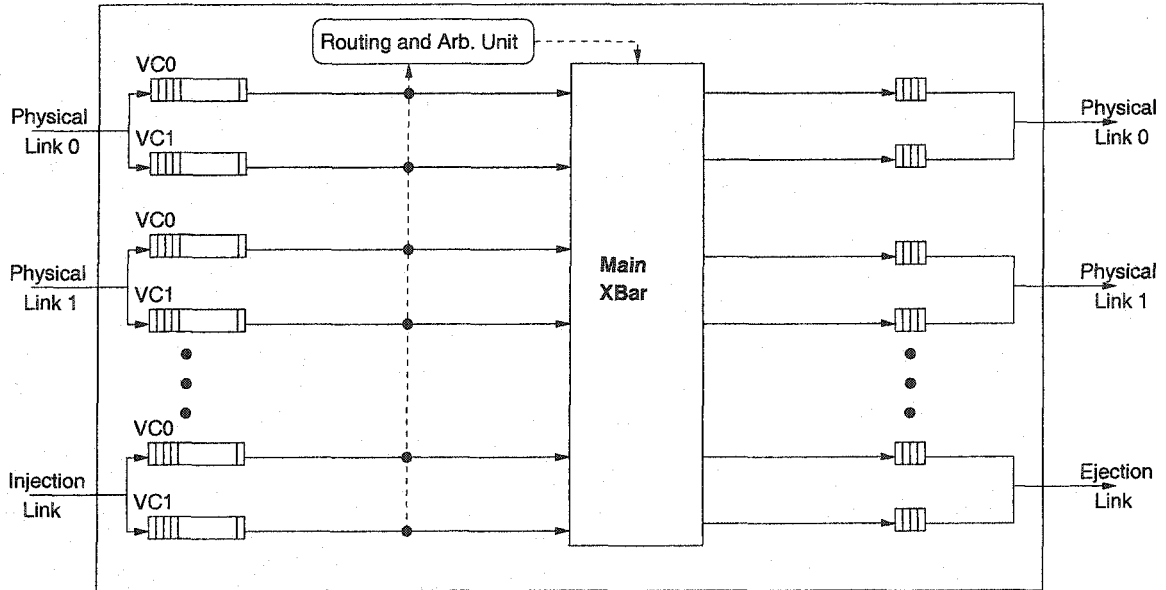
In this thesis, I consider *direct* networks of the k-ary, n-cube topology. Direct networks are networks where each node can produce/consume network packets (i.e. each node has a processor and memory). In contrast, *indirect* networks may have intermediate switches which relay other packets but do not produce/consume any traffic.

One attractive property of indirect networks is that the node containing the processor can use all the available pin bandwidth exclusively for its communication needs. In an integrated direct network, part of the node's pin bandwidth may be used to relay packets of two other nodes in the system. However, this thesis does not consider indirect networks (such as fat-trees, multistage interconnection networks) for two reasons: (1) The trend towards higher integration and system-on-a-chip is leading towards direct networks since the processor, caches and router can all be integrated on one die, (2) Indirect networks can be modeled with direct networks where certain hosts do not produce/consume any traffic, essentially acting as intermediate switches.

Of the various possible topologies among direct networks, this thesis considers the k-ary, n-cube topology (which is an n-dimensional matrix with k nodes along each dimension and wrap-around edges) because it has many desirable properties. k-ary, n-cube networks have a large number of alternate routes between a source and a destination. k-ary, n-cube topologies are a broad family that include many popular topologies such as rings (k-ary, 1-cube), meshes (k-ary, 2-cube with some routing restrictions) and hypercubes (2-ary, n-cube).

### 2.1.2 Base Router Architecture

Above, I described the network architecture. Here I discuss the architecture of each router in the network. Figure 2.1 shows the *minimal, adaptive* router with *multiple*



**Figure 2.1:** Base Minimal, Adaptive Router with Multiple Virtual Channels

*virtual channels* that I use as the base case router in the rest of this thesis. The next few paragraphs describe each of these characteristics of my base router in greater detail.

Each node has several incoming and outgoing network physical links and injection/ejection channels through which packets enter/exit the network. The physical channels are logically split into multiple virtual channels, each with its own buffers. The routing and arbitration unit sets up the crossbar connections linking input buffers to output buffers. Flits deposited in the output buffer are transferred across the physical link into the corresponding input buffer at the neighboring node.

A packet hop is classified as *profitable* or *minimal* if it takes the packet closer to its destination. A hop that takes a packet farther from its destination is known as a *misroute* or a *non-minimal hop*. A *profitable channel* is a channel on which a packet can make a profitable hop. We can classify routers as minimal or non-minimal. A minimal router offers only profitable hops whereas a non-minimal router may offer misrouting hops as well.

Adaptive routing dynamically chooses from multiple potential routes based on current local network state. This offers more routing flexibility compared to a deterministic routing algorithm, and thus provides higher performance. Unfortunately, full adaptive routing can cause deadlock cycles. The next section describes ways to handle deadlock cycles that form due to adaptive routing.

### **2.1.3 Deadlock Handling in Minimal Adaptive Routers with Escape Paths**

Virtual channels allow multiple packets to share a single physical link and can be used to eliminate deadlocks. Deadlock avoidance schemes work by preventing the cyclic dependencies between storage resources. In particular, I consider a scheme that reserves a small set of virtual channels for deadlock-free routing [18], while the remaining virtual channels use fully adaptive routing. This technique guarantees forward progress, since packets routed over the special channels will never deadlock, and eventually free up resources for the fully adaptive channels. The Alpha 21364 router [43] is an example of a system with multiple virtual channels, adaptive routing and deadlock avoidance.

Deadlock recovery [38] is an alternative to deadlock avoidance that can potentially achieve higher performance. Deadlock recovery uses full adaptive routing on all virtual channels, detects when deadlocks occur (typically via timeouts), then recovers by routing packets on a deadlock-free path which uses a central per-node buffer. This is the primary difference from deadlock avoidance, which requires buffers per physical channel.

In either deadlock avoidance or recovery, the frequency of deadlocks in the adaptive channels increases dramatically when the network reaches saturation [38]. When this occurs, packets are delivered over the relatively limited escape bandwidth available on the deadlock-free paths. This causes a sudden, severe drop in throughput

and corresponding increase in packet latency.

## 2.2 Related Work

The contribution of this thesis is the design and evaluation of three techniques to overcome the problem of performance degradation at saturation and to sustain high-bandwidth, low latency network operation. In this section, I discuss previous research and its relation to the three techniques. Section 2.2.1 describes various research studies that used the congestion control techniques and compares it with my technique described in Chapter 3. Section 2.2.2 describes previous load-balancing research and its relation to my *congestion aware via-routing* technique described in Chapter 4. Finally, Section 2.2.3 describes the livelock, deadlock and performance issues that crop up when non-minimal routing is considered. This provides the background for Chapter 5 where I describe and evaluate a non-minimal routing algorithm—BLAM.

### 2.2.1 Congestion Control

Most previous work on congestion control for multiprocessor networks relies on estimating network congestion independently at each node and limiting packet injection when the network is predicted to be near saturation. This reduces the problem to finding a local heuristic at each node to estimate network congestion. Lopez et al. [40, 41] use the number of busy output virtual channels in a node to estimate congestion. Baydal et al. [4] propose an approach that counts a subset (free and useful) of virtual channel buffers to decide whether to throttle or not. Because the above schemes rely on local symptoms of congestion, they lack knowledge about the global network state, and are unable to take corrective action in a timely manner. This reduces their effectiveness under different network load levels and communication patterns.

Smai and Thorelli describe a form of global congestion control [56]. A node that detects congestion (based on time-outs) signals all the other nodes in the network to also limit packet injection. This approach requires tuning the appropriate time-outs, and when the timeouts are tuned for robustness at higher loads, there is a performance penalty for light loads. Scott and Sohi describe the use of explicit feedback to inform nodes when tree-saturation is imminent in multistage interconnection networks [52]. This approach also requires tuning of thresholds. The technique proposed by Kim et al. [36] allows the sender to kill any packet that has experienced more delays than a threshold. This approach pads shorter packets to ensure that the sender can kill a packet at any time before its first flit reaches the destination. This can cause larger overheads when short messages are sent to distant nodes.

The above techniques for congestion control in multiprocessor networks all attempt to prevent network saturation at heavy loads. Unfortunately, these techniques either require tuning, lack necessary information about a network's global state to take preventive actions in a timely fashion, or do not provide high performance under all traffic patterns and offered load levels.

Flit-reservation flow control is an alternative flow control technique which improves the network utilization at which saturation occurs [47]. It uses control flits to schedule bandwidth and buffers ahead of the arrival of data-flits. This pre-scheduling results in better re-use of buffers than waiting for feedback from neighboring nodes to free up buffers. Basak and Panda demonstrate that consumption channels can be a bottleneck that can exacerbate tree saturation. They show that saturation bandwidth can be increased by having an appropriate number of consumption channels [3].

LANs (Local Area Networks) and WANs (Wide Area Networks) use self-tuned congestion control techniques. Various flavors of self-tuning, end-to-end congestion avoidance and control techniques have been used in the TCP protocol [33, 6]. TCP's



congestion control mechanism uses time-outs and dropped/unacknowledged packets to locally estimate global congestion and throughput. If congestion is detected, the size of the sliding window, which controls the number of unacknowledged packets that can be in flight, is reduced. Floyd and Jacobson [26] proposed a scheme where TCP packets are dropped when a router feels that congestion is imminent. Dropped packets give an early indication to end hosts to take corrective action and scale back offered load. Floyd [25] also proposed a modification of TCP where “choke packets” are explicitly sent to other hosts. Ramakrishnan and Jain describe a similar mechanism for DECbit to explicitly notify congestion whereby gateways set the ECN (*Explicit Congestion Notification*) bit depending on average queue size [50].

Congestion control in ATM [34] uses explicit packets called *Resource Management (RM) cells* to propagate congestion information. Switches along the packet path modify bits in the RM cells to indicate the highest data rate they can handle. The end-hosts are limited to using the maximum data-rate indicated by the switches to not overwhelm the network and/or switches.

Congestion control mechanisms for LANs and WANs that involve dropping packets are not directly applicable in multiprocessor networks. Some LANs and WANs can drop packets because higher network layers will retransmit dropped packets for reliable communication. The dropped packets serve as implicit hints of network congestion. However, multiprocessor networks are typically expected to guarantee reliable communication. Thus, additional complexity would have to be built-in to store and retransmit dropped packets. The alternative idea of propagating congestion information explicitly can be used.

The challenge is in determining the appropriate set of mechanisms and policies required to provide a self-tuned congestion control implementation for preventing saturation in multiprocessor networks. In Chapter 3, I present my solution for regu-

lar interconnection networks with adaptive routing, wormhole switching, and either deadlock recovery or deadlock avoidance.

My solution—*Tune*—is based on two key innovations that collectively overcome the limitations of previous congestion control techniques. First, I use a *global knowledge based congestion estimation* that enables a more timely estimate of network congestion. The second component is a *self-tuning* mechanism that automatically determines when saturation occurs allowing throttling packet injection. Chapter 3 describes and evaluates the *Tune* congestion control mechanism.

### 2.2.2 Load Balancing in the Minimum Rectangle

Load balancing schemes generally try to distribute packets evenly over the network. One mechanism that has been widely studied to achieve this is to require packets to traverse certain intermediate nodes on the way to their respective destination nodes. With such a mechanism, the problem of balancing load reduces to selecting intermediate nodes evenly over the network. The mechanism of requiring packets to traverse intermediate nodes is also called *multi-phase routing*.

Valiant [64] proposed a two-phase routing scheme that randomly selected a single via-node that was not necessarily in the minimum rectangle. While this approach guarantees balanced loads and good worst case behavior, it effectively doubles the expected path-length for each packet. Nesson et al. [44] describe a non-adaptive routing technique called ROMM routing which uses multiphase-routing by selecting random via-nodes in the minimum rectangle, but their scheme requires twice as many virtual channels as phases to achieve deadlock-free routing in a torus. Towles and Dally [62] demonstrate that the worst case behavior of ROMM routing is poorer than that of static dimension-ordered routing.

The Randomized Load Balancing (RLB) algorithm by Singh et al [55] uses two

levels of load balancing. It uses quadrant selection to balance global link loads. This results in non-minimal routing as well. It uses 2-phase routing within the selected quadrant to balance load locally. This scheme is not adaptive and further, it penalizes performance on well-behaved traffic patterns though it improves the performance for adversarial communication patterns.

Overlay networks [1] on top of the Internet achieve better performance through load balancing and fault tolerance by routing packets via overlay nodes that would not normally be on the path if the underlying Internet mechanism had routed the packet. Resilient overlay networks [2] use fast feedback to detect heavily loaded network regions (as indicated by packet loss rates) faster than the underlying Internet infrastructure and hence are able to direct packets over lightly loaded networks in a timely manner. However, in overlay networks on the Internet, the problem is not only because of slower feedback, but also because the underlying Border Gateway Protocol (BGP) deliberately exposes less information than it has. This is done to achieve scalability. In contrast, in distributed, shared memory multiprocessor networks with adaptive routing, there is no such artificial restriction on routing freedom. But lack of fast feedback can still result in load imbalances and consequent performance degradation.

The Source Demand Routing Protocol (SDRP) is an Internet protocol that lets source nodes specify the inter-domain hops of packets. However, the implementation of SDRP depends on deployment of SDRP-capable routers. In the absence of SDRP routers, routing defaults to ordinary hop-by-hop routing as in Border Gateway Protocol (BGP) and Internet Domain Routing Protocol (IDRP). A version of SDRP for IPv6 called Explicit Routing Protocol (ERP) is also in the works.

For local area networks, Flich et al. [24] propose a scheme to select via nodes, to achieve shortest path routing in *up\*/down\** routed irregular networks such as

Myrinet. This technique of breaking a route into sub-routes retains deadlock-freedom *and* achieves shortest paths.

The key innovations of my *congestion-aware via-routing* scheme, which is described and evaluated in greater detail in Chapter 4, is the use of global information to achieve centralized adaptive routing within the minimum rectangle. The goal is to use global buffer occupancy information and specify, at the source node, that packets move away from heavily loaded network regions. This is different from distributed adaptive routing in which adaptive routing decisions are made at each hop to achieve local load balance. Simulation results show the limitations of load balancing within the minimum rectangle and thus gives the necessary insight for achieving better performance with a new non-minimal routing algorithm—BLAM.

### 2.2.3 Deadlock, Livelock and Performance Issues in Nonminimal Adaptive Routers

Adaptive routing systems (including out base minimal adaptive router) may be prone to deadlocks. Previously I discussed one approach to handling deadlocks in adaptive channels: guarantee that packets are able to make forward progress on a logically separate subnetwork. Deadlock avoidance [18] and deadlock recovery [38] are examples of this approach. Below, I discuss a different approach to deadlock handling in non-minimal adaptive routers that differs in its performance under heavy load and in its livelock-freedom guarantees.

Deflection routing is another class of routing algorithms that avoid deadlocks in virtual cut-through networks by ensuring that no packets are blocked indefinitely. This technique works for networks where the number of input network channels is equal to the number of output channels at each node. This property makes it possible to match every incoming packet to an output channel. However, such a matching

cannot guarantee that all matched pairs correspond to profitable routes. In fact, using this approach to prevent deadlocks requires an unlimited number of misroutes. This approach eliminates the need for deadlock-free escape paths, but guarantees of livelock freedom are either weaker (probabilistic) or they come at the cost of added complexity to implement timestamping and router-wide priorities.

Synchronous deflection routers [23, 31, 42, 57, 58] assume that all packets arrive at an input port at the same time and they are routed to the output ports in a single step. Non-synchronous routers use the same principle of deflection routing but relax the constraint of synchronous operation by adding buffers that can hold incoming packets while waiting for output channels to become free [45, 37]. Such routers need additional mechanisms like the *packet exchange protocol* [45] to prevent deadlocks. This protocol demands that if a node  $a$  sends a packet on a link to a neighboring node  $b$ , node  $a$  should also be prepared to accept a packet from node  $b$ . The Chaos router [37] belongs to this category. Ngai et al. [45] and Coates et al. [11] describe other examples of non-synchronous deflection routers.

Ngai et al. propose a timestamp based technique to eliminate livelocks, by ensuring that the oldest packet in the network is never misrouted. This guarantees deterministic livelock freedom but the implementation of router-wide priorities (e.g. finding “oldest” packet at the router) can complicate the router and add overheads. The S-connect interconnect of the S3.mp system is another example of a non-synchronous deflection router [46]. Incoming packets that cannot be mapped to channels indicated by the routing table may be routed on any random channel. S-connect also uses an implementation of timestamping and router-wide priorities to guarantee livelock freedom.

I use the *Chaos* router [37] as a representative of non-synchronous deflection routing algorithms as it compares very well against previous deflection routers. The *Chaos*

router does not provide deterministic guarantees of livelock-freedom, thus eliminating the need to implement router-wide priorities. Instead, it uses randomization to select packets to misroute causing the probability of a packet remaining undelivered to diminish with time.

In the above discussion, I outlined a design space of routers with minimal adaptive routers – which disallow misroutes and thus eliminate livelocks – on one end, and chaotic routing – which achieves high performance and deadlock freedom by allowing unlimited, lazy misroutes but offers only probabilistic guarantees of livelock-freedom – on the other end. In the Chapter 5, I analyze this design space of routers with respect to network performance, livelock-freedom and deadlock-freedom guarantees. This analysis provides the insight necessary to develop a routing algorithm—BLAM—that has the best features of both classes of routers. Compared to the congestion control approach, BLAM offers an additional advantage as it increases the applied load at which saturation occurs *and* avoids throughput degradation at saturation without placing limits on packet injection beyond those imposed by simple back-pressure.

## Chapter 3:

### Tune: A Self-tuned Congestion Control Mechanism

This chapter presents a self-tuned, global information based, congestion control mechanism—*Tune*—for  $k$ -ary,  $n$ -cube interconnection networks. In general, congestion control techniques work by throttling packets at the source when they detect that saturation is imminent.

One key innovation of *Tune* is the use of global information that takes distant network conditions into account. Global information enables detection of distant network congestion earlier than alternative approaches described in previous studies (Section 2.2.1) that wait for network backpressure to create locally observable indicators of congestion (e.g., local buffer occupancy, timeouts). *Tune* uses global knowledge of the number of full network buffers to estimate network congestion. This global buffer occupancy is compared against a “threshold” to control packet injection. If it is higher than the threshold, packet injection is stopped. When the buffer occupancy drops below the threshold, packet injection is resumed.

The second key aspect of the *Tune* source throttling scheme is a self-tuning mechanism that monitors network throughput and automatically determines the appropriate threshold value. This eliminates manual tuning and allows my scheme to adjust to variations in communication patterns.

I believe that my congestion control mechanism is generally applicable to a broad range of packet-switched, multiprocessor networks, including virtual cut-through [35] networks and wormhole networks [15, 14]. However, in this thesis, I evaluate the technique in the context of wormhole switched,  $k$ -ary,  $n$ -cube networks.

Simulation results for a 16-ary, 2-cube (256 node network) show that the *Tune*

congestion control technique prevents the severe performance degradation caused by network saturation. By limiting packet injection, my scheme sustains high throughput and low latency. Compared to an alternative approach that uses local estimates of congestion [4], *Tune* is superior because global congestion estimation enables it to detect congestion in its early stages. I also show that a single static threshold cannot accommodate all communication patterns because a single threshold overthrottles some workloads and does not prevent saturation in other ones. In contrast, simulations reveal that my self-tuning technique automatically adapts to various communication patterns, including bursts of different patterns.

Section 3.1 and Section 3.2 discuss the two key innovations of this chapter. Section 3.1 presents the proposed global information gathering scheme and Section 3.2 describes the self-tuned congestion control scheme. Section 3.3 and Section 3.4 present the experimental methodology and simulation results, respectively. Section 3.5 summarizes this chapter.

### 3.1 Gathering global information

Any congestion control implementation requires a timely way to detect network congestion. Previous techniques estimate network congestion using a locally observable quantity (e.g., local virtual buffer occupancy, packet timeouts). While these estimates are correlated to network congestion, waiting for local symptoms of network congestion is less useful primarily because, by that time, the network is already overloaded.

Consider the case when network congestion develops at some distance from a given node. Schemes that use local heuristics to estimate congestion rely on back-pressure to propagate symptoms of congestion to the node (e.g. filling up of buffers, increase in queue delays, etc.). The node takes no corrective action until congestion



symptoms are observed locally.

It is possible to detect network congestion in its early stages by taking global conditions into account. To achieve this, I use the fraction of full virtual channel buffers of all nodes in the network as the metric to estimate network congestion. This ensures that far away congestion is accounted for early enough to take corrective action. However, there is additional cost, both hardware and latency, to propagate the global information.

*Tune* counts full buffers to estimate congestion but does not take the distribution of these full buffers among the nodes into account. At first glance, this appears to be a serious limitation because it is unable to distinguish between a case with localized congestion (i.e., a large fraction of full buffers are in relatively few nodes in the network) and a benign case (in which the same number of full buffers are distributed more or less evenly among all the nodes of the network). But the adaptivity of *Tune*'s self-tuning mechanism reduces the impact of this problem by setting the threshold differently in the two cases. *Tune* will set a higher threshold for the benign case than for the case with localized congestion.

In the next section, I show how global information can be gathered with reasonable cost and used to achieve a robust, self-tuned congestion control implementation.

### 3.1.1 Implementing Global Information Gather

*Tune* requires that every node in the network be aware of the aggregate number of full buffers and throughput for the entire network. (The relationship between full buffers, offered load and delivered bandwidth is explained in Section 3.2.) There are a variety of ways to implement this all-to-all communication. In this section, I study two alternatives: meta-packets, and a dedicated side-band.

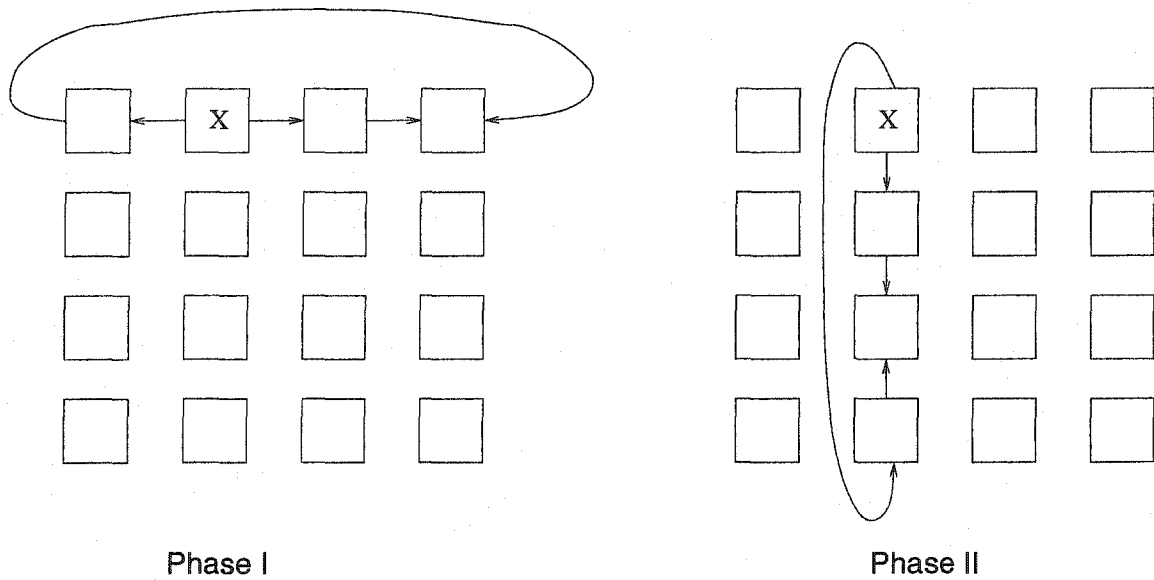
One approach to distribute information is to send out special meta-packets con-

taining the congestion and throughput information. The required all-to-all communication can be guaranteed by this approach. However, guaranteeing delay bounds involves additional complexity. Meta packets flooding the network will also consume some of the bandwidth and may add to the congestion. Adding a high-priority virtual channel reserved for these meta-packets is one way of addressing these concerns. My experiments show that even the addition of a high-priority virtual channel is not sufficient to ensure timely information distribution for the purpose of congestion control. I discuss the meta packet mechanism in Section 3.4.5 in greater detail.

For the rest of this chapter, I use an exclusive side-band reserved for communicating the congestion and throughput information. This is the costliest implementation in terms of additional hardware and complexity. However, it is easy to guarantee delay bounds on all-to-all communication and it does not affect performance of the main data network. While the extra bandwidth available on the side-band could be used for general packet routing, it will only postpone network saturation for a short time, and not provide a complete solution like my congestion control scheme.

Apart from meta packets and side-bands, it may be possible to piggy-back the extra information on normal packets and use this as an information distribution mechanism. However, this approach has the disadvantage that it is difficult to guarantee all-to-all communication. Since only nodes involved in communication see the piggy-backed information, it is possible that some nodes will not see the information for an extended period of time, if at all. As such, I do not consider this information distribution mechanism in this thesis.

To compute the global buffer occupancy using the exclusive side-band, I use a *dimension-wise aggregation* scheme. Assume the side-band incurs a neighbor-to-neighbor communication delay of  $h$  cycles. Every node communicates its number of full buffers and throughput in both directions along the lowest dimension of the

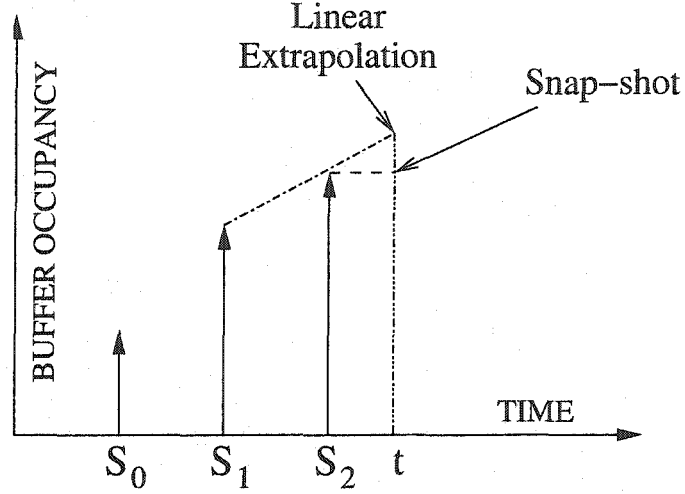


**Figure 3.1:** Dimension-wise global aggregation.

network. Each node that receives such information computes the aggregate and has the aggregate information for all its neighbors along the zeroth dimension at the end of  $k/2$  hops or  $(k/2) * h$  cycles. The nodes then communicate the aggregates to neighbors along the next higher dimension. Continuing this procedure along every dimension, global aggregation in a full-duplex,  $k$ -ary,  $n$ -cube network completes in  $(k/2) * h * n$  cycles. Assuming  $h = 2$ , for the 16-ary, 2-cube network configuration ( $n = 2, k = 16$ ) it takes 32 cycles. I refer to the time for such an all-to-all communication as the *gather-duration* ( $g$ ).

The mechanism described above provides  $g$ -cycle delayed snapshots of the network congestion every  $g$  cycles. *Tune's* congestion control policy requires a comparison, in every cycle, between the current estimated congestion to the threshold. If the system is currently at time  $t$  and the previously observed network snapshots have been observed at  $S_2, S_1, S_0$  and so on, *Tune* must estimate the network congestion at time  $t$  based on the previous snap-shots of global network congestion.

The simplest solution is to use the state observed in the immediately previous



**Figure 3.2:** Estimation of global congestion : Previous Snapshot vs. Linear Extrapolation.

network snapshot until the next snapshot becomes available. I use a linear extrapolation based on the previous two network-snapshots as an estimate of congestion. In general, any prediction mechanism based on previously observed network states can be used to predict network congestion. On average, I found the linear extrapolation technique yields an improvement in throughput of 3% for the deadlock avoidance configuration and 5% for the deadlock recovery configuration over using the state seen in previous network snapshot.

To communicate congestion information, nodes exchange full buffer counts. The number of bits needed to represent this information depends on the number of buffers in the network. The network configuration I use and the number of bits needed to represent congestion information for that configuration is specified in Section 3.3.

In summary, global measurement of virtual buffer occupancy provides an early estimate of network congestion. This estimate is compared against a threshold to determine if packet injection should stop or resume. Obtaining information on the global state of the network is only part of the solution. To translate this congestion estimate to good congestion control, *Tune* has to properly choose the threshold.

*Tune*'s self-tuning mechanism, described in the next section, dynamically tunes the threshold to the appropriate values.

### 3.2 Self-Tuning Mechanism

Proper threshold selection is a crucial component of my congestion control implementation. Inappropriate threshold values can produce unstable behavior at high loads or unnecessarily limit performance for light loads. Furthermore, there is no single threshold that works well for all communication patterns. This section presents a technique to automatically determine the proper threshold value.

The goal of *Tune*'s self-tuning mechanism is to maximize delivered throughput without dramatic increases in packet latency. Therefore, we can view the task as an optimization problem with delivered bandwidth as an objective function dependent on the number of full virtual buffers. Consider the relationship between offered load, full buffers and delivered bandwidth (See Figure 3.3). As offered load increases from zero, the number of full virtual buffers and delivered bandwidth also increase. When saturation occurs, the delivered bandwidth decreases while the number of full virtual buffers continues to increase.

*Tune*'s self-tuning technique is attempting to find the number of full virtual buffers (i.e., the threshold value) that maximizes delivered throughput (B in Figure 3.3). To achieve this, *Tune* uses a hill-climbing algorithm including a technique to avoid local maxima. It is possible obtain a measure of global network throughput (the objective function) in a manner similar to the way *Tune* obtains the global count of full virtual buffers (see Section 3.1.1). Nodes exchange the number of flits delivered in the past  $g$  cycles to measure throughput. Since the maximum possible delivered bandwidth is 1 flit/node/cycle, the count will not exceed  $g * NodeCount$ .

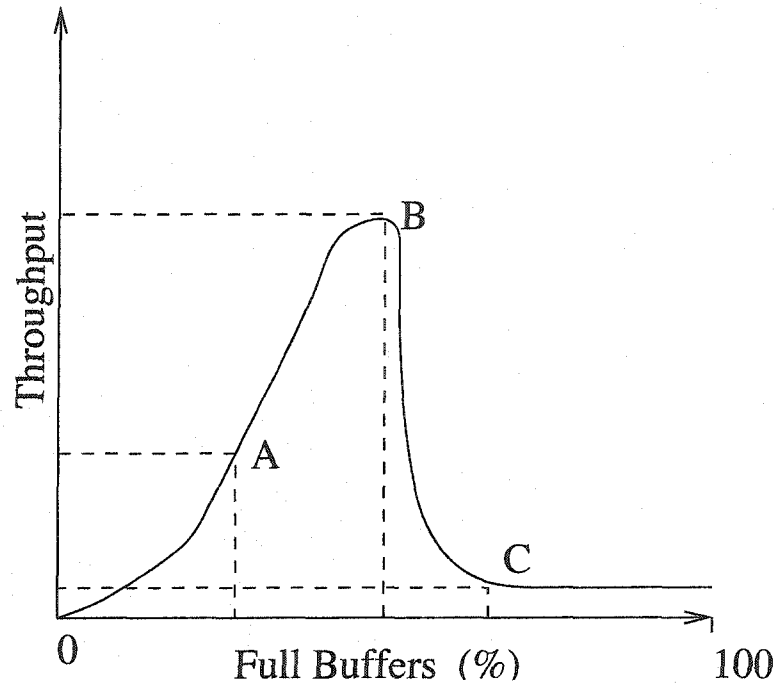


Figure 3.3: Throughput vs. Full Buffers

### 3.2.1 Hill Climbing

To automatically tune the threshold, *Tune* begins with an initial value based on network parameters (e.g., 1% of all buffers). Using intuition about the relationship between the number of full buffers and delivered bandwidth, I specify a tuning decision table that indicates when the threshold value must be increased or decreased. Too low a value (A or lower in Figure 3.3) prevents the network from reaching the peak by over throttling packet injection. In contrast, too high a value pushes the network beyond the peak (C or higher in Figure 3.3), causing saturation just like a network without any congestion control.

A discussion of *Tune*'s hill-climbing component of its self-tuning mechanism will have to answer the following questions.

- *How often does Tune alter this estimate?* It uses a very simple scheme in which it updates the estimate at a constant frequency. The tuning period is an

Drop in Bandwidth > 25%?	Currently Throttling?	
	Yes	No
Yes	Decrement	Decrement
No	Increment	No Change

**Table 3.1:** Tuning decision table

exact multiple of the *gather-duration*. If the tuning period is very large, there is likely to be slow and inefficient tuning leading to network underutilization or network saturation. If it is too small, short-lived crests and troughs in throughput could alter the estimate. However, in my experiments, I found that, for a reasonable range of values (32 cycles to 192 cycles) the performance did not alter significantly. In most experiments, I use a 96 cycle tuning period.

- *What policy determines whether the tuning mechanism should increment or decrement the estimate?* Tune uses the policy outlined in Table 3.1. The two dimensions in the table correspond to observed network throughput and whether or not the network is currently throttled. A tuning decision is made once every *tuning period*. A “drop-in-throughput” is said to occur when the throughput observed at the end of one tuning period is less than 75% of the throughput seen at the end of the previous tuning period. The drop in throughput is not necessarily an indication of saturation. It may also be due to a reduction in offered load. My scheme treats both cases identically and decrements the estimate. If indeed the drop in throughput was due to a reduction in offered load, the reduced threshold will not harm performance as the threshold will be incremented gradually to the desired level when offered load increases.
- *By what increments/decrements does Tune alter the estimate?* Tune uses constant additive increments and decrements. Simulations with multiplicative

decrements show that constant additive tuning is adequate for effective self-tuning (See Section 3.4.3). For a reasonable range of values, (1% to 4% of all buffers) performance is insensitive (within 4%) to variations in increment/decrement values in a reasonable range of values and all values succeed in preventing performance degradation at saturation. The simulations showing the variation in Tune performance with different increment/decrement values are presented in Section 3.4.3. I use an increment of 1% of all buffers and a decrement of 4% of all buffers. For the network I consider, this corresponds to an increment of 30 and a decrement of 122.

To avoid local maxima, Tune resets the threshold to  $\min(N_{max}, T_{max})$  when the throughput achieved in any single tuning period drops below 50% of the *max* value. It restarts *max* computation if the threshold is reset to  $\min(N_{max}, T_{max})$  for  $r = 5$  consecutive tuning periods.

To summarize the proposed implementation of *Tune*, it uses a sideband for communicating global congestion and throughput information. This mechanism provides delayed (32 cycles for the 16-ary, 2-cube) global snapshots of full buffer counts and throughput. *Tune* updates its threshold every 96 cycles in increments of 30 and decrements of 122. To avoid local maxima, *Tune* uses the scheme outlined in Section 3.2.2 with  $r = 5$ .

### 3.2.2 Avoiding Local Maxima

To avoid settling at local maxima, *Tune* “remembers” the conditions that existed when maximum throughput was achieved. To do this, it keeps track of the maximum throughput (*max*) achieved during any single tuning period and remember the corresponding number of full buffers ( $N_{max}$ ) and threshold ( $T_{max}$ ).

If the throughput in any tune-period drops significantly below the maximum



throughput, *Tune* tries to recreate the conditions that existed when maximum throughput was achieved. This is done by setting the threshold to  $\min(T_{max}, N_{max})$ . If  $N_{max}$  is higher than  $T_{max}$ , it means that the network was throttling with a threshold value of  $T_{max}$  when it achieved the maximum observed throughput. In this case, the threshold is set to  $T_{max}$  so that the network can throttle new packets and drain existing packets till it reaches the desired load level. If, on the other hand,  $N_{max}$  is smaller than  $T_{max}$ , then setting the threshold to  $N_{max}$  is a better choice because it is possible that  $T_{max}$  is not low enough to prevent saturation. This guarantees that the tuning mechanism is not stuck at a local maximum after the network saturates.

It is possible that the threshold value which sustains high throughput for one communication pattern is not low enough to prevent saturation for another communication pattern. The *Tune* congestion control mechanism detects and adapts the threshold to such changes. If *Tune* resets the threshold to  $\min(T_{max}, N_{max})$  for  $r$  consecutive tuning-periods, this means that even the  $\min(T_{max}, N_{max})$  value is too high to prevent saturation, and *Tune* must recompute  $max$  value. In this case,  $max$  is reset to zero and the maximum locating process starts all over again. This ensures that the threshold adapts to changing communication patterns. I use  $r = 5$  for all experiments.

### 3.2.3 Summary

The above discussion provides a general overview of a technique we believe can provide a robust, self-tuned congestion control. *Tune* gathers full-buffer counts and throughput measurements every 32 cycles. The full-buffer counts are used to estimate current congestion using linear extrapolation. This estimate is compared to a threshold to decide whether new packets are throttled. *Tune* uses a hill climbing algorithm to update its threshold every 96 cycles in increments and decrements of

1% and 4% of total buffer count, respectively. The hill climbing algorithm, when used alone, is susceptible to settling on local maxima after network saturation. *Tune* includes a mechanism to prevent this from happening by remembering maximum ( $max$ ) observed throughput. Finally, *Tune* recomputes the maximum ( $max$ ) if the threshold is reset for  $r = 5$  consecutive tuning periods.

On a high level, *Tune* is somewhat analogous to TCP's self-tuning congestion control. Both have an idea of what the network performance should be. Expected round-trip time (RTT) in the case of TCP and  $max$  throughput in *Tune*'s case. Both schemes allow offered load to incrementally increase as long as network performance is not penalized. The sliding window size increases as long as no packets are dropped in the case of TCP and threshold increases as long as there is no decrease in throughput in *Tune*. Both techniques take corrective action if network performance suffers. TCP reduces its window size and *Tune* either decrements the threshold or resets it to  $\min(N_{max}, T_{max})$ . Finally, both schemes periodically refresh their estimate of network performance. TCP recomputes expected round-trip-time if packets are dropped, whereas *Tune* recomputes  $max$ ,  $N_{max}$  and  $T_{max}$  if  $max$  is stale, i.e. if there are  $r$  consecutive corrective actions.

### 3.3 Methodology

To evaluate the *Tune* self-tuned congestion control scheme, I use the `flexsim` [60] simulator. I simulate a 16-ary, 2-cube (256 nodes) with full duplex physical links. Each node has one injection channel (through which packets sent by that node enter the network) and one delivery channel (through which packets sent to that node exit the network). I use three virtual channels per physical channel and edge-buffers (buffers associated with virtual channels) which can hold eight flits.

The router's arbiter is a central resource which only one packet can use at a time

and there's a one cycle routing delay per packet header. Packets obtain control of the router's arbiter on a *demand-slotted round-robin distribution*. This is not a bottleneck because routing occurs only for the header flit of a 16-flit packet. The remaining flits simply stream behind the header flit along the same switch path. It takes one cycle per flit to traverse the cross-bar switch and one cycle per flit to traverse a physical link.

I evaluate the *Tune* congestion control mechanism with both deadlock avoidance and deadlock recovery mechanisms. Deadlock avoidance uses the method proposed by Duato [18] with one escape virtual channel using oblivious dimension-order routing. I use the Disha [38] progressive deadlock recovery scheme with a time-out of 8 cycles.

All simulations execute for 60,000 cycles. However, I ignore the first 10,000 cycles to eliminate warm-up transients. Most results are presented in two parts: normalized delivered throughput (accepted flits/node/cycle) and average packet latency versus offered load in terms of packet injection rate.

The default load consists of each node generating 16 flit packets at the same fixed rate. I consider four different communication patterns, *uniform random*, *bit-reversal*, *perfect-shuffle* and *complement*. These communication patterns differ in the way a destination node is chosen for a given source node with bit co-ordinates  $(a_{n-1}, a_{n-2}, \dots, a_1, a_0)$ . The bit co-ordinates for the destination nodes are  $(a_{n-2}, a_{n-3}, \dots, a_0, a_{n-1})$  for *perfect shuffle*,  $(a_{n-1}, a_{n-2}, \dots, a_1, a_0)$  for *complement* and  $(a_0, a_1, \dots, a_{n-2}, a_{n-1})$  for *bit-reversal*.

I use synthetic workload, instead of full-blown multiprocessor workloads, for three reasons. First, my simulation environment cannot handle full-blown multiprocessor workloads. Second, my packet generation frequency corresponds to realistic miss rates in databases and scientific applications, which gives me confidence in the results. Third, the synthetic workloads nicely demonstrate the problem of network saturation

and avoids interactions with application-specific features.

For the network under consideration (with 3072 buffers), 12 bits are enough to count all buffers in the network. The network configuration needs 13 bits to represent the maximum possible aggregate throughput  $g * NodeCount * MaxTraffic = 32 * 256 * 1 = 8192$  flits). Thus, *Tune* needs a total of 25 bits for the sideband signals. However, in Section 3.4.2, I demonstrate that it is possible to send these 25 bits using 9-bit sideband channels with very little performance degradation.

For comparison, I also simulate the At-Least-One (*ALO*) [4] congestion control scheme. *ALO* estimates global network congestion locally at each node. If at least one virtual channel is free on every *useful*<sup>1</sup> physical channel or if at least one *useful* physical channel has all its virtual channels free, then packet injection is allowed. Otherwise, new packets are throttled.

### 3.4 Simulation Results

The primary conclusions from my simulations are:

- *Tune* provides high performance consistently across different communication patterns and offered load levels.
- *Tune* outperforms an alternative congestion control technique that uses local estimates of congestion.
- *Tune*'s self-tuning mechanism adapts the threshold dynamically to varying workloads and to bursty traffic.

The remainder of this section elaborates on each of these items.

---

<sup>1</sup>*useful* is an output channel that can be used without violating the minimal-routing constraint.

### 3.4.1 Overall Performance

I begin by examining the performance of a complete implementation, as described in Sections 3.1.1 and 3.2. Figure 3.4 shows the bandwidth and latency for a *uniform-random* traffic pattern for both deadlock recovery (a & b) and deadlock avoidance (c & d). Note the logarithmic scale used on the y-axis for the latency graphs (b & d).

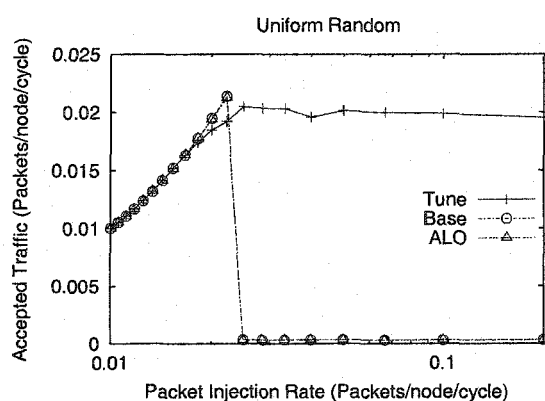
The curve for the base case illustrates the network saturation problem. As load increases, the network throughput increases to a certain extent. However, at saturation, there is a sudden drop in throughput since only the escape channels are available to drain deadlocks. The configuration with deadlock recovery has lower bandwidth beyond saturation because Disha deadlock recovery requires that a packet obtain exclusive access to the deadlock-free path. In contrast, the deadlock avoidance scheme can break multiple deadlock cycles concurrently.

The results for uniform random traffic in Figure 3.4 clearly show the key point that the my congestion control technique (*Tune*) is stable at high loads. The *ALO* congestion control scheme improves performance in the early stages of congestion for the deadlock avoidance case, but it does exhibit severe performance degradation eventually. *Tune*, however, maintains latency and throughput close to the peak values.

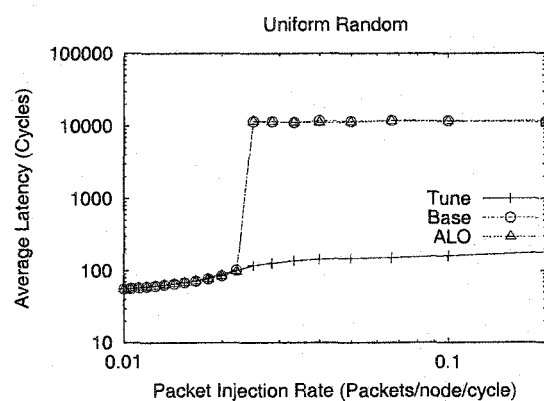
Figures 3.5, 3.6 and 3.7 show similar performance curves for *bit reversal*, *perfect shuffle* and *complement* traffic patterns respectively. As in the case of uniform random traffic, *Tune* sustains near-peak throughput and latency network operation whereas the throughput for the base case and ALO see degraded performance beyond saturation load. Note, the Y-axis limits are different for the graphs since the peak throughput varies with traffic patterns. There are some data points for the base case ( $\circ$ ) and ALO ( $\triangle$ ) in Figures 3.5, 3.6 and 3.7 when the latency shows sudden im-

provements beyond saturation loads even though the throughput remains poor. This is because many packets suffer extended latencies and are not delivered till the end of the simulation do not contribute to the average latencies. This manifests itself in the form of lower number of delivered packets. In general, the latency numbers are not meaningful when the throughput has collapsed since overall system performance depends on both throughput and latency.

### Deadlock Recovery

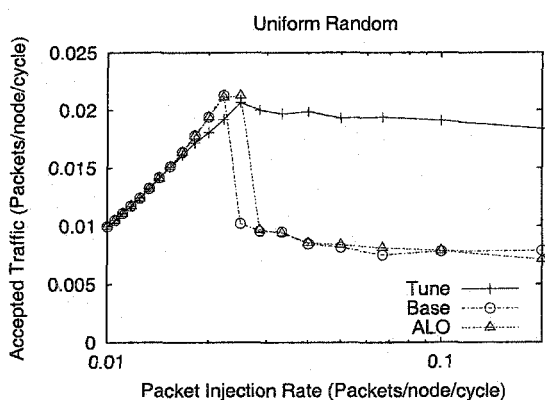


(a) Delivered Throughput vs. Offered Load

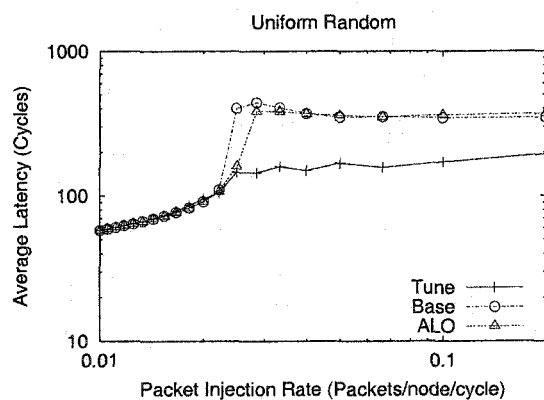


(b) Average Latency vs. Offered Load

### Deadlock Avoidance



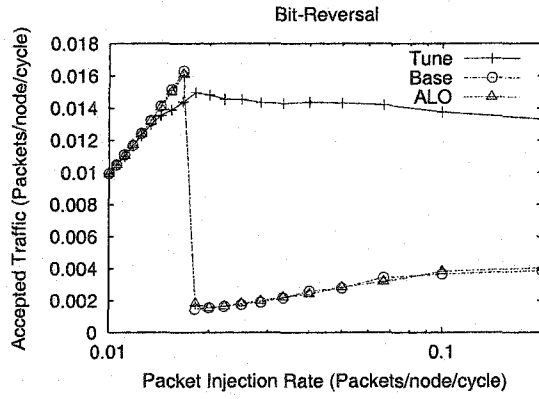
(c) Delivered Throughput vs. Offered Load



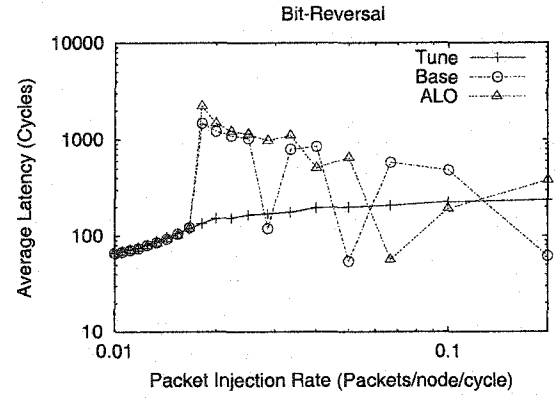
(d) Average Latency vs. Offered Load

**Figure 3.4: Overall Performance With Random Traffic**

### Deadlock Recovery

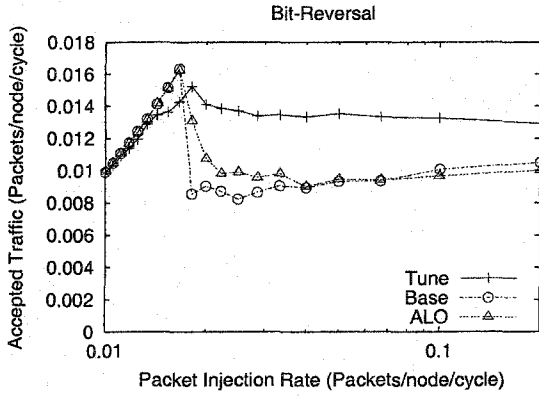


(a) Delivered Throughput vs. Offered Load

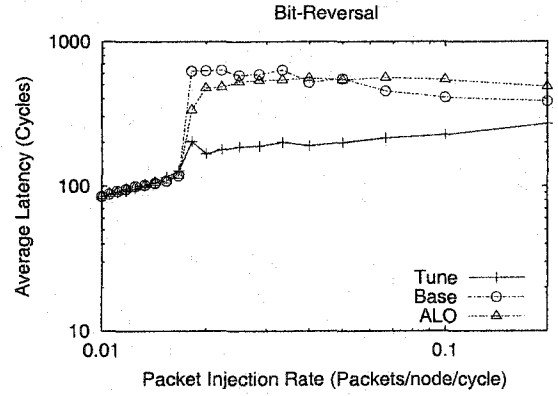


(b) Average Latency vs. Offered Load

### Deadlock Avoidance



(c) Delivered Throughput vs. Offered Load

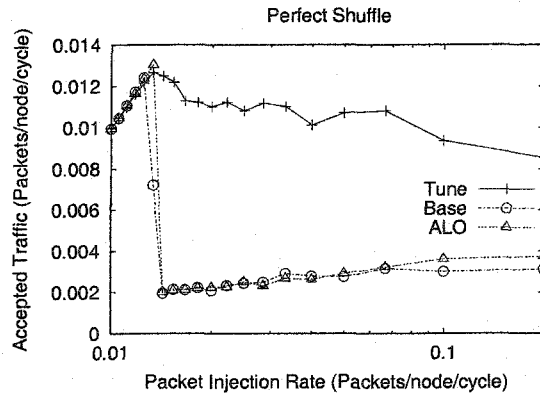


(d) Average Latency vs. Offered Load

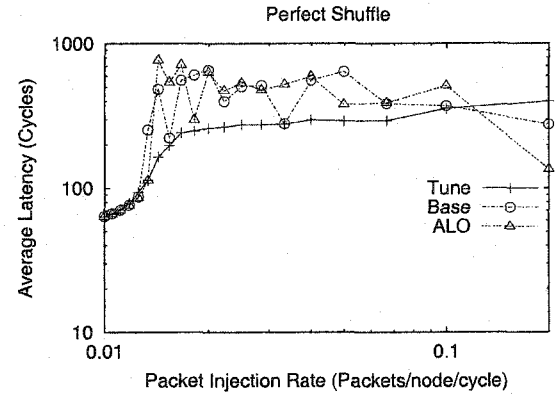
**Figure 3.5:** Overall Performance With Bit-Reversal Traffic Pattern



### Deadlock Recovery

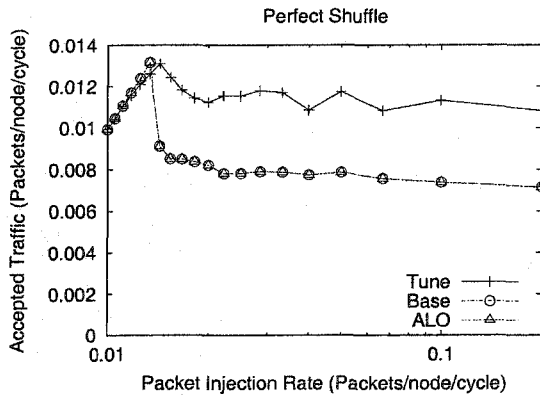


(a) Delivered Throughput vs. Offered Load

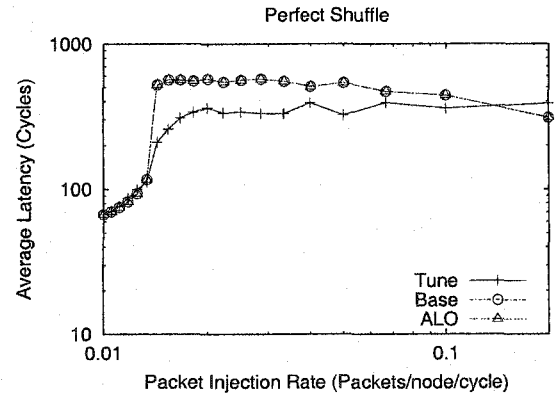


(b) Average Latency vs. Offered Load

### Deadlock Avoidance



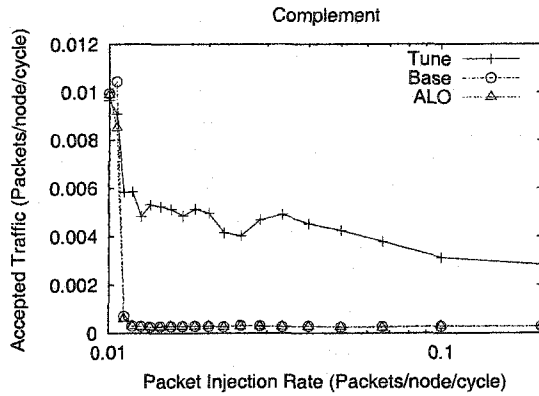
(c) Delivered Throughput vs. Offered Load



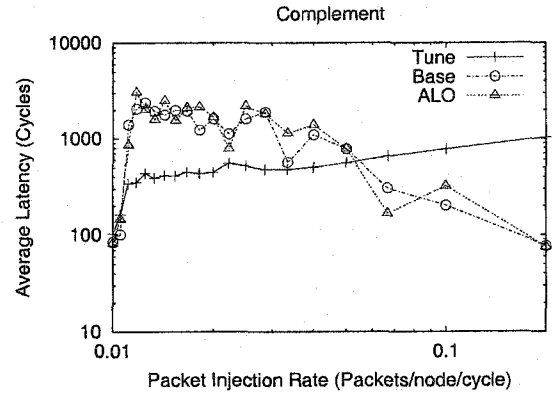
(d) Average Latency vs. Offered Load

**Figure 3.6: Overall Performance With Perfect-Shuffle Traffic Pattern**

### Deadlock Recovery

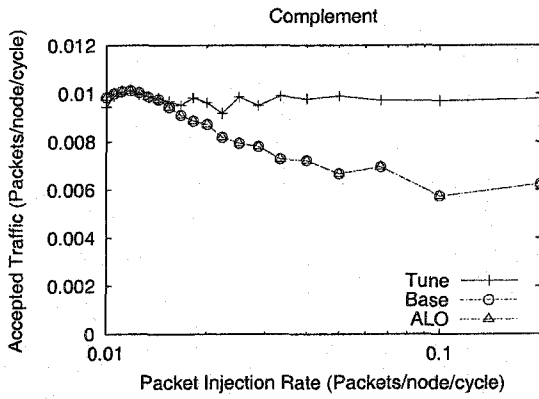


(a) Delivered Throughput vs. Offered Load

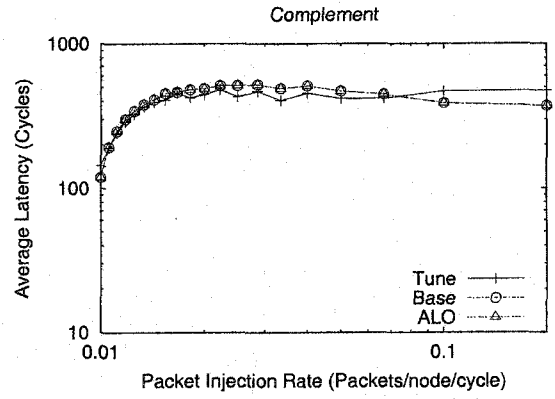


(b) Average Latency vs. Offered Load

### Deadlock Avoidance



(c) Delivered Throughput vs. Offered Load



(d) Average Latency vs. Offered Load

**Figure 3.7:** Overall Performance With Complement Traffic Pattern

### 3.4.2 Obtaining Global Information

This section considers the effect of varying the delay ( $h$ , the time for one neighbor-to-neighbor communication on the side-band) to gather global information on overall performance. If  $h = 2$ , that corresponds to a global-aggregation time ( $g$ ) of 32 cycles. For  $h = 3$ ,  $g = 48$  and for  $h = 6$ ,  $g = 96$ . Note that the tuning period remains 96 cycles.

Intuitively, we know that stale information is less useful than fresh information. That is indeed the case if *Tune* uses the stale information directly. However, with linear extrapolation, it is possible to estimate current information from stale information with some accuracy. In fact, for “noisy” curves, linear extrapolation may reflect the trends better than increased sampling.

Figure 3.8 shows quantitatively the performance losses due to delays in gathering global information. Figure 3.8(a) and (b) show the results for the deadlock recovery configuration. While there is very little performance degradation going from a 32 cycle to a 48 cycle gather delay, increasing the gather delay to 96 cycles starts to hurt performance at heavy loads because the stale information cannot be effectively used by *Tune*. For the deadlock avoidance configuration (Figure 3.8(c) and (d)), *Tune* suffers a small penalty even with a gather delay of 96 cycles. There are some points where *Tune* works better with a 48 cycle delay than with a 32 cycle delay. For the points in question, linear extrapolation gives an average increase in throughput of 10% over the “previous snapshot” approach with a delay of 48 against an average increase in throughput of 4% with a delay of 32. Figure 3.8 also shows that a delay of 96 cycles shows performance degradation due to very stale information.

Note, increasing the neighbor-to-neighbor communication delay can be treated as equivalent to narrowing the side-band signal. Consider the 16-ary, 2-cube configura-

tion discussed in this section. It used a 25-bits wide side-band with a neighbor-to-neighbor communication latency of  $h = 2$  cycles. If the side-band were to be narrowed down to 9-bits wide, the same 25 bits of information can still be transmitted in three instalments of 9 bits each. Breaking up the information transfer into three steps will mean that the effecting neighbor-to-neighbor communication delay is now 6 cycles. This way, the results in Figure 3.8 can also be interpreted as the effect of varying the bit-width of the side-band.

### Self-Tuning

This section describes two important aspects about *Tune*'s self-tuning technique. First, I show the importance of having a congestion control mechanism that adapts to the congestion characteristics of different communication patterns. This is followed by an examination of the hill-climbing algorithm and the technique for avoiding local maxima.

Recall from Figure 1.4 that saturation occurs at different levels of offered load for random and complement communication patterns. These different levels of offered load correspond to different buffer occupancies in the network. If saturation was occurring at the same buffer occupancies for different workloads, a well-chosen, single, static threshold could prevent network saturation. To show that this is not so, Figure 3.9 compares the performance on the deadlock recovery network configuration of a congestion control mechanism with static thresholds to *Tune*.

Consider the uniform random (the four solid lines) and complement (the four dashed lines) communication patterns in Figure 3.9. A static threshold of 250 (8% buffer occupancy) works very well for random traffic but the same threshold is unable to prevent saturation for the complement communication pattern. In contrast, a static threshold of 50 (1.6% buffer occupancy) works well for the complement com-

munication pattern but over-throttles the random traffic. This indicates that the buffer occupancy at which the network saturates is not uniform across communication patterns. Therefore, it is necessary to have a self-tuning mechanism that adapts the threshold as communication patterns change.

To understand the behavior of *Tune*'s self-tuning technique, I analyze its operation for a specific configuration. As stated in Section 3.2, I use a *gather-period* ( $g$ ) of 32 cycles, a tuning period of 96 cycles, an increment of 1% of all virtual channel buffers and a decrement of 4% of all virtual channel buffers. The load is of uniform random distribution with a packet regeneration interval of 10 cycles and I use the deadlock avoidance configuration. With these parameters, Figure 3.10(a) shows the tuning of the threshold over time for the duration of the simulation. Recall, the first 10,000 cycles are ignored to eliminate start-up transients. Figure 3.10(b) shows the throughput achieved over the same interval.

The hill climbing mechanism tries to increase the threshold as long as there is no decrease in bandwidth and tries to scale back when bandwidth decreases. But it can settle down at a local maximum when the decrease in bandwidth happens gradually. When this occurs, the network "creeps" into saturation and throughput falls.

Without a mechanism to avoid local maxima, the hill climbing algorithm can settle on a local maximum corresponding to deep saturation. The solid line in Figure 3.10 shows this behavior. A gradual drop in throughput begins at approximately 26,000 cycles. Recall that *Tune* decrements the threshold only when there is a throughput drop of 25% or more in any tuning period. Figure 3.10(a) shows that, although there are many decrements, the gradual nature of the decrease in throughput results in an overall rise in the threshold, eventually saturating the network.

The dashed line in Figure 3.10 shows the behavior of *Tune*'s technique to avoid local maxima. The sharp dip in the threshold value (specifically the one at ap-

proximately 26,000 cycles) illustrates the corrective action taken when the network “creeps” towards saturation. As a result, *Tune* avoids the performance degradation at saturation and sustain higher throughput.

### 3.4.3 Varying *Tune*’s increment/decrement values

In this section, I take a look at the effects of varying the increment/decrement values of *Tune*’s hill-climbing mechanism.

The goal was not to develop the fastest and most robust tuning mechanism, but to demonstrate that it is possible to do this for *some* values of increments/decrements to effectively prevent performance degradation at saturation. As such, my experiments are fairly coarse-grained to select some values which demonstrate the effectiveness of global information based, self-tuned congestion control in the tightly-coupled, inter-connection network context. Feed-back based tuning mechanisms that employ more sophisticated control theoretic techniques will only improve the performance of *Tune*.

The results in Figure 3.11 shows the experiments with various values of decrements (1% to 4%) including one case with multiplicative decrement (factor of 2). I fix the increment at 1%. The results show that the tuning mechanism is fairly insensitive to changes in decrement values and that the difference in performance for different decrements is not significant.

### 3.4.4 Bursty Traffic

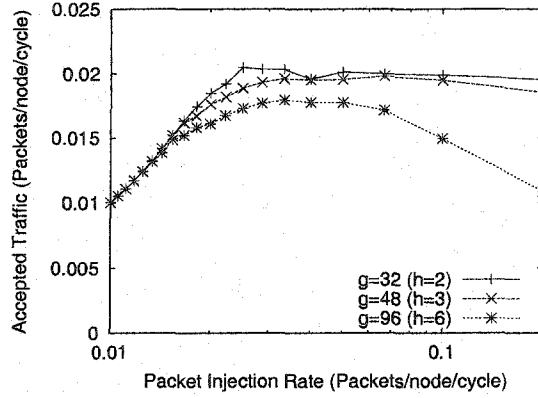
Previous sections evaluated *Tune* on steady traffic only. Real applications do not typically generate steady communication traffic. The communication patterns and the load levels generated by real applications are typically bursty in terms of load levels and of varying communication patterns during different phases of the application. To confirm that *Tune*’s self-tuning mechanism works well under varying load,

I use a bursty load created by alternating low loads and high loads. In addition, I also change the communication pattern for each high load burst. The offered bursty load is shown in Figure 3.12. In the low load phase, the communication pattern is *uniform random* and every node tries to generate one packet every 1,500 cycle period (corresponding to a packet injection rate of 0.00067 packets/node/cycle). In the high load phase, every node tries to generate a packet every 15 cycles (corresponding to a packet injection rate of 0.067 packets/node/cycle). The communication pattern in each of these high load bursts is different and is indicated in Figure 3.12.

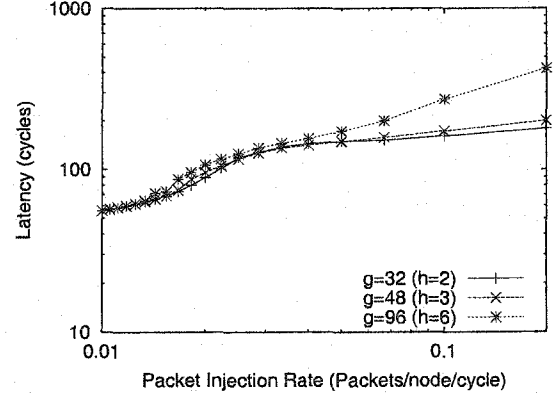
Figure 3.13(a) and Figure 3.13(b) show the delivered throughput with bursty load for the deadlock recovery and the deadlock avoidance configurations, respectively. With deadlock recovery, the average packet latency for *Base*, *ALO* and *Tune* configurations are 2838 cycles, 2571 cycles and 161 cycles respectively. With deadlock avoidance, the average packet latency for *Base*, *ALO* and *Tune* configurations are 520 cycles, 509 cycles and 163 cycles respectively. In the high-load phase, *Tune* consistently delivers sustained throughput and predictable latencies. The *ALO* scheme and the base scheme initially ramp up the throughput but throughput collapses soon thereafter due to network saturation.

The deadlock recovery results exhibit an interesting phenomenon in the *Base* and *ALO* schemes (Figure 3.13a). There are small bursts in throughput long after the offered load is reduced. This is because the network absorbs the heavy offered load but goes into deep saturation with many deadlock cycles. We observe this happening approximately between 20,000 and 21,000 cycles in Figure 3.13(a). There is a period when network packets are draining through the limited escape bandwidth available (approximately between 21,000 and 27,000 cycles). It is only when the deadlock cycles break that full adaptive routing begins again. The network then drains quickly showing the spurt in throughput (approximately between 27,000 and 29,000 cycles).

### Deadlock Recovery

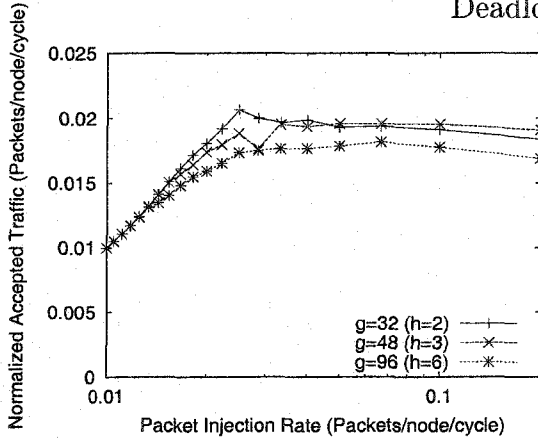


(a) Delivered Throughput vs. Offered Load

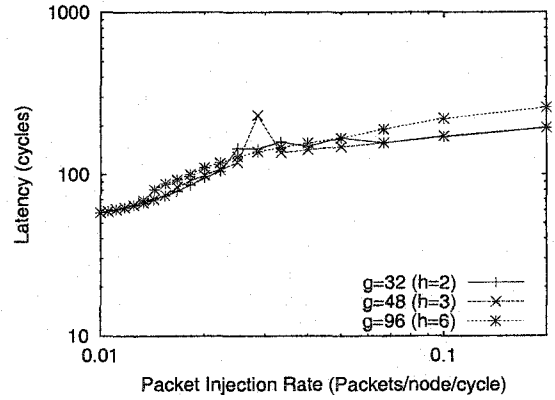


(b) Average Latency vs. Offered Load

### Deadlock Avoidance



(c) Delivered Throughput vs. Offered Load



(d) Average Latency vs. Offered Load

**Figure 3.8:** Effect of Global Information Gathering Delays for Deadlock Recovery (a & b) and Deadlock Avoidance (c & d).



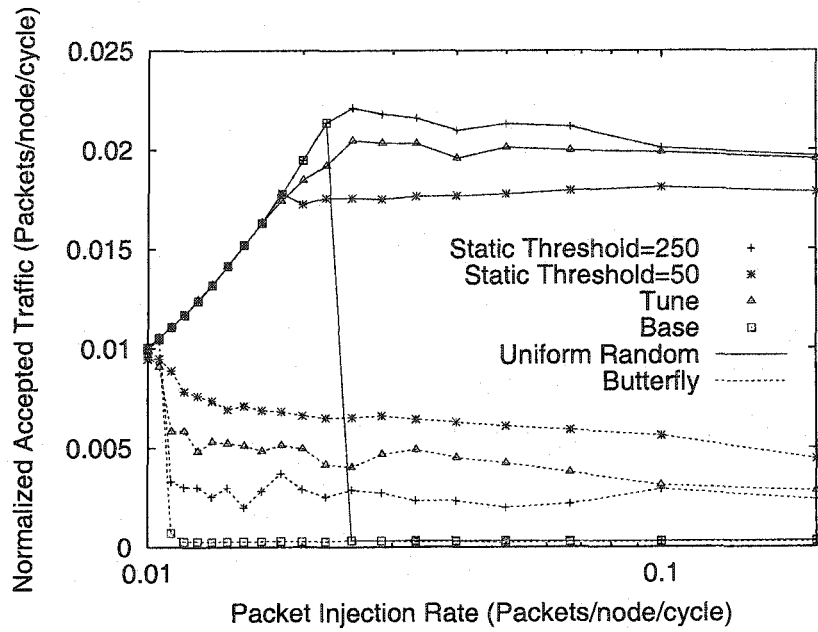


Figure 3.9: Static Threshold vs. Tuning.

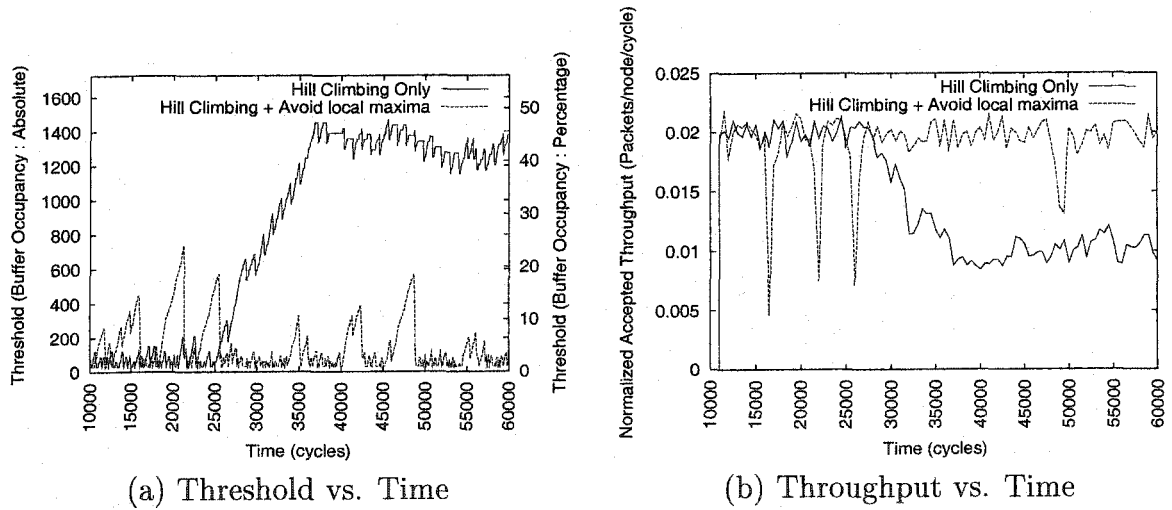
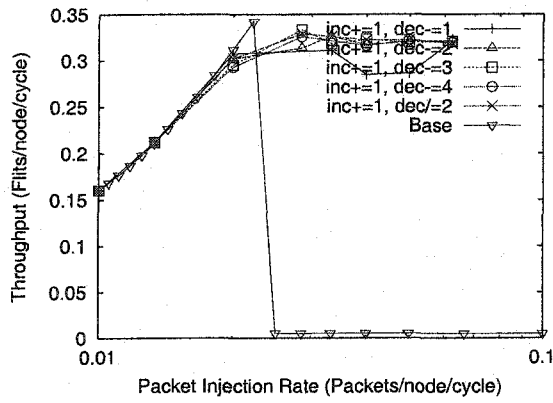
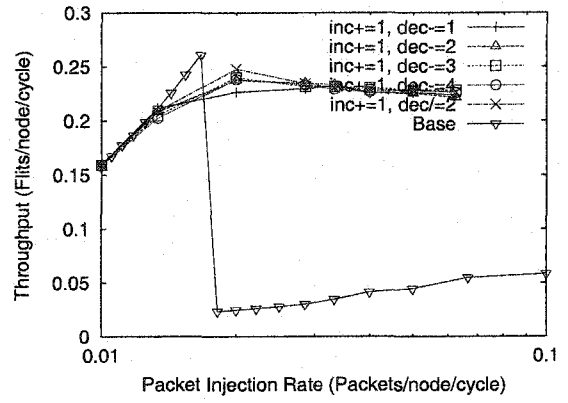


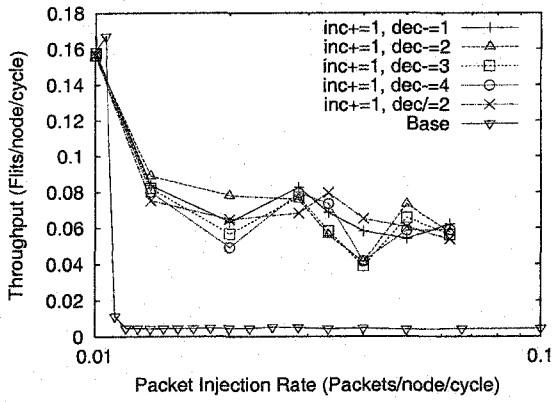
Figure 3.10: Self-Tuning Operation : An Example



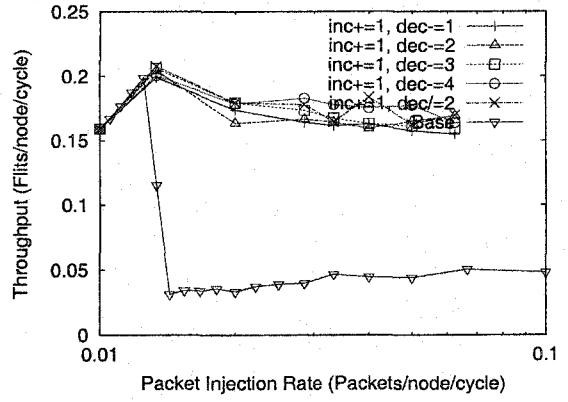
(a) Uniform Random



(b) Bit Reversal



(c) Complement



(d) Perfect Shuffle

Figure 3.11: Choice of increment/decrement quanta

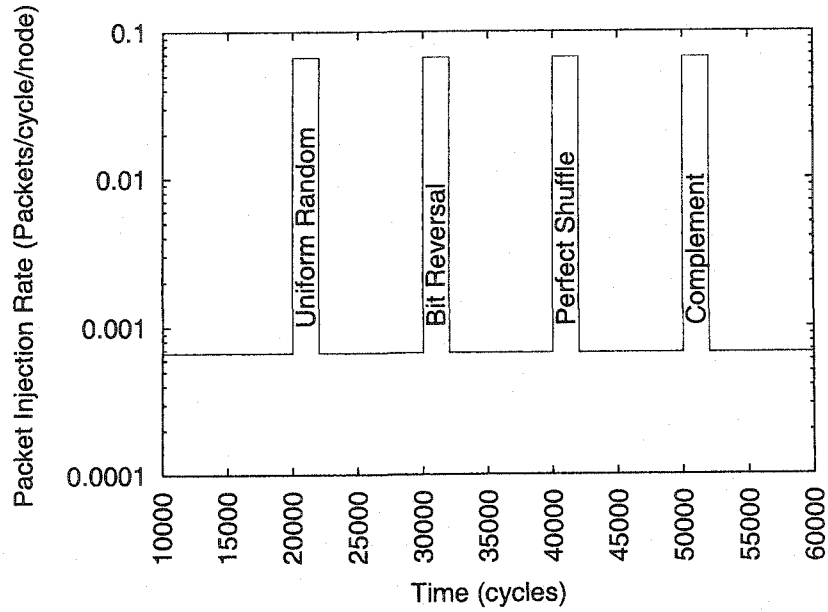


Figure 3.12: Offered bursty load

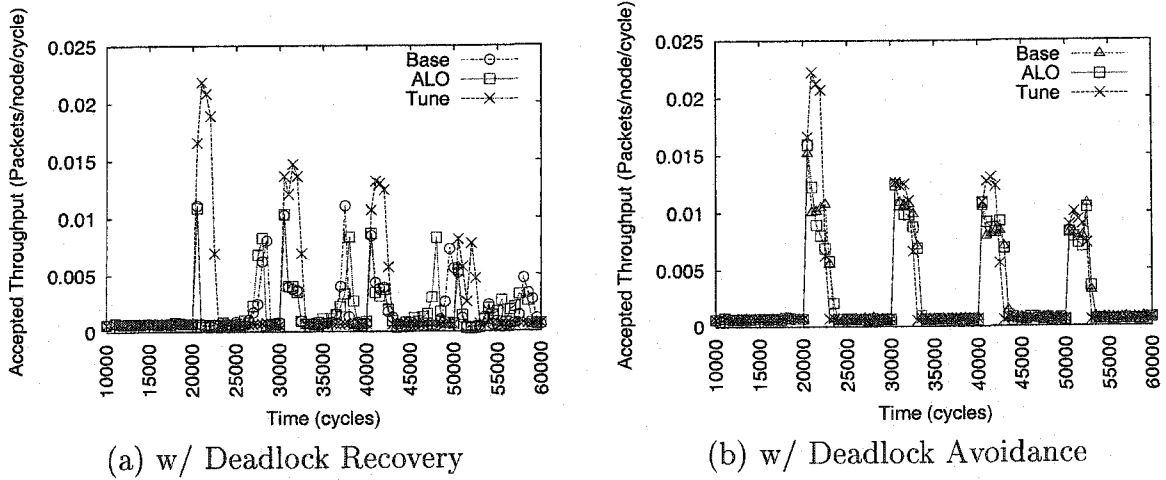


Figure 3.13: Performance with Bursty Load : Delivered Throughput

### 3.4.5 Distributing Information using Meta Packets

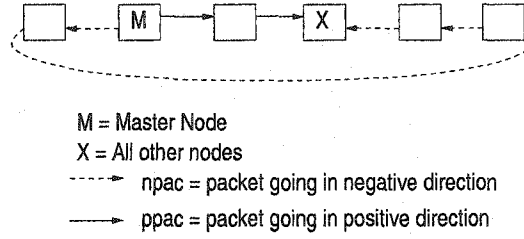
In this section, I evaluate an alternative mechanism to distribute global information. Buffer occupancy and throughput information are distributed in special meta-packets over the same physical links as regular data packets but they use an exclusive, additional virtual channel. The additional virtual channel for meta-packets also has higher priority than other virtual channels to ensure delay bounds on information distribution.

Simulations demonstrate that the *Tune* congestion control scheme is not effective in preventing saturation when using meta-packets as the information distribution mechanism. This is partly because of the contention for physical links caused by meta-packets and partly because of the larger gather times resulting in stale information. In the rest of the section, I first describe the aggregation technique using meta-packets and then present experimental results using 2-flit meta-packets.

#### Dimension-wise Aggregation using Meta Packets

As in the case of side-band, the mechanism aggregates quantities along lower dimensions before proceeding along higher dimensions. I now describe the method by which my mechanism aggregates buffer occupancy and throughput along a single dimension.

I define a  $k$ 'th dimension row as the set of nodes which differ only in their  $k$ 'th dimension co-ordinate. (In a  $k$ -ary,  $n$ -cube, there are  $k^{n-1}$  rows of  $k$  nodes each along each dimension.) I select one *master* node in each row for a given dimension. This master node sends out two meta packets (*ppac* and *npac* as shown in Figure 3.14), one in each direction. One of these packets has the *master* node's buffer occupancy and throughput information and the other packet is initialized to zero. These packets traverse the entire row till they are delivered back to the *master* node after completing a circuit. Each intermediate node modifies the meta-packet by adding in its own



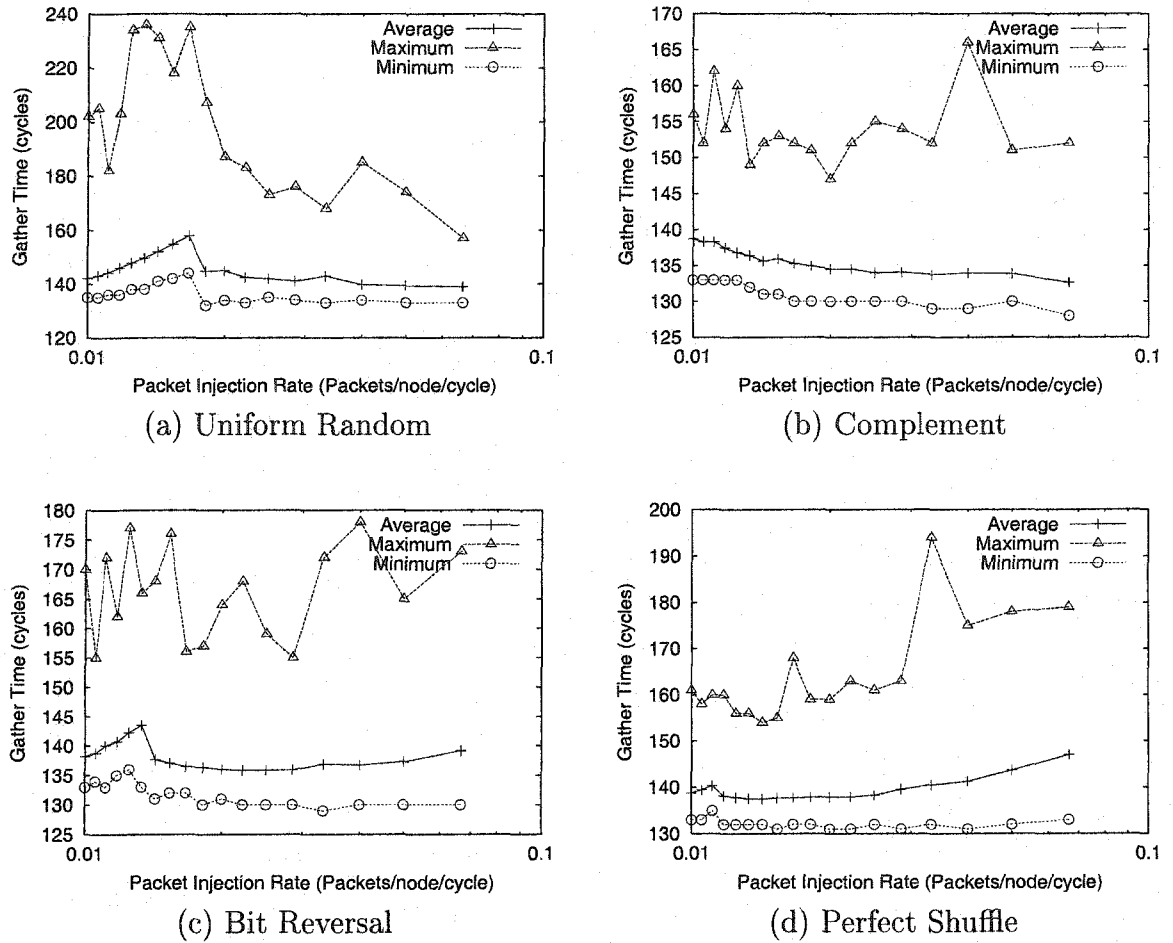
**Figure 3.14:** Row aggregation using two meta packets

buffer occupancy and throughput information before forwarding the packet. Since each node modifies the contents of the packet before forwarding it onwards, I assess a one cycle adder-delay for the meta packets. When the two packets arrive at some node  $X$  in the row, they contain the partial aggregates of information for two disjoint sections of the row (See Figure 3.14). The node  $X$  accumulates these partial aggregates contained in the meta packets and adds in its own local buffer occupancy and throughput information to compute the aggregate for the entire row.

This cycle repeats for rows along the next dimension but instead of adding in local information, each node adds in the row-aggregate along the lower dimensions. After  $n$  such phases, aggregation in a  $k$ -ary,  $n$ -cube is complete. In the next section, I present experimental results that quantify the gather times using the information distribution schemes described above and evaluate the performance of the *Tune* congestion-control scheme using meta-packets.

## Experimental Results

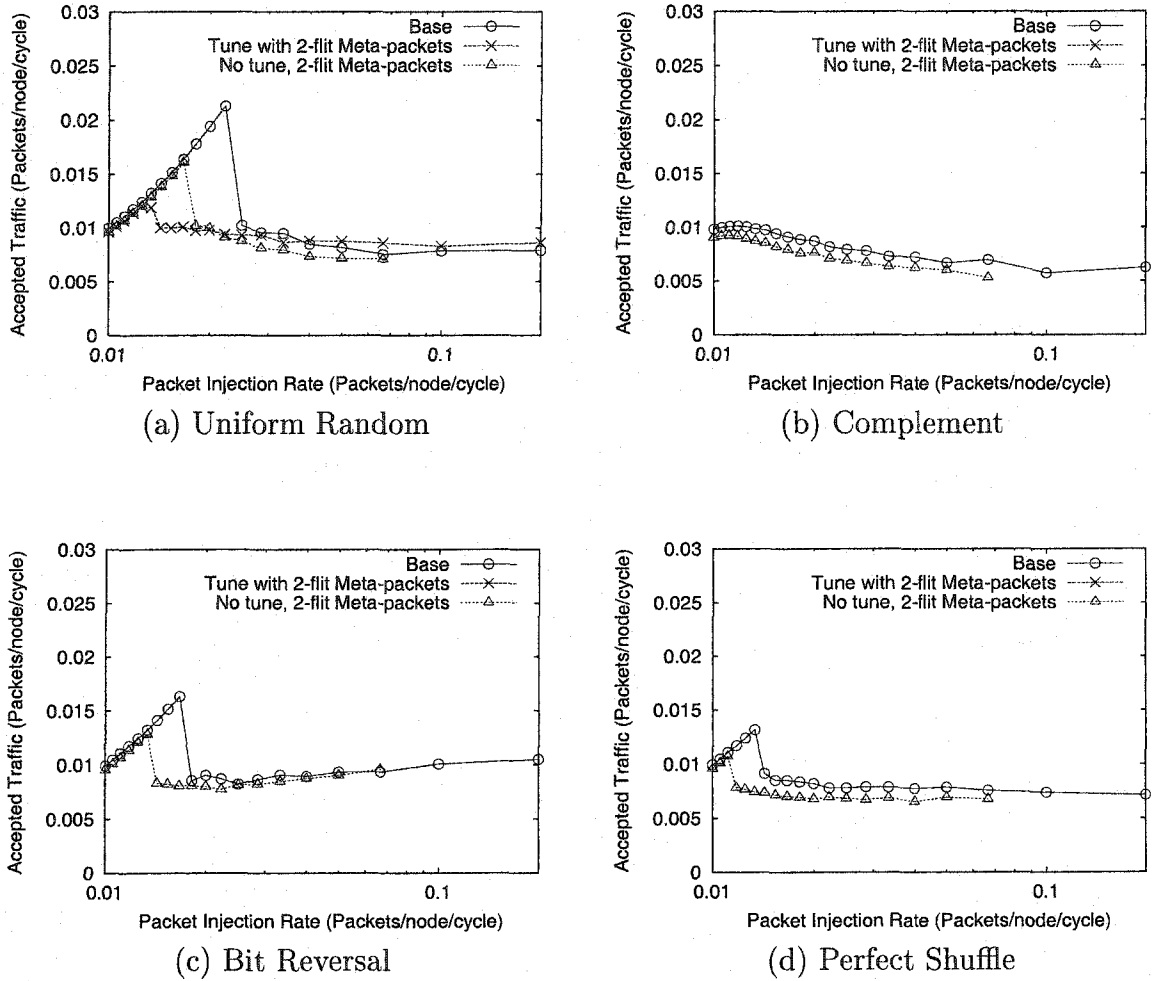
In this experiment, I measure gather times and study the effect of varying regular data-packet load on the gather times. Figure 3.15 plots maximum, minimum and average gather times using meta-packet against varying data-packet load for the deadlock recovery configuration. The X-axis shows the applied data-packet load in packets/load/cycle and the Y-axis shows the gather times in cycles. Results show that, on average, the gather time stays between 130 and 160 cycles. The maximum



**Figure 3.15:** Minimum, Maximum and Average gather times using meta packets

gather times do not exceed 300 cycles because of the higher priority accorded to the meta-packet virtual channel. Further, we observe that gather times drop beyond saturation loads. This is because data packets are blocked at saturation and thus meta-packets see no contention for physical links.

The next experimental result, shown in Figure 3.16, illustrates the effect of using the *Tune* congestion-control scheme with meta-packets as the information gathering mechanism for the four communication patterns. Apart from the base case and the *Tune* configuration using meta packets, I consider another case called *No tune*. The *No tune* configuration does gather the aggregate buffer occupancy and throughput



**Figure 3.16:** Effect of Meta-packets and Tuning on Throughput

information using meta-packets but the gathered information is not used to throttle network packets. This allows me to isolate the effect of contention caused by meta packets and the effect of using meta-packet information to throttle network packets. Figure 3.16 shows that distributing meta-packets causes saturation at lighter loads. Further, using the *Tune* mechanism further exacerbates the situation causing saturation at lighter loads. This is partly because of the larger gather delays which causes *Tune* mechanism to act on stale information. Recall, experiments with varying gather delay (Section 3.4.2) showed that *Tune* performance suffers at gather times greater

than 96 cycles and average gather times with meta packets vary between 130 and 160 cycles. In conclusion, using meta-packets as the information distribution mechanism is not suitable for the *Tune* self-tuned congestion control scheme.

### 3.5 Summary

Interconnection network saturation, and the commensurate decrease in performance, is a widely known problem in multiprocessor networks. Limiting packet injection when the network is near saturation is a form of congestion control that can prevent such severe performance degradation. Ideal congestion control implementations provide robust performance for all offered loads and do not require any manual tuning.

The highlight of this chapter is the development of a robust, self-tuned congestion control technique—*Tune*—for preventing performance degradation at network saturation. Two key components form the basis for the proposed design. First, I use global knowledge of buffer occupancy to estimate network congestion and control packet injection. When the number of full buffers exceeds a tunable *threshold*, packet injection is stopped. When congestion subsides, the full buffer count drops below the *threshold* and packet injection restarts.

The second piece of my solution is a self-tuning mechanism that observes delivered network throughput to automatically determine appropriate threshold values. Inappropriate thresholds can either over-throttle the network, unnecessarily limiting throughput, or under-throttle and not prevent saturation. A self-tuning mechanism is important since no single threshold value provides the best performance for all communication patterns.

Using simulation, I show that the *Tune* design prevents network saturation by limiting packet injection. The results also show that *Tune* is superior to an alternative implementation that uses local estimates of congestion because global information



can detect congestion in its early stages. I demonstrate that different communication patterns require different threshold values to prevent saturation without unnecessarily limiting performance, and that *Tune*'s self-tuning mechanism automatically adjusts to changes in communication patterns.

## Chapter 4:

### Load Balancing in the Minimum Rectangle

Load imbalances create pockets of congestion where packets suffer delays. The *tree saturation* phenomenon, as described in Chapter 1, can cause such pockets of congestion to spread rapidly and degrade network performance. In this chapter, I investigate the possibility of sustaining high throughput, low latency operation of the base minimal, adaptive router by balancing the load on the network to prevent the creation of such “hotspots”, thus preventing tree saturation.

The key innovation of my *congestion-aware via-routing* technique is the use of global congestion information—obtained using an oracle—to direct packets towards lightly loaded network regions. I assume the use of an oracle for obtaining global information for two reasons. First, the global information gathering scheme outlined in Chapter 3 is not sufficient for this scheme. This is because *congestion-aware via-routing* requires knowledge of individual buffer occupancies at the nodes rather than just the global aggregate. Second, the oracle assumption provides an upper bound on the performance improvement possible with realistic mechanisms to gather global information.

Routing decisions whose outcome depends on network status are called adaptive routing decisions. It is important to differentiate between *congestion aware via-routing* and *adaptive routing* since both techniques involve routing packets depending on network conditions. What is commonly referred to as adaptive routing may more accurately be described as *distributed adaptive routing* because adaptive routing decisions are made in a distributed manner at each hop. This flavor of adaptivity typically considers only locally visible network conditions in making adaptive

routing decisions.

The technique I propose in this section proposes the use of the global congestion information at the source. The source node uses its knowledge of global buffer occupancy to direct packets away from heavily loaded network regions and towards lightly loaded network regions. Since the adaptive (i.e., network status dependent) routing decision is being made centrally at the source node, this form of adaptivity is called *centralized adaptive routing*. My scheme uses this centralized adaptive routing mechanism (that uses global congestion information) on top of distributed adaptive routing (that uses local link status information).

Section 4.1 describes the mechanism of my centralized adaptive routing scheme. Sections 4.2 and 4.3 examine different load balancing policies. Performance evaluation of this scheme is presented in Section 4.4.

#### 4.1 Via Routing: The Mechanism

The centralized adaptive routing mechanism I examine is called *via routing* or *multi-phase routing* [19]. In this mechanism, each packet is routed towards certain intermediate nodes, or *via-nodes*, on its way to the final destination. The packet header carries additional state to indicate the intermediate nodes that it must route toward. The challenge is to design policies to pick the intermediate nodes for each packet at the time of injection, such that a packet routes away from highly loaded network regions and towards lightly loaded network regions.

My mechanism treats via-nodes as hints rather than requirements. This is necessary for deadlock-freedom given the other design parameters. This is because, the routing constraints on the deadlock-free escape paths are violated if they are used for intermediate nodes. To illustrate the violations with an example, consider the deadlock avoidance configuration where deadlock-free routing channel uses dimen-

sion ordered routing. This constraint requires that the packet has to exhaust all hops in the X dimension (say) before traversing deadlock-free channels in the Y dimension. However, packets that are routed on deadlock-free channels towards an *intermediate node* will violate this condition since they may be routed on the Y dimension deadlock-free channel even though they haven't exhausted all hops in the X dimension towards the *destination node*.

One way to make traversing via nodes obligatory and still maintain deadlock-freedom guarantees is to require that the packets have to be drained at each via node and re-injected to ensure deadlock freedom. Draining and re-injecting can increase latency because the packet will suffer injection queue delays again at the intermediate nodes. This increase in latency through the network can hurt overall system performance since packet latency lies on the communication critical path.

An alternative is to make sure that packets do not violate the conditions of deadlock-free routing towards the final destination even on paths towards intermediate nodes. This can reduce performance if a packet has deadlock-free paths available towards the final destination but is unnecessarily stalled because it is waiting for an adaptive path towards a via-node. There are other deadlock avoidance mechanisms where the packet can be forced to go to intermediate nodes without violating deadlock-freedom guarantees. But these typically require the use of a larger number of virtual channels per physical channel. A large number of virtual channels results in more complex arbitration and switch circuitry and that can adversely affect clock cycle time.

Due to the above mentioned constraints, I chose to treat via nodes as hints to try and route towards rather than required intermediate nodes that *must* be visited. In the following sections, I discuss policies of how to choose the via-nodes to achieve load balancing in the minimum-rectangle. Section 4.2 proposes a policy that broadly

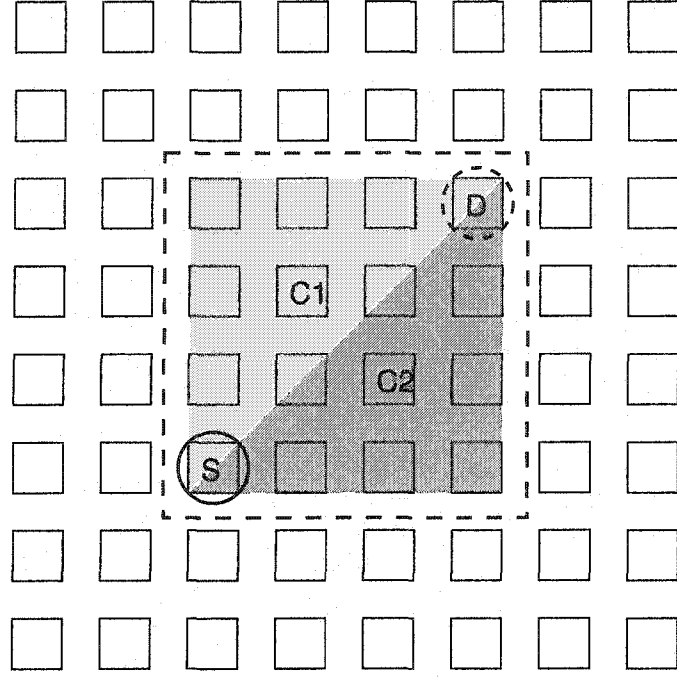
selects one half of the minimum rectangle as the preferred half to route packets through. Section 4.3 examines a technique to outline an entire “corridor” from the source node to the destination node such that the congestion a packet encounters in this corridor is minimized.

## 4.2 Min-triangle and Random-triangle Via Selection Policies

In a two-dimensional torus network ( $k$ -ary, 2-cube), the minimum rectangle between a source-destination pair is defined as the set of nodes and edges that can be reached by all possible minimal paths between two nodes. The dashed-rectangle in Figure 4.1 shows the minimal rectangle for nodes  $S$  and  $D$ .

Figure 4.1 also shows the minimum rectangle as being composed of an *upper* and *lower* triangle, with the primary diagonal forming the dividing line between the two triangles. I define primary diagonal as the diagonal that connects the source node to the destination node. The secondary diagonal is the other diagonal in the minimum rectangle. I define the “central” nodes of the upper and lower triangles as the nodes on the secondary diagonal of the minimum rectangle which are one quarter and three-quarters along the length of the secondary diagonal. In Figure 4.1,  $C1$  is the central node of the *upper triangle* and  $C2$  is the central node of the lower triangle.

I use the central nodes as via-nodes to try and achieve load balancing. I study the performance of the following two via-routing policies: the *rand-triangle* policy and the *min-triangle* policy. For any source-destination pair, the *rand-triangle* scheme randomly picks a central node in either the upper triangle or the lower triangle between those nodes. The *min-triangle* scheme makes an oracle assumption that each node knows instantaneously the number of full buffers in the upper and lower triangles. It uses this information to pick the central node of the triangle that has fewer full buffers.



**Figure 4.1:** Via Routing : Upper and Lower Triangles

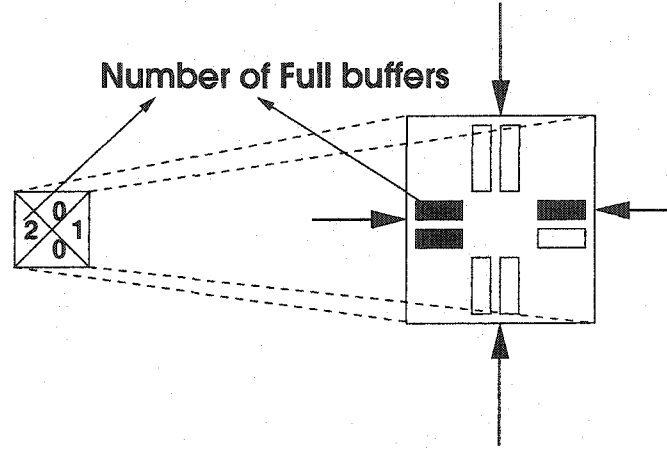
### 4.3 Min-Corridor Via Selection Policy

In this via selection, I assume perfect knowledge of buffer occupancies in all nodes in the network and select via nodes along the *Least Congested Path* in the minimum rectangle ( $LCP_{min}$ ) at the time of injection. I recast the problem of finding  $LCP_{min}$  for a given source-destination pair, to the problem of finding the shortest path ( $SP$ ) in a weighted, directed graph  $G = (V, E)$  where:

- $V$ , the set of vertices contains all nodes in minimum rectangle,
- $E$ , the set of edges is the set of all possible minimal hops a packet can make between the nodes in the minimum rectangle, and
- the edge weights for an edge from node A to node B is given by the number of full input buffers at node B on the physical input from node A.

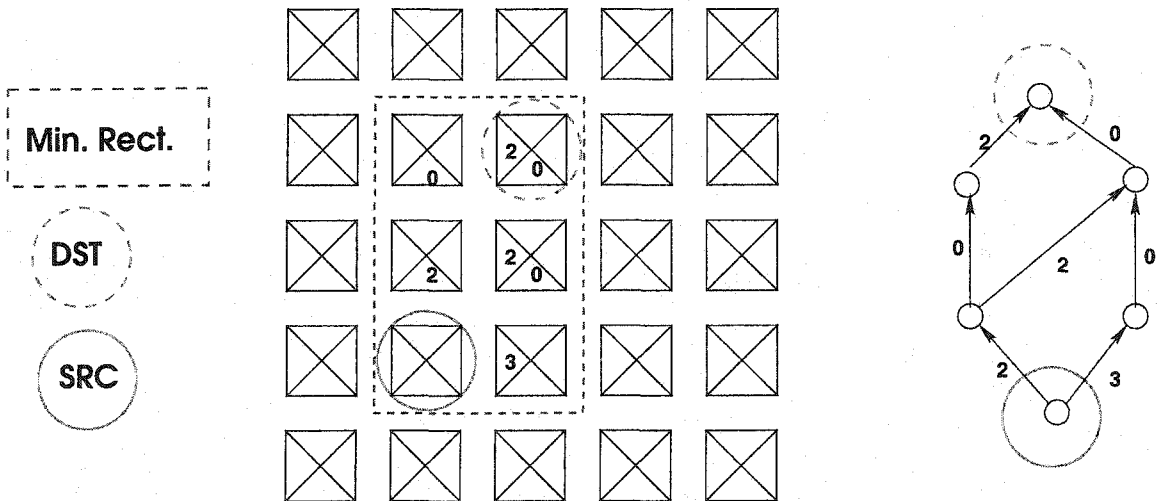
To illustrate this with an example, I use the notation in Figure 4.2 to represent the buffer occupancy at each node for a two-dimensional network. In the example shown in Figure 4.2, the diagram on the right shows the expanded view of a two-dimensional router with its four input ports and two input buffers associated with each physical input port. Full input buffers are shown as “filled” (black) rectangles and free input buffers are shown as outline (unfilled) rectangles. I represent the state of the router in an abbreviated manner as shown in the diagram on the left side of Figure 4.2. I use a square divided into four quarters to represent a router and its four input ports. The numbers in the four quarters represent the number of full input buffers along the physical each of the four network input ports.

I use this notation to demonstrate how the problem of finding  $LCP_{min}$  in a network (shown on the left Figure 4.3) can be recast as a *Shortest Path* problem in a weighted, directed graph (shown on the right in Figure 4.3). Figure 4.3 shows a source node, a destination node and the corresponding minimum rectangle. The buffer occupancies of the input ports on all possible paths between the source and the destination nodes are shown for all nodes in the minimum rectangle. Following the procedure for recasting the  $LCP_{min}$  problem to the *Shortest Path* problem, I generate a graph with six nodes (representing the six routers in the minimum rectangle) and edges corresponding to every possible minimal hop on the route between the source and destination. For example, a packet going from the source node to the destination node shown in Figure 4.3 can make an initial hop in one of two directions: (a) horizontally to the right, where the neighboring node has three input buffers full or (b) vertically upward, where the neighboring node has two input buffers full. These two possibilities translate to the two edges originating from the source with edge weights set to three and two respectively. Similarly, the graph for the whole minimal rectangle is constructed. The shortest path between source and destination nodes in



**Figure 4.2:** Representation of full input buffers

the graph can be mapped back to an equivalent least congestion path between the source and destination routers in the network. In my implementation, I use Dijkstra's algorithm [17] to compute the shortest path and use the nodes along the shortest path as via-nodes.



**Figure 4.3:** Recasting *LCPmin* as SP problem in directed graph

## 4.4 Via-Routing Results

Using the same methodology as described in Chapter 3, the various via-routing policies were simulated. Figs 4.4–4.7 shows the throughput and latency graphs for the



via-routing policies for the four traffic patterns. The important observations from the results are: (a) Via-routing alone does not prevent the performance degradation at saturation. All four traffic patterns see the performance degradation at higher loads. (b) Performance benefits of via-routing, in the form of increase in load at which saturation occurs, are seen only for the *bit reversal* pattern (Figure 4.5). For the other three communication patterns, there is either no change in saturation load (*uniform random* and *complement*) or even early saturation (*perfect shuffle*).

Since, load balancing in the minimum rectangle is not sufficient to sustain high-bandwidth, low latency network operation, I now examine combining my *Tune* congestion control system with via-routing. Figs 4.8–4.11 present simulation results combining the use of via-routing *and* the *Tune* congestion control mechanism for the four traffic patterns. From these results, we see that: (a) Peak throughput at saturation increases for three of the four (*uniform random*, *bit reversal* and *complement*) traffic patterns. (b) For perfect shuffle, while there is no throughput increase, there is no performance penalty either. This is vast improvement over the use of *via-routing* alone.

Note, the “spikes” seen in latency beyond saturation are not meaningful when the throughput is low. This phenomenon occurs when latency averages reflect latencies of packets that have left the network. The latency suffered by packets that do not reach the destination till the end of the simulation gets left out. Consequently, the latency behavior varies widely with changes in number of delivered packets. This phenomenon was also observed in Chapter 3.

Comparing the via-selection policies, we see that for both cases (i.e., via routing alone and via-routing with *Tune*) the performance of Min-Triangle and Rand-Triangle closely match. This means that randomization achieves the goal of balancing loads and that the use of global knowledge offers very little improvement over randomiza-

tion. Further,

From the above discussion, we arrive at three conclusions. First, the use of via-routing to achieve load balancing within the minimum rectangle does not give uniform benefits across communication patterns and is unable to prevent eventual performance degradation at saturation. Second, using via-routing with *Tune* congestion control does prevent performance degradation at saturation and for three out of four traffic patterns, it results in increased saturation load as well. Third, load balancing obtained by random via selection is hard to improve upon in any significant manner. Using perfect knowledge of congestion in the network to select via nodes adds little to no value over random via selection.

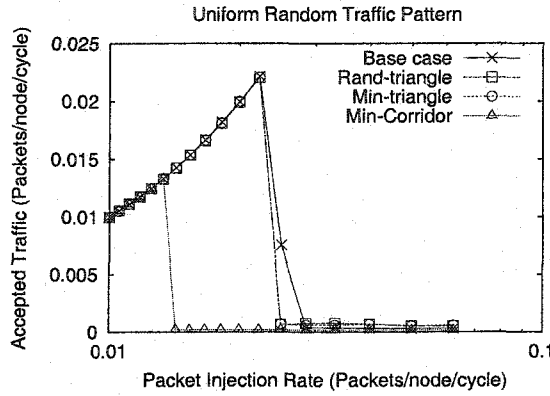
## 4.5 Summary

My *congestion-aware via-routing* scheme tries to exploit global knowledge of load imbalances to bias the routing of packets towards lightly loaded network regions within the minimum rectangle. In this section, I proposed and evaluated the mechanism (via-routing) and policies (Rand-triangle, Min-Triangle and Min-Corridor) to implement load balancing. My evaluations demonstrate that using via-routing on top of the *Tune* congestion control mechanism results in throughput improvement for three of the four communication patterns over the use of *Tune* alone. My results also show that assuming perfect knowledge of global buffer occupancies using an oracle assumption does not improve performance in any significant way over the use of randomized load balancing.

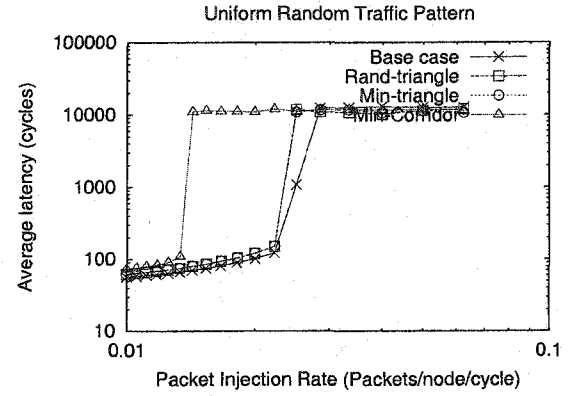
In both the *Tune* congestion control mechanism and load balancing using *via-routing*, I have restricted packets to minimal routing. However, previous research shows that there is scope to further improve performance and sustain high bandwidth, low latency operation beyond saturation loads if this constraint is relaxed [37]. Apart

from experimental results, theoretical limits [62] on performance within the minimum rectangle also lead to the conclusion that non-minimal routing is an interesting segment of the design space. In the next chapter, I explore the use of non-minimal routing techniques to achieve high bandwidth, low latency network operation at high loads.

## Deadlock Recovery

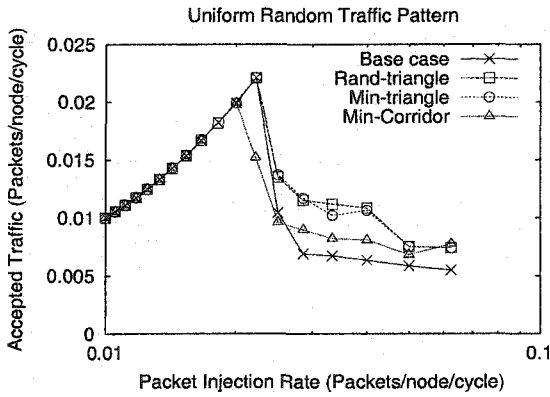


(a) Delivered Throughput vs. Offered Load

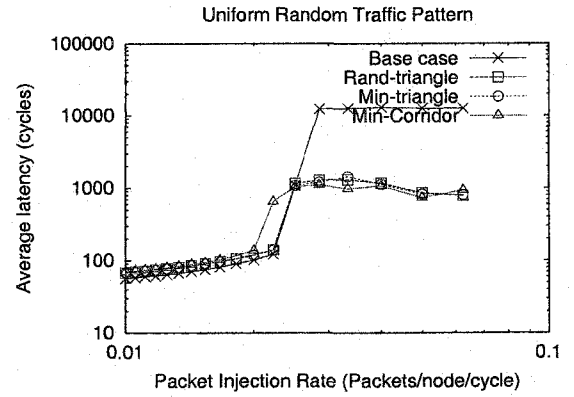


(b) Average Latency vs. Offered Load

## Deadlock Avoidance



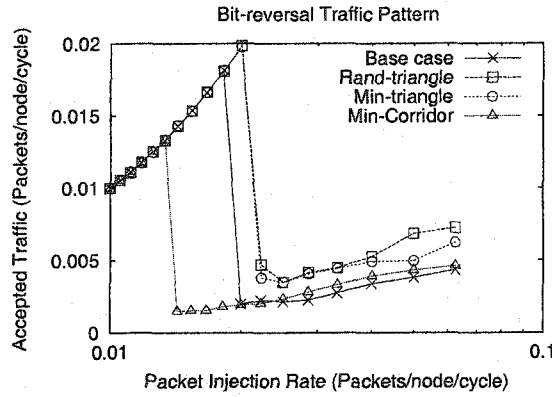
(c) Delivered Throughput vs. Offered Load



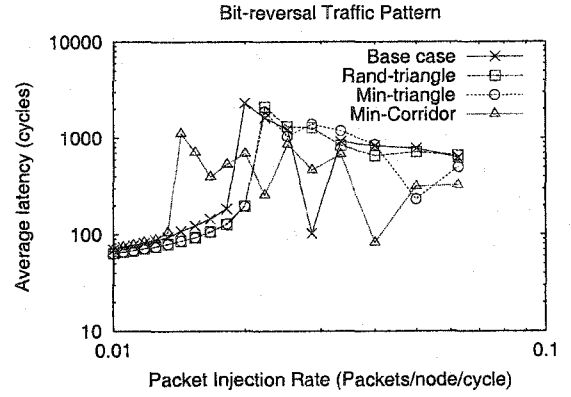
(d) Average Latency vs. Offered Load

Figure 4.4: Via-routing for Uniform Random pattern

## Deadlock Recovery

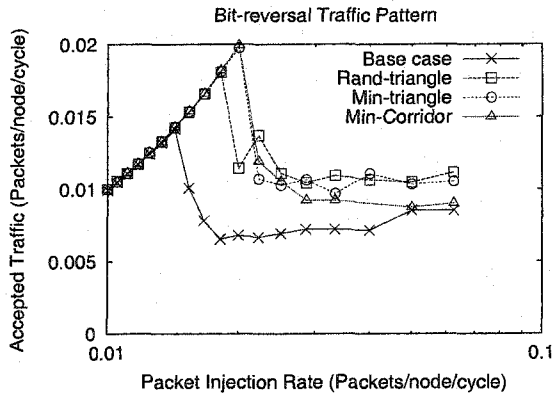


(a) Delivered Throughput vs. Offered Load

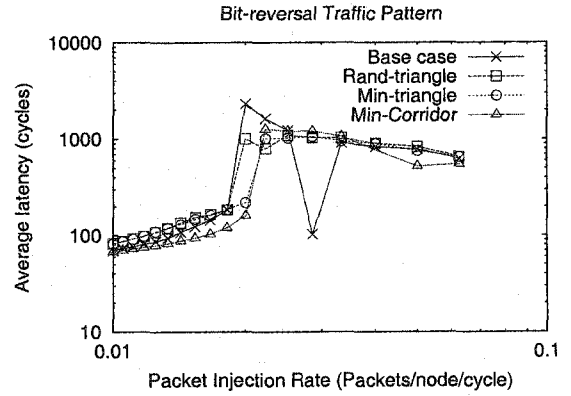


(b) Average Latency vs. Offered Load

## Deadlock Avoidance



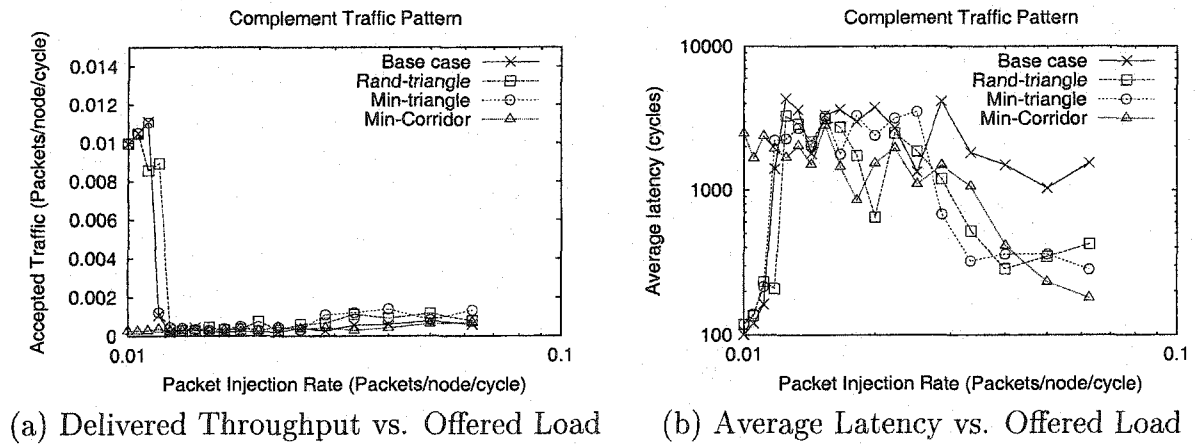
(c) Delivered Throughput vs. Offered Load



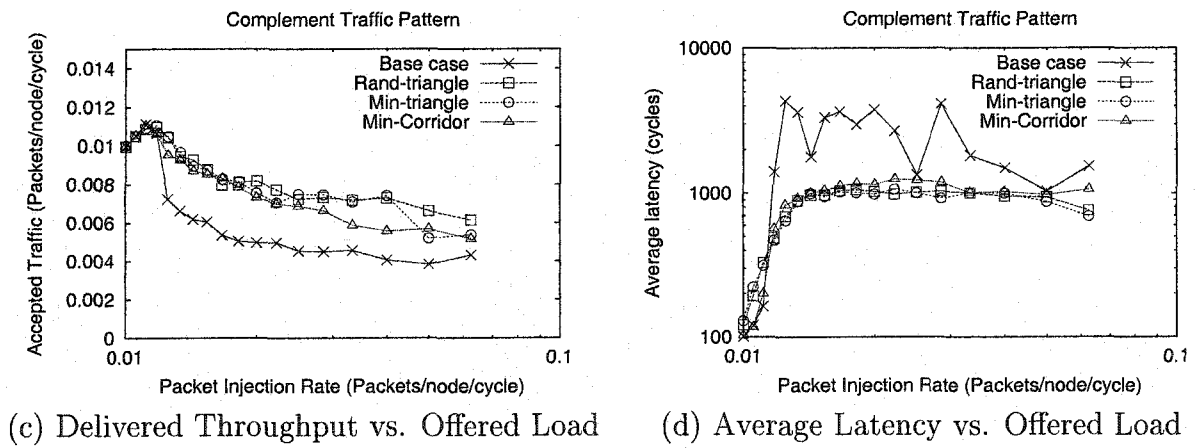
(d) Average Latency vs. Offered Load

**Figure 4.5:** Via-routing for Bit Reversal pattern

## Deadlock Recovery

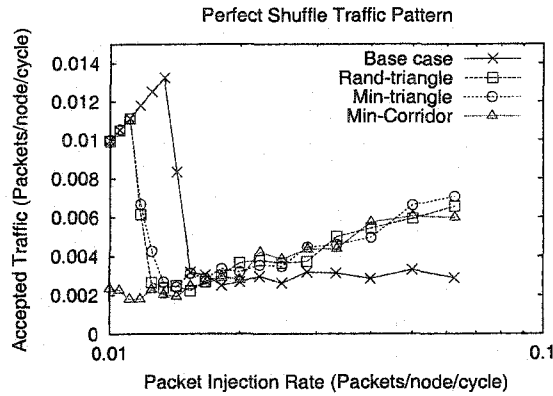


## Deadlock Avoidance

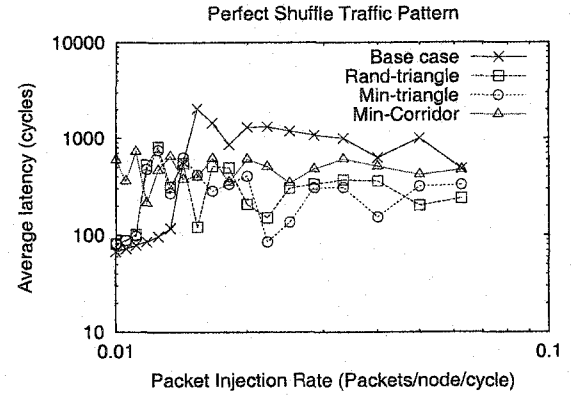


**Figure 4.6:** Via-routing for Complement pattern

## Deadlock Recovery

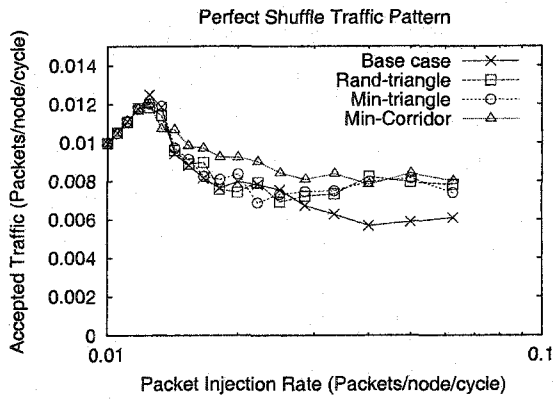


(a) Delivered Throughput vs. Offered Load

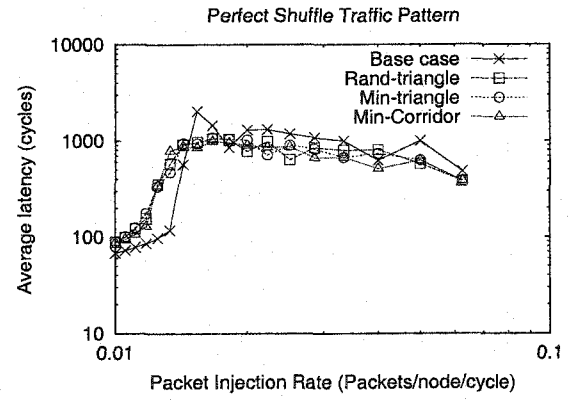


(b) Average Latency vs. Offered Load

## Deadlock Avoidance



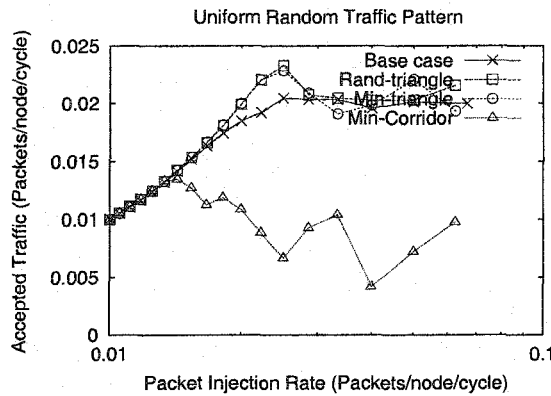
(c) Delivered Throughput vs. Offered Load



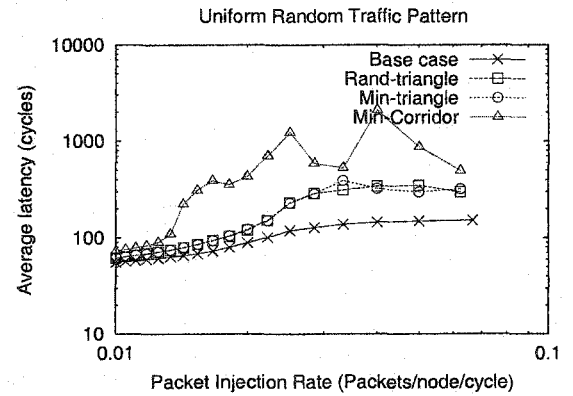
(d) Average Latency vs. Offered Load

**Figure 4.7: Via-routing for Perfect Shuffle pattern**

## Deadlock Recovery

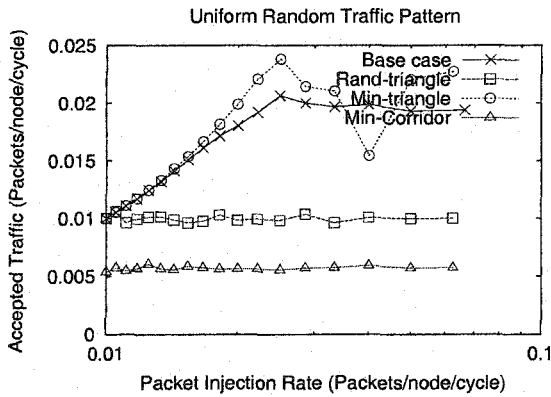


(a) Delivered Throughput vs. Offered Load

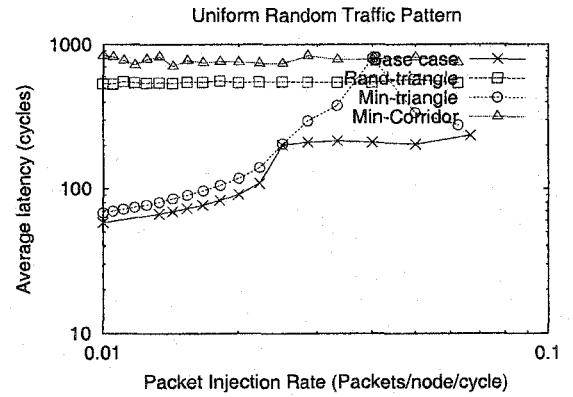


(b) Average Latency vs. Offered Load

## Deadlock Avoidance



(c) Delivered Throughput vs. Offered Load

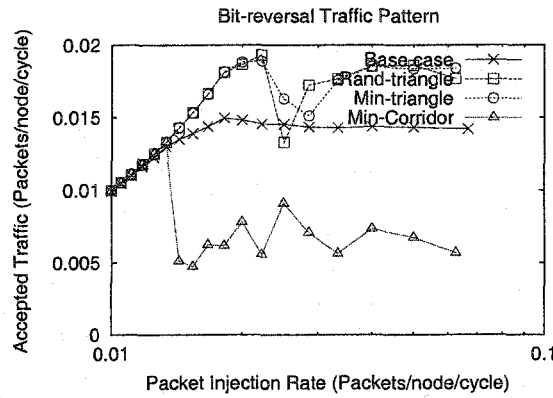


(d) Average Latency vs. Offered Load

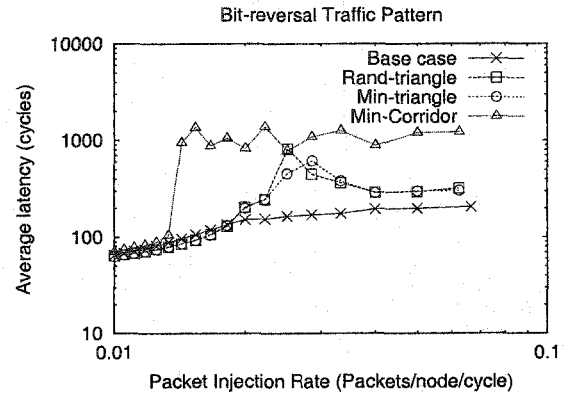
Figure 4.8: Via-routing with *TUNE* for Uniform Random pattern



## Deadlock Recovery

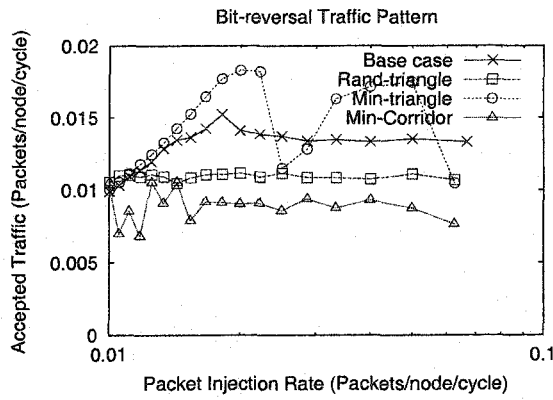


(a) Delivered Throughput vs. Offered Load

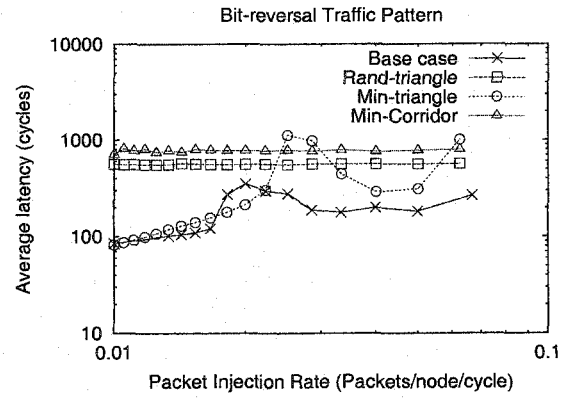


(b) Average Latency vs. Offered Load

## Deadlock Avoidance



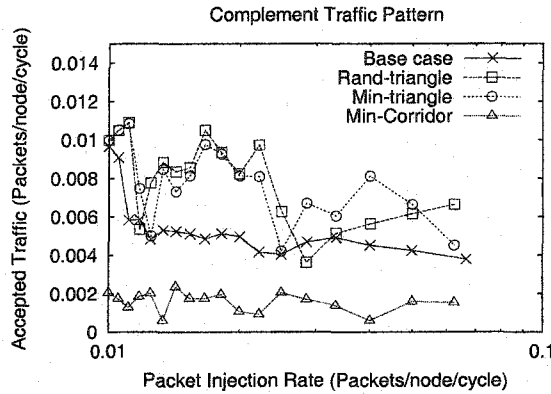
(c) Delivered Throughput vs. Offered Load



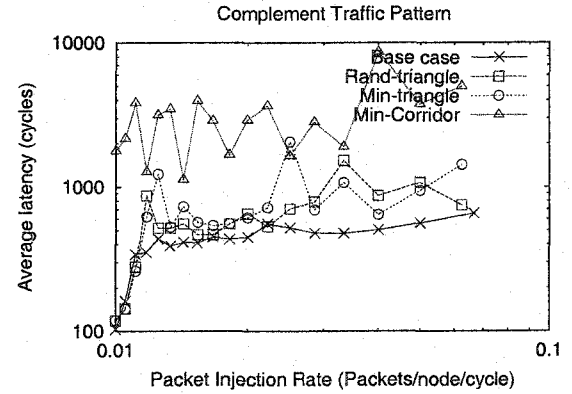
(d) Average Latency vs. Offered Load

Figure 4.9: Via-routing with *TUNE* for Bit Reversal pattern

## Deadlock Recovery

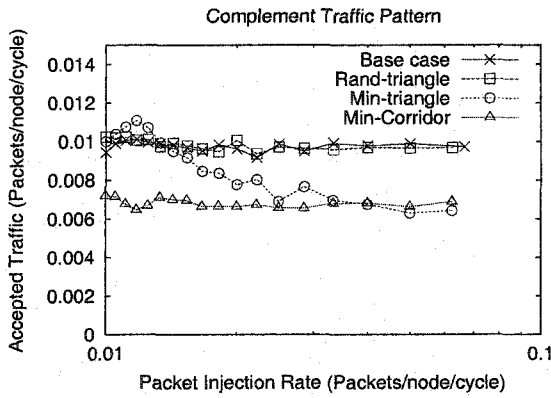


(a) Delivered Throughput vs. Offered Load

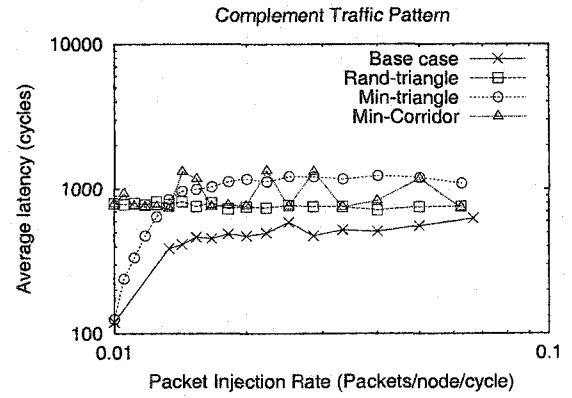


(b) Average Latency vs. Offered Load

## Deadlock Avoidance



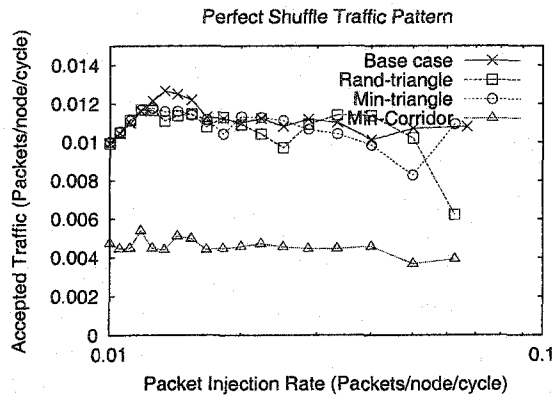
(c) Delivered Throughput vs. Offered Load



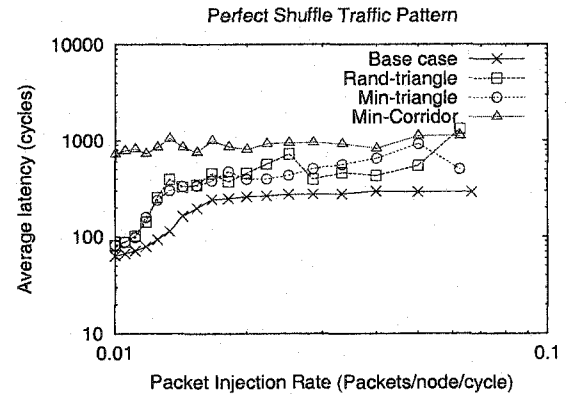
(d) Average Latency vs. Offered Load

**Figure 4.10:** Via-routing with *TUNE* for Complement pattern

## Deadlock Recovery

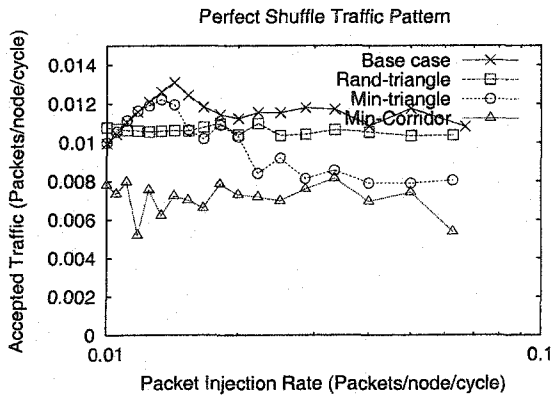


(a) Delivered Throughput vs. Offered Load

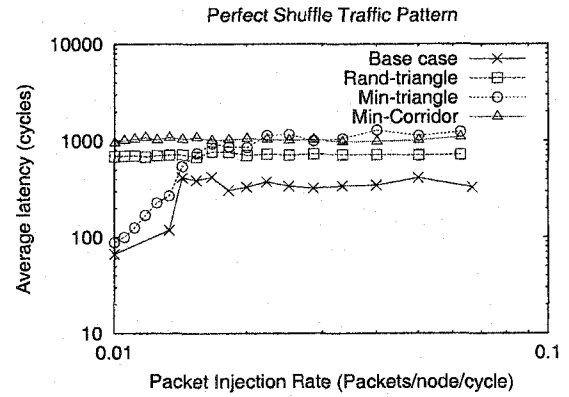


(b) Average Latency vs. Offered Load

## Deadlock Avoidance



(c) Delivered Throughput vs. Offered Load



(d) Average Latency vs. Offered Load

**Figure 4.11:** Via-routing with *TUNE* for Perfect Shuffle pattern

## Chapter 5:

### BLAM routing

In the previous two chapters, I described techniques to sustain high bandwidth, low latency network operation at high loads that ensure that packets are routed on minimal paths, i.e., packets never take a hop that takes them farther from the destination. The first was a congestion control technique (*Tune*) and the second was a load balancing technique (*via-routing*).

In this chapter, I will describe BLAM—a non-minimal adaptive routing algorithm<sup>1</sup> that achieves high bandwidth and low latencies at high offered loads.

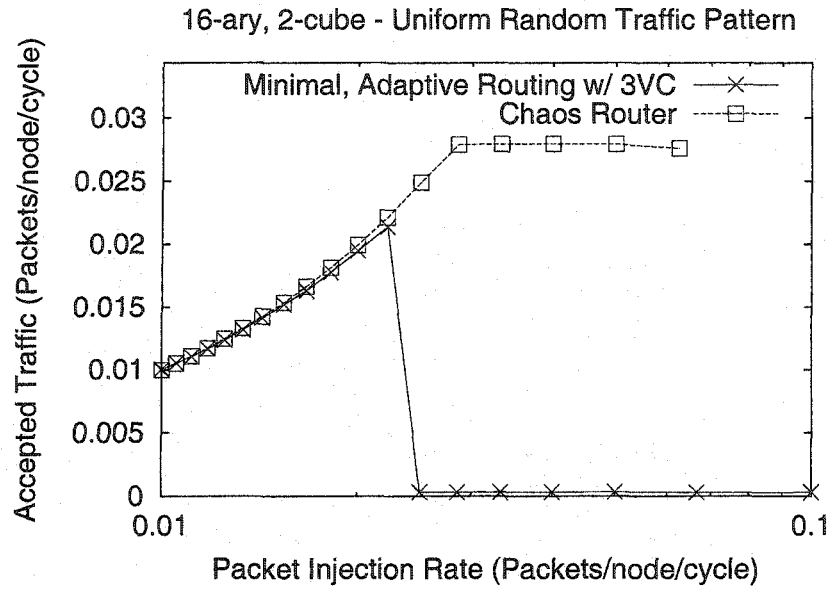
Non-minimal routing algorithms may use different deadlock handling approaches that change their livelock-freedom guarantees and performance. Section 5.1 compares and contrasts the livelock, deadlock and performance characteristics of non-minimal routing algorithms and minimal routing algorithms and describes a point (BLAM) in the router design space. Section 5.2 and Section 5.3 elaborates on this design by look the two key components of the BLAM routing algorithm: misroutes and bypass buffers. Section 5.4 and Section 5.5 present the experimental methodology and simulation results, respectively. Finally, Section 5.6 summarizes and concludes this chapter.

#### 5.1 Deadlocks, Livelocks and Performance in Nonminimal Adaptive Routing

Adaptive routing algorithms can be deadlock-prone in networks, such as k-ary n-cube networks, that allow packets to create a cyclic dependence. In previous chapters, I concentrated on one approach to handle deadlocks in adaptive virtual channels. This

---

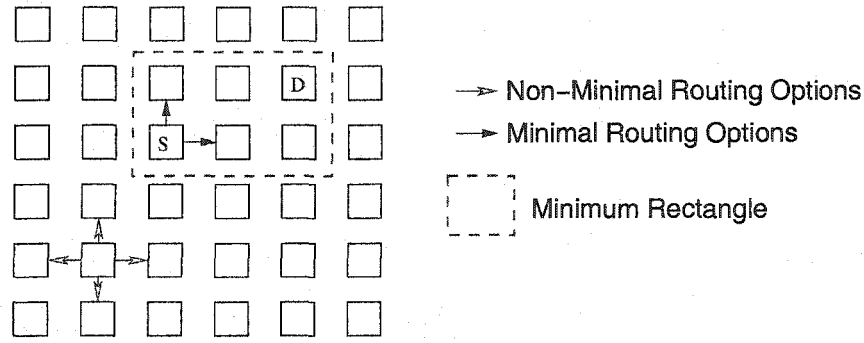
<sup>1</sup>A routing algorithm that allows packets to take hops that take them farther from the destination



**Figure 5.1:** Minimal Adaptive Routing w/deadlock recovery and Chaotic Routing approach guarantees the presence of forward escape paths for packets that deadlock in the adaptive channels and examples of this approach include deadlock avoidance [18] and deadlock recovery [38]. These solutions can be thought of as consisting of two logical networks: one fully adaptive network and another deadlock-free network. Deadlock is not possible because stalled packets can always make forward progress on the deadlock-free network.

A different approach to handling deadlocks in non-minimal routing algorithms such as Chaotic routing, rely on the deflection principle and the *packet exchange protocol* (see Chapter 2) to avoid deadlocks. This approach guarantees deadlock avoidance by making sure that packets are never blocked indefinitely, but it is possible that packet hops may be taking them farther from the destination.

Adaptive routing algorithms can be livelock-prone if the routing algorithm does not guarantee delivery of a packet from source to destination within a finite number of hops. Livelock can never occur in minimal adaptive routing, (e.g., Alpha 21364 [43]), as packets always reach their destination within a finite number of hops because every



**Figure 5.2:** Routing Options Minimal vs. Nonminimal

hop take a packet closer to its destination. In contrast, non-minimal adaptive routing, such as the Chaos routing algorithm [37], is not provably livelock-free, because it allows packets to be “misrouted” outside the minimal rectangle at every hop. That is, it allows hops that takes a packet farther from its destination and, hence, does not guarantee delivery within a finite number of hops.

Interestingly, however, the Chaos routing algorithm performs significantly better than a minimal adaptive routing algorithm (Figure 5.1) at high offered loads. This is because the Chaos routing algorithm offers greater routing freedom compared to a minimal adaptive routing algorithm (Figure 1.5.) The minimal adaptive routing algorithm I simulated (Figure 5.1) saturates and, thereby, causes the performance to degrade rapidly beyond a certain load.

Unfortunately, in spite of its high performance, to the best of my knowledge no commercially available interconnection network uses the Chaos routing algorithm, even though it has been over a decade since the design was proposed. The presence of livelocks—however low its probability may be—causes network designers to shy away from using such algorithms in real products. The challenge is to develop a solution that has the benefits of each of the two routing algorithms (minimal adaptive and Chaos) without either technique’s pitfalls.

In this chapter, I propose a new adaptive routing algorithm called BLAM that

achieves Chaos-like performance without livelocks or deadlocks. BLAM has four salient features. First, like Chaos, BLAM allows packets to be misrouted outside the minimal rectangle, thereby giving packets greater routing freedom. Second, to avoid livelocks, BLAM limits the number of times a packet is misrouted to a predefined threshold. Third, BLAM uses “lazy” misrouting in which packets are misrouted only after they fail consistently, over a period of time, to route within the minimal rectangle. Finally, BLAM uses “bypass buffers” at input ports to make sure that packets that fail to route profitably do not block the paths of other packets that follow.

In the following sections, I examine the issues Section 5.2 discusses misroutes and Section 5.3 discusses how I use bypass buffers to implement lazy misrouting.

## 5.2 Misroutes

Minimal adaptive routers do not allow misroutes by definition because this eliminates the possibility of livelock. A packet is guaranteed to move closer to the destination in each hop. Minimal routing, combined with deadlock-freedom, guarantees that a packet will be delivered to its destination. Another disincentive for the use of misroutes is that they may waste network bandwidth since packets move farther from their destinations.

However, there are three motivations to use misroutes. First, misroutes can be used to avoid deadlocks in the adaptive channels. Chaos uses misroutes, in addition to the packet exchange protocol, to avoid deadlocks. Chaos can, thereby, avoid the use of a separate, logical deadlock-free network. Second, by allowing non-minimal routing, they can provide fault tolerance by routing around faulty links. Third, again, by non-minimal routing, misrouting can provide higher network throughput by routing around congested areas in the network.

To use misroutes, policies that answer the following questions must be in place.

- *Should there be a limit on the number of misroutes?* Unlimited misroutes are fundamental to ensure deadlock-freedom in chaotic routing, but this results in probabilistic (i.e., not deterministic) livelock-freedom. Since the goal is to have deterministic guarantees of livelock-freedom, I place a limit on the number of misroutes a packet may take. The decision to limit misroutes removes the guarantee of deadlock-freedom on the adaptive channels. This necessitates inclusion of a deadlock handling mechanism.

- *What should the misroute limit be?* It is difficult (or even impossible) to recommend a single number for this limit without any information about the workload. Instead, I examine the tradeoffs involved if the limit is too high or too low. The idea is to set the limit high enough to ensure that most packets get delivered before they use all their misroutes. This reduces the latency of packet delivery from the source to the destination. In networks that have low-bandwidth deadlock-free paths, a high enough limit also ensures that these paths do not get unduly congested.

- *When is a packet misrouted?* *Eager* misrouting is a policy that lets packets misroute on a free channel if they cannot obtain a free profitable channel. *Lazy* misrouting policies impose some other condition that delays using misroutes.

In general, the choice of when packets are misrouted depends on the motivation for misroutes. When the purpose for misroutes is fault tolerance, an *eager* misrouting strategy may be sufficient. (If the only profitable channels for a given packet are known to be faulty, there is no point in delaying the misroute.)

The purpose is to achieve high performance and to avoid deadlocks in adaptive channels. Anjan and Pinkston [25] have shown that eager misrouting can hurt performance for uniform traffic. I confirm this result in Section 5.5.4. As such, a *lazy* misrouting strategy that postpones misrouting until it is either hurting performance



because of packets stalling behind it or because misrouting is mandated by the packet exchange protocol to avoid deadlocks in adaptive channels is preferable. This lazy misrouting strategy minimizes wasted network bandwidth. Note, Chaos also uses lazy misrouting.

With a *lazy* misrouting strategy, it is important to ensure that a blocked packet waiting to be misrouted does not block other packets that could otherwise make forward progress. I achieve this by using bypass buffers. Experiments show that lazy misrouting without bypass buffers decreases performance (Section 5.5.4).

### 5.3 Bypass buffers

I use bypass buffers to facilitate lazy misrouting. A bypass buffer allows a blocked packet to “step aside” from the critical path of other packets by buffering the blocked packet and releasing the input buffer. Once a packet enters a bypass buffer, packets behind it can use the free input buffer and bypass the blocked packet if they find profitable channels. Packets resident in these bypass buffers are candidates for lazy misrouting.

Bypass buffers have a secondary effect of increasing the total amount of buffer space at an input port. This may help improve performance. However, my simulations show, if the number of input buffers are chosen appropriately (perhaps using Little’s Law), then additional buffering provides no or marginal improvement in performance. I demonstrate this effect in Section 5.5.3 by adding bypass buffers (but with no misrouting) to a minimal adaptive routing algorithm.

Below, I examine the policies needed to manage bypass buffers for lazy misrouting.

- *When do packets enter the bypass buffers?* Packets in input buffers that are unable to make progress on profitable channels move to bypass buffers when they have waited “sufficiently long” or for implementing the packet exchange protocol.

The Chaos router considers a packet to have waited sufficiently long at a node if the whole packet (including the tail flit) has arrived at that node. I use the same definition. (The packet cannot be moved before all flits arrive. Waiting in the input buffer after arrival of the full packet can only delay the

The packet exchange protocol dictates that if a node sends a packet to a neighboring node, it should also be prepared to accept a packet from that node. To this end, when a packet is sent out on an output channel, an input channel in the reverse direction (on the same physical link) should be made free in anticipation of an incoming packet. To do so, one packet that is in the input buffer is moved to a bypass buffer.

- *Should packets in the bypass buffers have priority?* If both the bypass buffer and its corresponding input buffer have packets to nominate to a particular virtual channel, then the routing algorithm must decide which packet to pick. One policy is to use a fair-mechanism like round robin among all input and bypass buffers. Another option is to give priority to packets in the bypass buffers, since these packets are older. Routing older packets first is a good heuristic to achieve better performance. The *Rotary Rule* mode of the Alpha 21364 network uses a similar heuristic to assign higher priority to packets arriving from a network link than to new packets trying to enter the network [43].

Priorities for broad classes of packets (e.g., priority packets in bypass buffers over packets in input buffers, packets arriving from network link over packets arriving from node, coherence replies over requests, etc.) can be implemented in simple and scalable ways. However, router-wide priorities (e.g., priority for “oldest” packet) are not only more complex designs, they also require central structures that can become clock-scaling bottlenecks.

Note, giving priority to packets in input buffers over packets in bypass buffers is

not a valid design point if it comes without additional safeguards to prevent starvation of packets in the bypass buffers.

### 5.3.1 Implementing Distributed Bypass Buffers

In this section, I describe an implementation of bypass buffers using distributed buffers. First, I examine the centralized implementation used in *chaos* and then suggest my own. Note, I assume that additional bypass buffers are required. The same effects may be achieved by managing input buffers differently. For example, the Alpha 21364 router allows packets to be routed in non-FIFO order [43]. A *local arbiter* considers all the packets at an input port and nominates packets in *least-recently selected* order to achieve the effect of bypassing.

#### The Chaos Router Multiqueue

The original *Chaos* router design for two dimensional networks augments a basic router with an additional central *multiqueue* to provide a central pool of bypass buffers (see Figure 5.3). This central *multiqueue* requires additional routing logic and a cross bar on the input side of the queue. Note, Figure 5.3 shows only the datapath of the two-dimensional chaotic router. The routing and arbitration units for the two crossbars are omitted. The *Chaos* router does not use multiple virtual channels per physical channel. As such, the input buffers (or frames) are associated with the physical channel. The four network physical channels that connect to neighboring nodes are marked with labels that indicate the dimension (X or Y) and direction (positive or negative) they traverse. Each header contains the number of hops required in each dimension to reach the destination. This header information is modified (incremented or decremented) appropriately depending on its output channel.

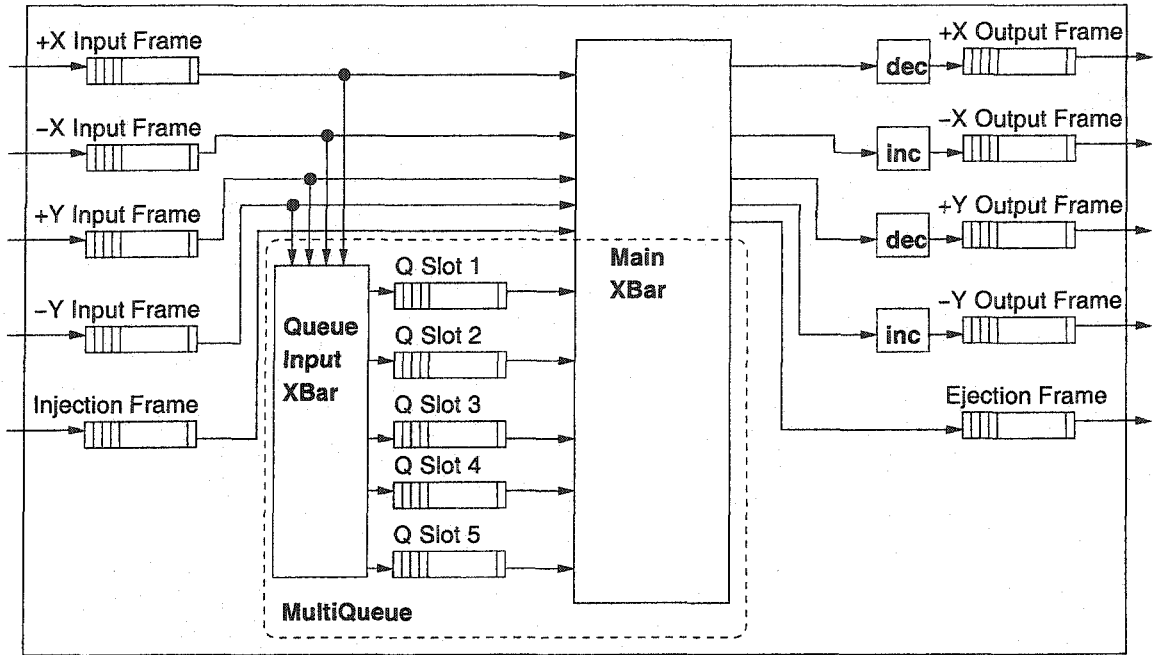


Figure 5.3: Central Bypass Buffers: Chaos

### Distributed Bypass Buffers BLAM

For my BLAM implementation (Figure 5.4), I use distributed bypass buffers rather than the centralized pool approach of the *Chaos* router. Previous research [46, 59] has recommended centralized buffer-pools over distributed buffers arguing that dynamic sharing of the central buffer pool leads to more efficient use of buffers. However, buffer efficiency is not a critical concern when we consider on-chip routers (such as the Alpha 21364 router [43]) where additional buffers are cheap. Clock scaling is more important design concern for high-speed routers and central structures (queues, implementation of router-wide priorities, etc.) are unsuitable as they become bottlenecks for clock scaling. Further, with enough buffers, a shared buffer pool and distributed buffers should be similar in performance. Consequently, routers with distributed buffers are attractive design points [43] and distributed bypass buffers are a natural fit for such designs.

Figure 5.4 shows the datapath of the BLAM router. Since BLAM permits a finite

number of misroutes for each packet, the header maintains a count of the number of misroutes taken. This is incremented (in the *Inc/Nop* block) each time the packet is misrouted. No change is required for profitable hops. There is an additional bypass-buffer associated with each input buffer belonging to an adaptive virtual channel. Bypass buffers are not associated with injection channels. Note, Figure 5.4 assumes fully adaptive routing with the deadlock recovery scheme. Therefore, all virtual channels are adaptive channels with associated bypass buffers. If deadlock avoidance is used, there are no bypass buffers associated with the deadlock-free channels.

The operation of bypass buffers is similar to the chaotic router implementation in some respects. Packets move from the input buffers to the bypass buffers when the whole packet has arrived at the node or when such a transfer is necessary due to the packet exchange protocol. In the common case, when packets are making forward progress, the bypass buffers are not on the critical path and packets go directly from the input buffers to the output buffers. Packets in the bypass buffers have priority over packets in the input buffers when they compete for the same output channel. A packet in a bypass buffer is misrouted when the corresponding input buffer entry wants to enter the bypass buffer (either due to stalling or due to the packet exchange mechanism.)

Unlike Chaos, BLAM's one-to-one correspondence between adaptive virtual channel input buffers and bypass buffers eliminates the need for the queue input cross bar and associated routing logic. However, this approach removes the element of randomization present in chaotic routing. In chaotic routing, when a packet needs to enter the "multiqueue" (either due to stalling or due to the packet exchange protocol) and the "multiqueue" is full, one entry is selected at random to be misrouted. This is fundamental to the probabilistic guarantees of livelock-freedom. Since my BLAM implementation allows a packet to move to only one possible bypass-buffer,

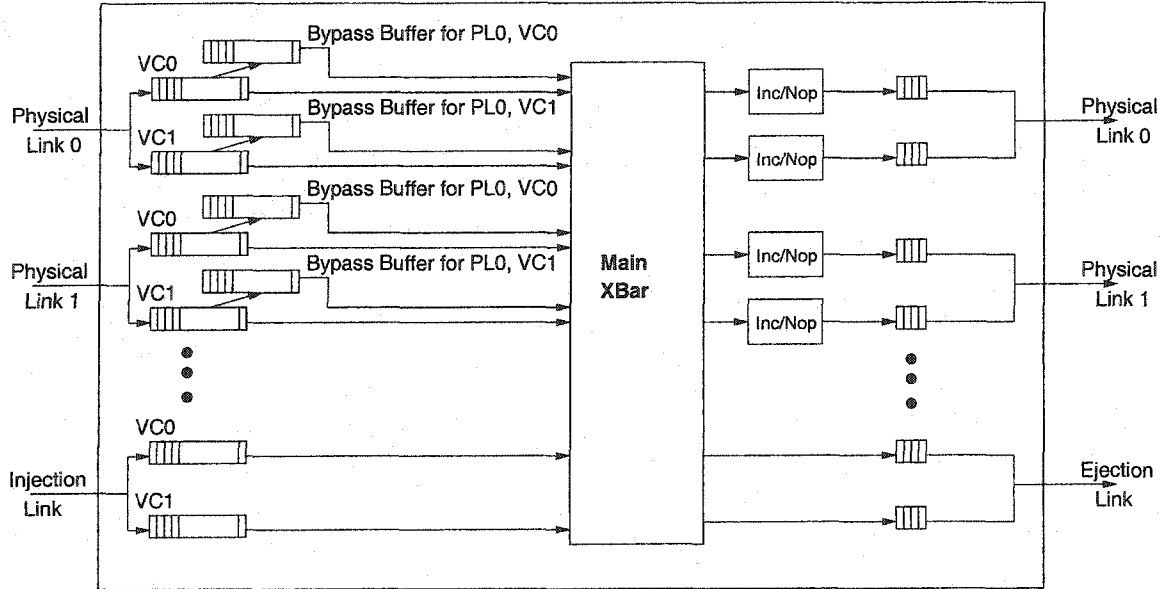


Figure 5.4: Distributed Bypass Buffers

the packet in that bypass buffer must be selected for misrouting and there is no scope for randomization. However, my design provides deterministic guarantees of livelock-freedom without using randomization because of the limit on the number of misroutes.

### 5.3.2 Summary

From the above discussion, we see that there exists a potential design point between a minimal adaptive router and the chaos router. This router uses Bypass buffers, with Limited, Adaptive, lazy Misroutes and deadlock handling (BLAM). Limited misroutes gives BLAM three advantages: livelock-freedom (compared to chaos), more routing flexibility and reduced frequency of use of deadlock-free escape paths (compared to the base router). Lazy misrouting with bypassing is an important feature of BLAM that enables chaos-like high performance.

Table 5.1 summarizes the properties of BLAM and compares it against a  $M$ -

Routing Scheme	Misroutes	Bypass buffers	Deadlock Handling	Comments
$M$ -misroute, adaptive	$M$ ( $M = 0$ for minimal)	No	Required	Deadlocks possible in adaptive channels. Deadlock handling required. Guaranteed livelock-free
Chaos	Unlimited	Yes	Not Required	Guaranteed deadlock-free. Probabilistic livelock-freedom.
BLAM	$M$	Yes	Required	Deadlocks possible in adaptive channels. Deadlock handling required. Minimize deadlock handling use by increasing $M$ . Guaranteed livelock-free

**Table 5.1:** Design variables for various routing schemes

misroute, adaptive router<sup>2</sup> and the chaos router.

## 5.4 Evaluation Methodology

I describe the simulation details in Section 5.4.1. The simulators I use are not full system simulators that model the entire cache-coherent, distributed shared memory multiprocessor (DSMP) system. Instead, I simulate only the interconnection network with synthetically generated traffic. Section 5.4.2 discusses how my simulation parameters model realistic DSMPs. Section 5.4.3 describes the nature of the synthetic traffic I use to evaluate BLAM. The network and router architecture that my simulator models is outlined in Section 5.4.4.

### 5.4.1 Simulation Details

To evaluate the various routing schemes I use the `flexsim` simulator [60] and the chaos simulator available from the University of Washington [8]. The Chaos router is simulated on the chaos simulator. I use the `flexsim` simulator for all other con-

---

<sup>2</sup>An  $M$ -misroute adaptive router is an adaptive router that allows each packet to take atmost  $M$  misroutes. Note, a minimal adaptive router is an  $M$ -misroute adaptive router with  $M = 0$ .

figurations. All simulations execute for 60,000 cycles. However, I ignore the first 10,000 cycles to eliminate warm-up transients. I have verified, for a subset of experiments, that longer simulations of 300,000 cycles do not change the results significantly. (Bandwidth is within 2% and latency is within 4% of the results from a 60,000 cycle simulation.) I use an extension of the *Chaos Normal Form (CNF)* [7] standard for presenting simulation results. I present two graphs (throughput vs. applied load and latency vs. applied load) for each configuration. The CNF standard requires throughput and load to be normalized to its definition of 100% throughput for uniform random traffic. However, I use a modification that uses additional lines on the graph to show normalized throughput (as defined by CNF) and I calibrate the axes to provide absolute (i.e. not normalized) throughput and load values in terms of packet injection/delivery rate.

#### 5.4.2 Open vs. Closed Loop

The simulators I use model open-loop systems with independent messages and a source queue size of 1024 packets. In real closed loop systems, there are request-response dependencies and this results in inherent throttling when network latencies increase. Consider, a program suffers a remote miss in a DSMP. It sends a request for the required cache block. In the absence of latency tolerance mechanisms, the program has to stall till the cache block is supplied in a response packet. However, out-of-order lockup-free execution, aggressive data-speculation, multiple contexts per node (due to technologies like SMT and CMP) result in significant latency tolerance. For example, current technologies with 8-way multithreading and 8 outstanding misses per thread (due to out-of-order lockup-free ILP exploitation) can already generate 64 outstanding requests. Further, we have to consider the requests received at a node that result in response packets that need to be sent back. If we make the simplifying



assumption that each node receives as many requests as it sends out, the number of outstanding packets stand at 128. Thus, the use of a source queue size of 1024 packets, while aggressive, is appropriate if we extrapolate the technology trends.

The extension of a system with independent messages to a system with request-response dependencies can be achieved by splitting request and response virtual channels.

### 5.4.3 Network Workload

The offered load consists of each node generating 16-flit packets at the same fixed rate for the duration of the simulation. The goal is to show that the simulation results are valid for a variety of loads. Ideally, we would like to measure interconnect performance by using real communication workloads from parallel applications. Unfortunately, due to inadequate simulation infrastructure and problems associated with trace-based simulation [10], it is the state-of-the-practice to evaluate interconnection network performance with synthetic communication workloads.

The use of synthetic communication patterns that are “difficult” (i.e., they increase contention for resources, resulting in sub-optimal/worst-case performance [62]) and/or “useful” (i.e., they correspond to the communication pattern for various parallel numerical algorithms [20]) to evaluate interconnection network performance is widespread in the literature [4, 27, 36, 40, 41, 61, 49]. Towles and Dally [62] demonstrate a technique to construct a synthetic traffic pattern that results in the worst-case performance for oblivious routers, but this technique is not applicable to adaptive routers.

Apart from the widely used, *uniform random* traffic pattern, we consider three synthetic communication patterns, *bit-reversal*, *perfect-shuffle* and *complement* to stress the network in non-uniform ways. The communication patterns differ in

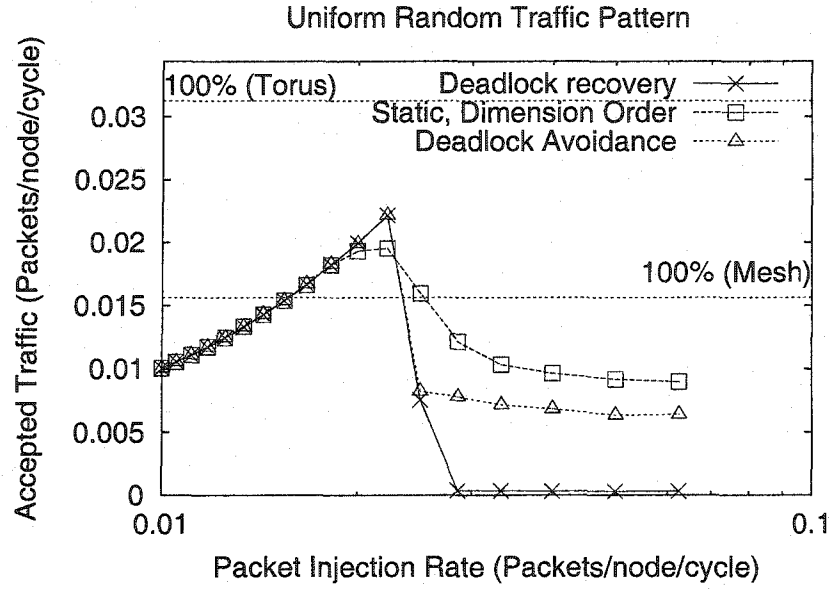


Figure 5.5: Choice of Base configuration

the way a destination node is chosen for a given source node with bit co-ordinates  $(a_{n-1}, a_{n-2}, \dots, a_1, a_0)$ . The bit co-ordinates for the destination nodes are  $(a_{n-2}, a_{n-3}, \dots, a_0, a_{n-1})$  for *perfect shuffle*,  $(a_{n-1}, a_{n-2}, \dots, a_1, a_0)$  for *complement* and  $(a_0, a_1, \dots, a_{n-2}, a_{n-1})$  for *bit-reversal*.

#### 5.4.4 Network and Router Architectures

I use a minimal, fully-adaptive router (Figure 2.1) as the base configuration. I evaluate the base (minimal, adaptive) and BLAM network configurations with both the Disha [38] progressive deadlock recovery scheme with a time-out of 25 cycles and deadlock avoidance [18]. (Note, Chaos does not need either of these two schemes.)

I use these configurations to maximize performance in the common case (i.e. before saturation). Deterministic deadlock-free routing (□ in Figure 5.5) places restrictions on routing which results in lower performance and earlier saturation. Turn model based schemes offer partial adaptivity and better performance in meshes, but these schemes are not enough to prevent deadlocks in k-ary, n-cubes because the

wrap-around edges in the torus enable deadlock cycle formation without taking any turns in the topology. The maximum throughput achievable using the turn model in a 16x16 mesh (the *100% (mesh)* line in Figure 5.5) is poorer than the throughput achieved in a 16-ary, 2-cube (torus) using a static dimension-ordered routing. This is because the addition of the wrap-around edges alters the network properties significantly enough to double the theoretical maximum throughput [5]. As such, the excellent performance demonstrated by various flavors of the turn model are not applicable in the context of  $k$ -ary,  $n$ -cubes [30, 63].

I have evaluated the following  $k$ -ary,  $n$ -cube topologies: 16-ary, 2-cube (256 nodes), 8-ary, 3-cube (512 nodes), 32-ary, 2-cube (1024 nodes). I present detailed results and analysis for the 16-ary, 2-cube in Sections 5.5.1–5.5.4. Section 5.5.5 describes BLAM performance on the other two topologies that I consider, i.e., 8-ary, 3-cube and 32-ary, 2-cube.

Each router has one injection channel (through which packets sent by that node enter the network) and one delivery channel (through which packets sent to that node exit the network). The routers use edge-buffers (buffers associated with virtual channels) that can hold an entire packet. There is a one cycle arbitration delay and a one cycle routing delay per packet. This is not a bottleneck because routing/arbitration occurs only for the header flit of a packet. The remaining flits simply stream behind the header flit along the same switch path. It takes one cycle per flit to traverse the cross-bar switch and one cycle per flit to traverse a physical link. Each physical link is capable of full duplex communication.

## 5.5 Simulation Results

This section presents simulation results. I begin by examining the overall performance of the BLAM router. This is followed by a comparison of BLAM with three virtual

channels (i.e., three each of the input and bypass buffers per physical channel) against a minimal, adaptive router with six virtual channels. I call this a "resource-neutral" comparison, because these two configurations have the same number of buffers and crossbar inputs. Next, I evaluate the effects of varying the maximum number of allowed misroutes. Finally, I dissect BLAM performance by examining the effects of adding *eager* misroutes, *lazy misroutes* and bypass buffers, in isolation, to the base router.

The primary conclusions from the simulations are:

1. The BLAM router sustains near-*Chaos* throughput for all considered communication patterns at high offered load levels where the base case suffers a drop in throughput.
2. The combined use of lazy misroutes and bypass buffers are necessary for high performance.
3. I isolate the effects of misroutes to validate and extend previous research findings that misroutes alone, whether *eager* or *lazy*, decreases performance.

### 5.5.1 Overall Performance

This section examines the performance of a complete BLAM router implementation with a limit of 16 misroutes on a 16-ary, 2-cube with 16-flit packets. I explore other topologies and packet sizes later in this section. Figure 5.6 shows the bandwidth (left graphs [a] and [c]) and latency (right graphs [b] and [d]) with the random communication pattern for the base cases with deadlock recovery (top graphs [a] and [b]) and deadlock avoidance (bottom graphs [c] and [d]). Figure 5.7 have similar graphs for the bit-reversal, complement and perfect shuffle traffic patterns respectively. Note the logarithmic scale used on the y-axis for the latency graphs.

Consider the graphs for the uniform random traffic (Figure 5.6). The curve for the base router illustrates the network saturation problem. As load increases, the network throughput increases to a certain extent. However, at saturation, there is a sudden drop in throughput since only the escape paths are available for packets to make forward progress.

There are two curves corresponding to the BLAM router: one for the configuration with three virtual channels (3VC) per physical channel ( $\times$  in Figure 5.6-Figure 5.8) and another for the configuration with six virtual channels (6VC) per physical channel ( $\triangle$  in Figure 5.6-Figure 5.8). Both curves show that BLAM is able to prevent the drop in performance that occurs at high loads due to deadlocks in adaptive channels. Furthermore, BLAM achieves performance comparable or superior to chaos while providing deterministic guarantees.

Similar results for deadlock avoidance show that, qualitatively, the only difference in performance as compared to the deadlock recovery configuration is the higher saturation throughput for the base case. BLAM (on top of deadlock avoidance) still outperforms the base deadlock avoidance configuration.

### **Throughput/Latency Tradeoff**

Figure 5.6 also demonstrates a throughput/latency tradeoff based on the number of virtual channels. The 6VC configuration shows higher throughput than the 3VC configuration, however this comes at the cost of higher latency. The reason is the 6VC BLAM has a higher number of bypass-buffers, and a packet is able to stay in a buffer for a longer period of time. Recall, a packet is misrouted only when another packet needs to enter the same bypass buffer. I verified this phenomenon by measuring the amount of time a packet spends in the bypass-buffers. For the six heaviest applied loads in Figure 5.6, where the difference in latencies becomes obvious, packets

stay in the bypass buffers 35%, 77%, 55% and 48% longer, on average, in the 6VC configuration than in the 3VC configuration for the *uniform random*, *perfect shuffle*, *complement* and *bit reversal* traffic patterns, respectively. To better understand this throughput/latency tradeoff, I compare the bandwidth and latency of 6VC BLAM and 3VC BLAM to the throughput and latency of Chaos for the six heaviest applied loads.

6VC BLAM achieves better throughput than *chaos* for three of the four communication patterns. The throughput improves by as much as 56% (for *bit-reversal*) and is 5% lower only in the case of *uniform random* traffic. The behavior of 3VC BLAM is qualitatively similar with slightly lower throughput. For 3VC BLAM, the throughput vary from 10% lower to 48% higher than *chaos*.

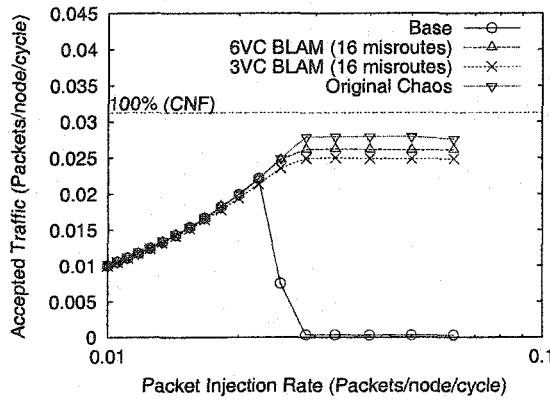
The latencies for 6VC BLAM are between 35% (for *bit-reversal*) and 151% higher (for *perfect shuffle*) than Chaos latencies, on average. A similar latency comparison for 3VC BLAM shows that it achieves latencies varying from 22% lower (for *complement*) to 42% higher (for *perfect shuffle*) than Chaos latencies. In conclusion, 6VC BLAM suffers from increased latency to achieve better-than-*chaos* throughput. In contrast, 3VC BLAM achieves much better latencies than 6VC BLAM for a small penalty on throughput.

### Resource-Neutral Comparison

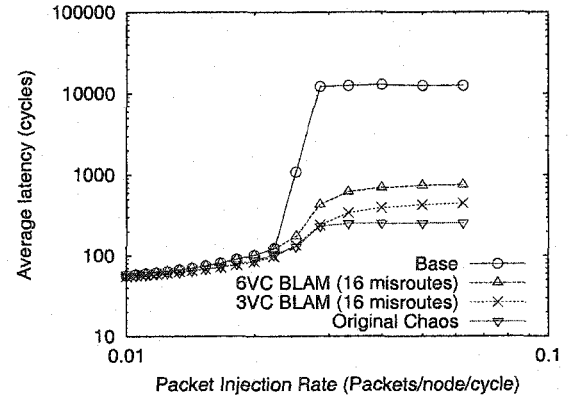
Comparing the six virtual channel base case against the BLAM router with six virtual channels is not a resource-neutral comparison. This is because BLAM requires additional bypass buffers for each virtual channel, in this case doubling the number of buffers. BLAM also requires a larger crossbar with twice as many inputs: the base virtual channel inputs and the bypass buffer inputs. A resource neutral comparison can be made between the six virtual channel minimal adaptive router and the BLAM

router with three virtual channels. From Figs 3.4–3.7 this resource neutral comparison reveals that the 3VC BLAM router ( $\times$ ) outperforms the base router ( $\circ$ ) for all traffic patterns. The 3VC BLAM router maintains high throughput at higher load levels whereas the 6VC base router saturates.

### Deadlock Recovery

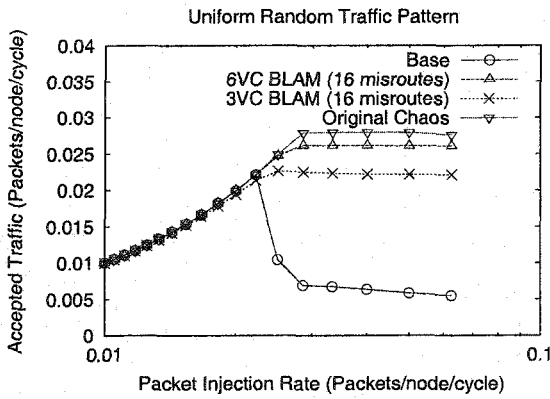


(a) Delivered Throughput vs. Offered Load

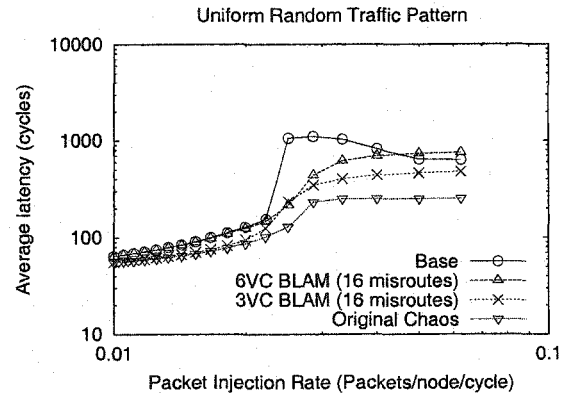


(b) Average Latency vs. Offered Load

### Deadlock Avoidance



(c) Delivered Throughput vs. Offered Load

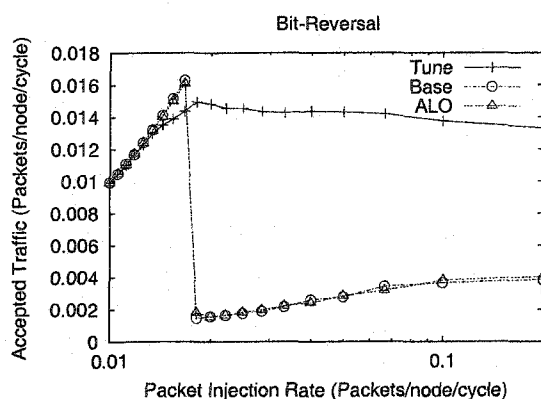


(d) Average Latency vs. Offered Load

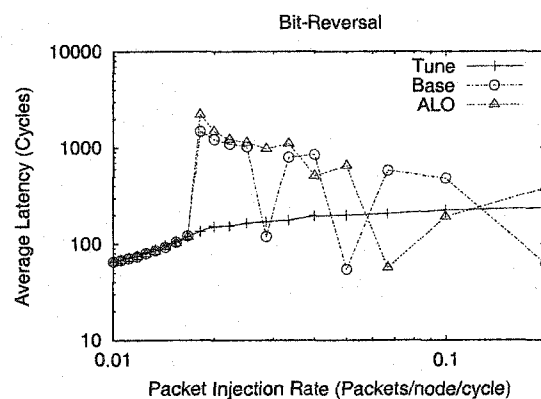
**Figure 5.6: Overall Performance With Random Traffic**



## Deadlock Recovery

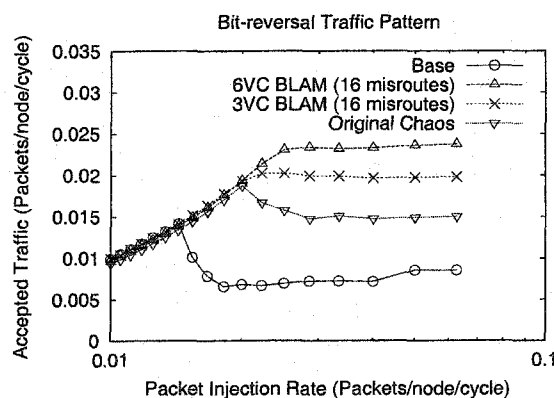


(a) Delivered Throughput vs. Offered Load

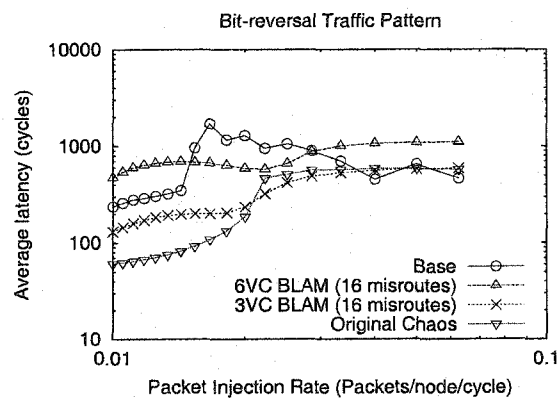


(b) Average Latency vs. Offered Load

## Deadlock Avoidance



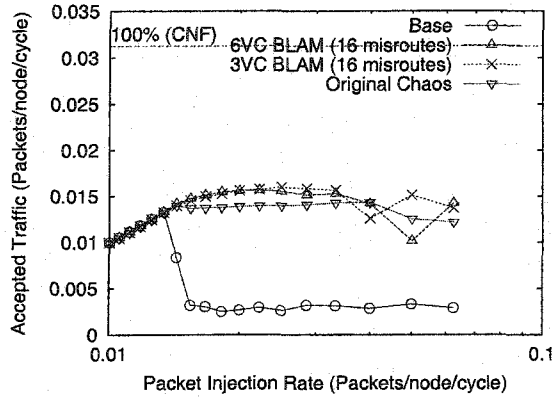
(c) Delivered Throughput vs. Offered Load



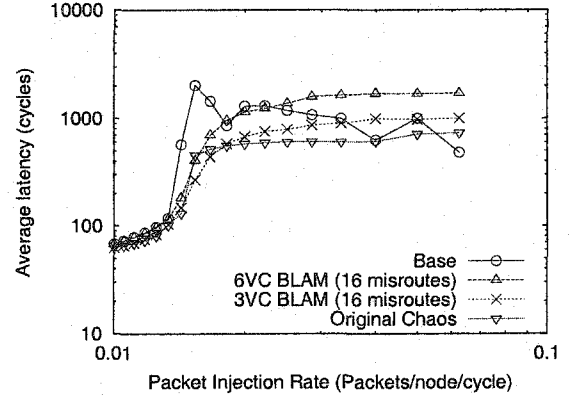
(d) Average Latency vs. Offered Load

**Figure 5.7:** Overall Performance With Bit-Reversal Traffic Pattern

### Deadlock Recovery

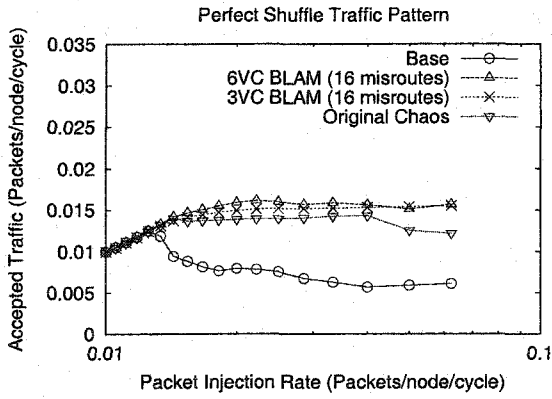


(a) Delivered Throughput vs. Offered Load

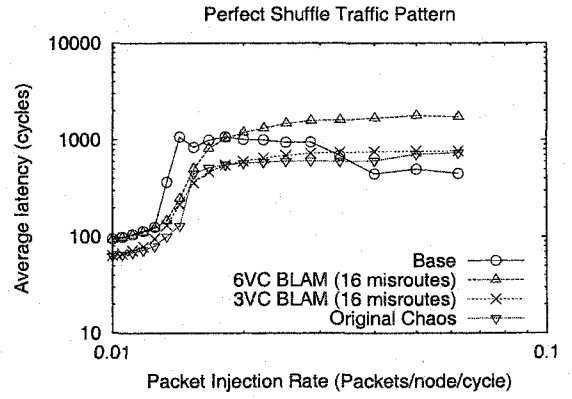


(b) Average Latency vs. Offered Load

### Deadlock Avoidance



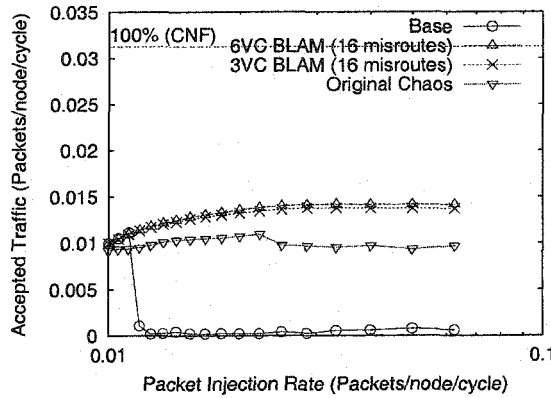
(c) Delivered Throughput vs. Offered Load



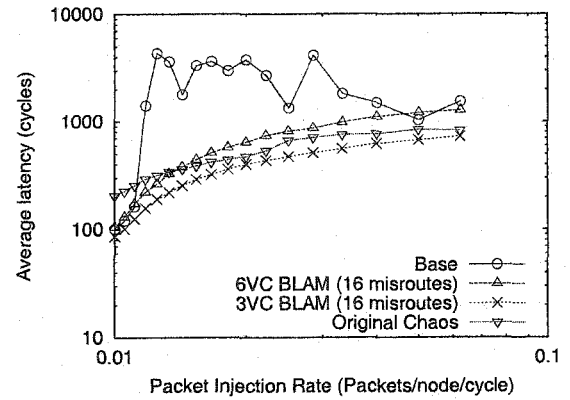
(d) Average Latency vs. Offered Load

**Figure 5.8:** Overall Performance With Perfect-Shuffle Traffic Pattern

### Deadlock Recovery

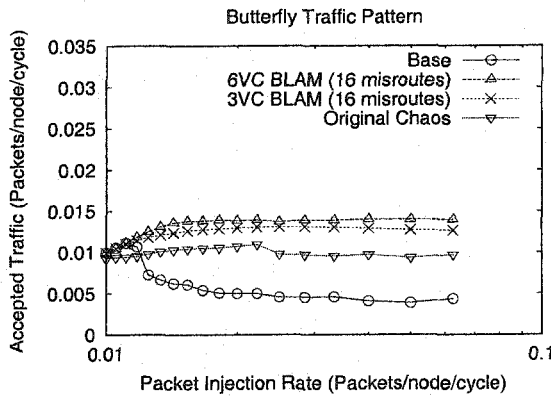


(a) Delivered Throughput vs. Offered Load

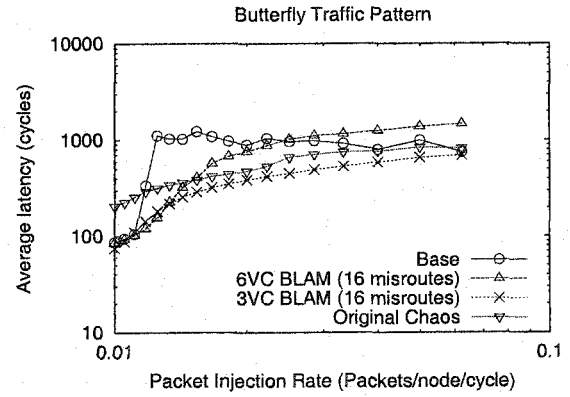


(b) Average Latency vs. Offered Load

### Deadlock Avoidance



(c) Delivered Throughput vs. Offered Load



(d) Average Latency vs. Offered Load

**Figure 5.9: Overall Performance With Complement Traffic Pattern**

### 5.5.2 Varying the Misroute Limit

The previous results use a BLAM router with up to 16 misroutes. In this subsection, I examine the trade-offs involved in varying this limit. Figure 5.11 shows the performance of 6VC BLAM as the misroute limit is increased progressively from 3 to 8 to 16. In general, increasing the number of misroutes postpones saturation as packets can use the adaptive channels for a longer period of time.

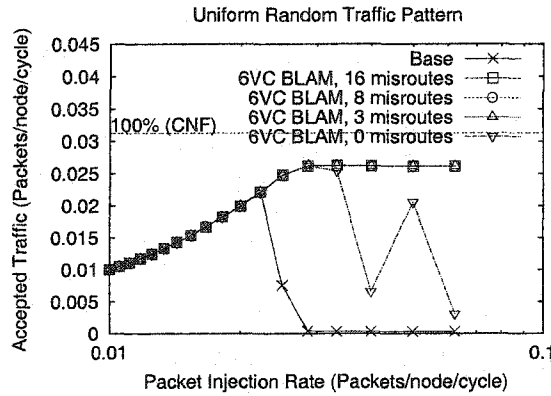
For *uniform random* traffic, a limit of three misroutes is enough to prevent saturation. Increasing the number of misroutes beyond three does not change the behavior in any significant manner. In contrast, the *perfect-shuffle* communication pattern (see Figures 5.11c & 5.11d) shows well-separated performance curves as the misroute limit is varied. This facilitates explanation of network behavior in response to variations in the misroute limit.

As the misroute limit increases, network saturation (and the corresponding drop in performance) occurs at higher and higher loads. Allowing up to three misroutes prevents saturation at certain loads. But at higher loads, the packets use up all the allowed misroutes and are then constrained to route only within the minimum rectangle. This increases the frequency of deadlock cycles forming in the adaptive channels and hence increases the frequency of deadlock-free channel usage. Similarly, 8 misroutes, while better than 3 misroutes, is unable to prevent saturation at the higher loads.

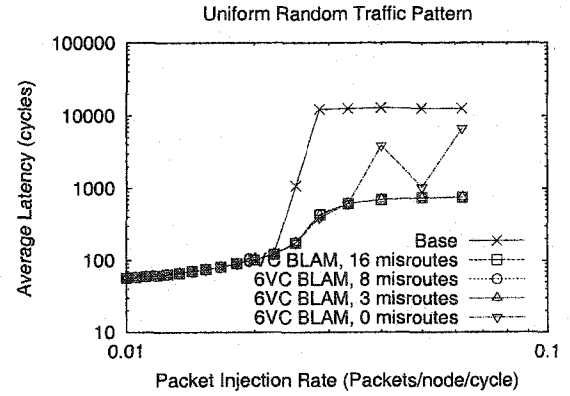
Note, 16 misroutes is high enough for all the loads and communication patterns I considered. In general, the number of misroutes should be set to a value that is high enough to reduce the probability of using the deadlock-free escape paths for any workload that the network may handle. However, for some workloads the network could saturate but the resulting behavior will be no worse than that of a minimal

adaptive router. Theoretically, high latencies are possible if the limit on misroutes is very high, because my BLAM implementation does not have the randomization property of the *Chaos* router which increases the probability of packet delivery with increasing time. However, I have not observed this in practice.

## Deadlock Recovery

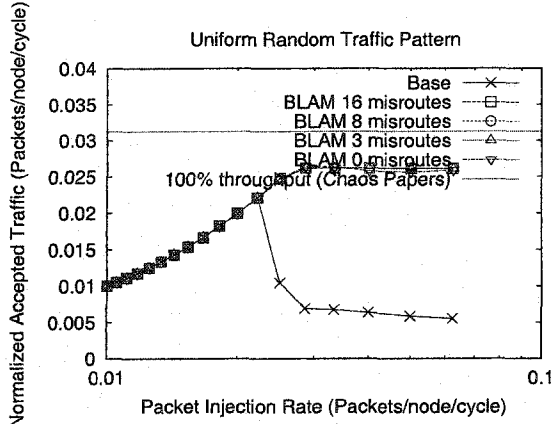


(a) Delivered Throughput vs. Offered Load

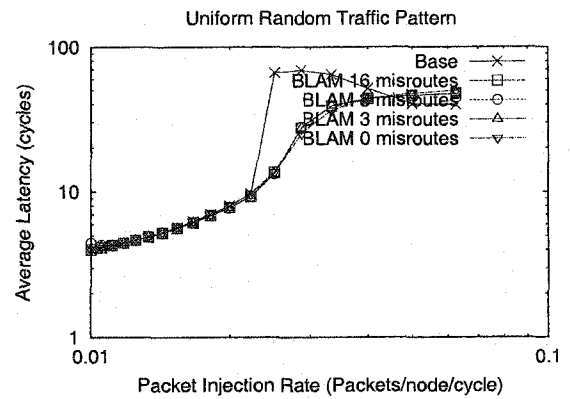


(b) Average Latency vs. Offered Load

## Deadlock Avoidance



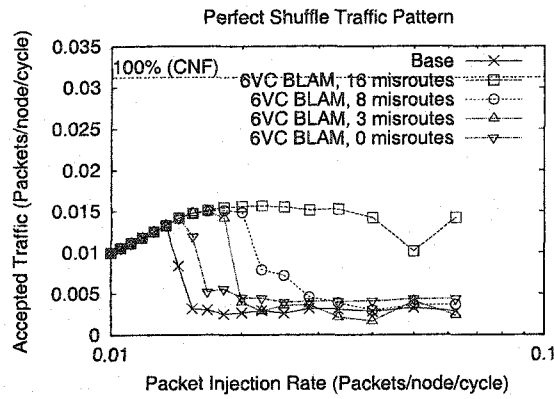
(c) Delivered Throughput vs. Offered Load



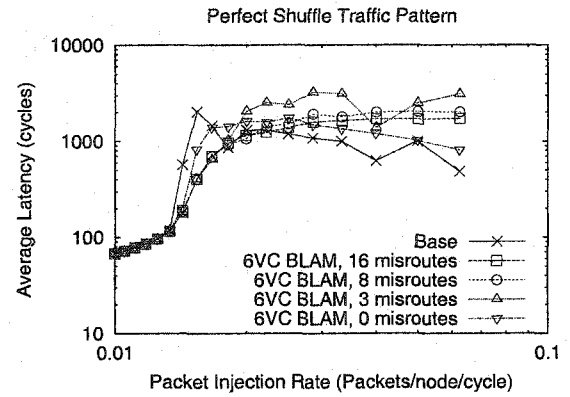
(d) Average Latency vs. Offered Load

**Figure 5.10:** Effects of Varying the Misroute Limit of BLAM, Uniform Random Traffic

## Deadlock Recovery

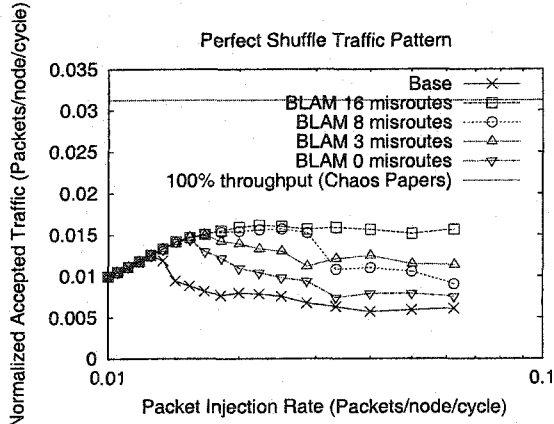


(a) Delivered Throughput vs. Offered Load

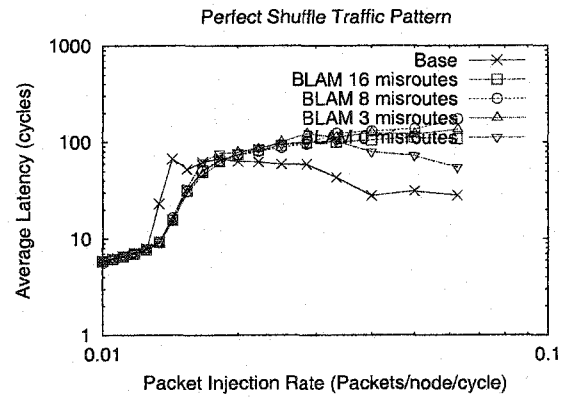


(b) Average Latency vs. Offered Load

## Deadlock Avoidance



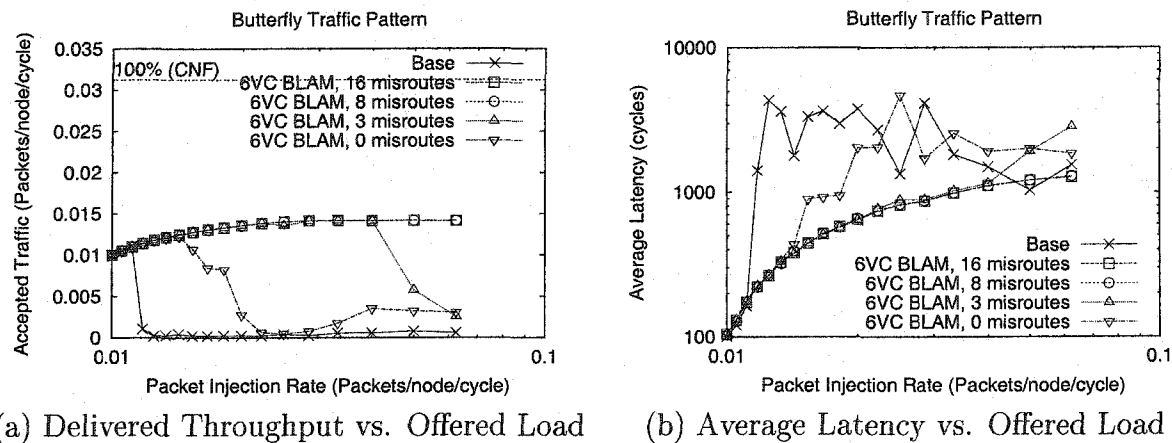
(c) Delivered Throughput vs. Offered Load



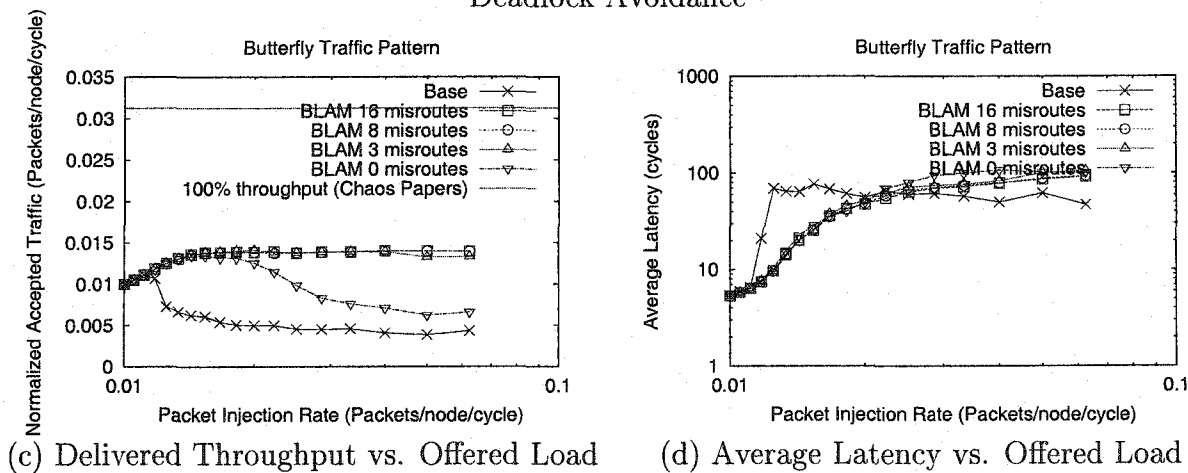
(d) Average Latency vs. Offered Load

**Figure 5.11:** Effects of Varying the Misroute Limit of BLAM, Perfect Shuffle Traffic

## Deadlock Recovery



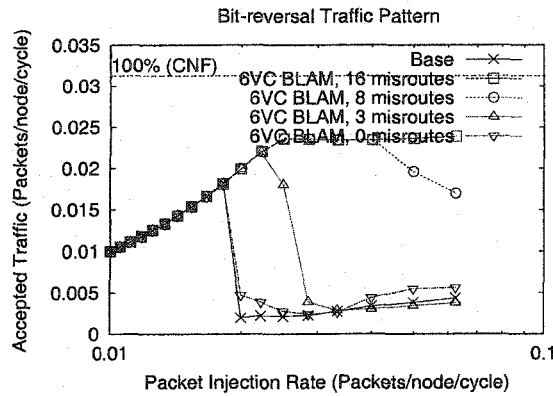
## Deadlock Avoidance



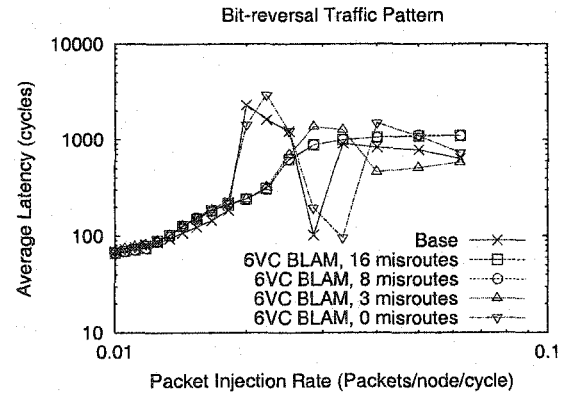
**Figure 5.12:** Effects of Varying the Misroute Limit of BLAM, Complement Traffic



## Deadlock Recovery

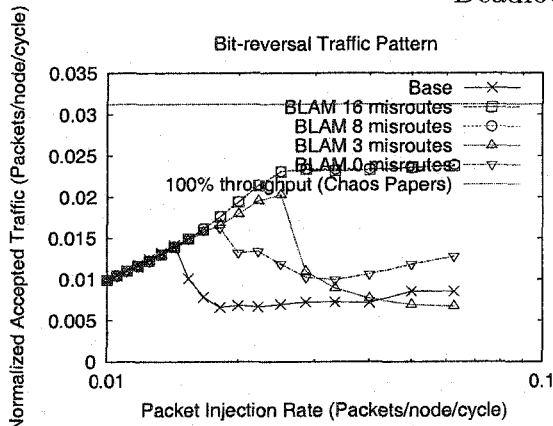


(a) Delivered Throughput vs. Offered Load

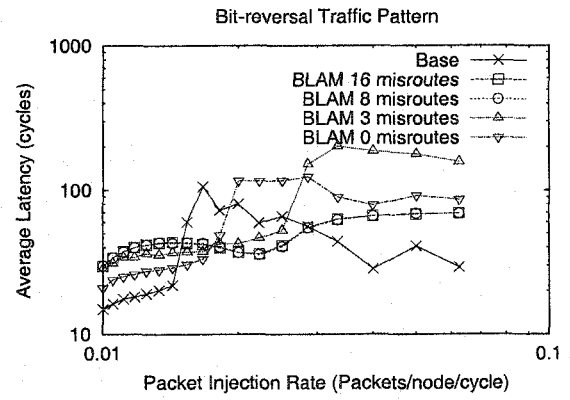


(b) Average Latency vs. Offered Load

## Deadlock Avoidance



(c) Delivered Throughput vs. Offered Load



(d) Average Latency vs. Offered Load

**Figure 5.13:** Effects of Varying the Misroute Limit of BLAM, Bit Reversal Traffic

### 5.5.3 Effect of Adding Bypass Buffers

The bypass buffers create additional buffering capacity in each node which can break some hold-and-wait cycles, thus improving performance over the base case. The curve for zero misroutes ( $\nabla$ ) in Figure 5.11 isolates the effect of only adding bypass buffers to the base router. While there is some marginal improvement over the base case, increasing network load eventually saturates the additional buffers.

### 5.5.4 $M$ -misroute, Adaptive router

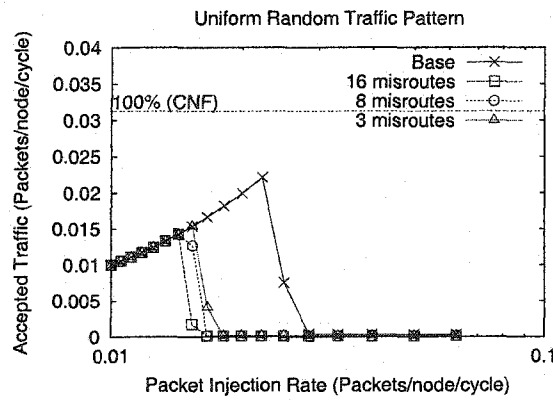
The next section examines the effects of adding misroutes (both *eager* and *lazy*) without bypass buffers to the base router.

*Eager Misroutes:* In this scheme misroutes are initiated eagerly whenever a packet is unable to find a profitable route. Figure 5.15 shows the simulation results with eager misroutes. From these simulations, we observe that eager misrouting without bypass buffers is insufficient to prevent saturation. In fact, eager misrouting consistently performs worse than the base router, and increasing the misroute limit further exacerbates saturation. This behavior matches expectations.

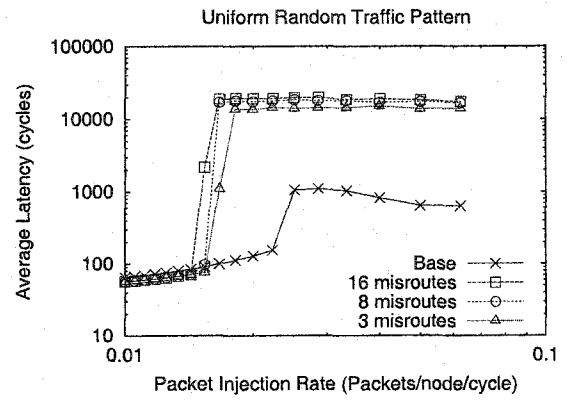
Previous research [38] shows that starting out with a minimal adaptive router (with wormhole routing and Disha deadlock recovery) and enabling a limited number of misroutes does not improve performance for uniform random traffic. My experiments reproduce this result for virtual cut-through switching and other traffic patterns. Note, previous work did show that misroutes can be helpful in the case of hot-spot traffic pattern (where one node is the destination for packets coming from many source nodes.)

*Lazy misroutes:* BLAM performance combines the effects of bypassing and *lazy* misrouting. To separate the effects of bypassing from *lazy* misrouting I modified the eager misrouting technique to delay misrouting for a set number of cycles—called spin-cycles. During this time the packet continues to try and obtain a profitable route but packets that follow cannot bypass the stalled packet. Simulation results show that while *lazy* misrouting is better than *eager* misrouting, it is still worse than the base case. Figures 5.18 and 5.19 shows the performance of the base configuration with upto 3 misroutes and various spin-cycles. These results demonstrate that the benefit of avoiding unproductive misroutes is lost because of the lack of bypassing, which stalls packets that follow.

## Deadlock Recovery

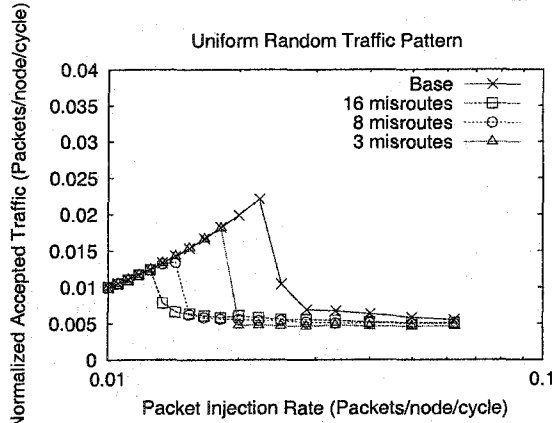


(a) Delivered Throughput vs. Offered Load

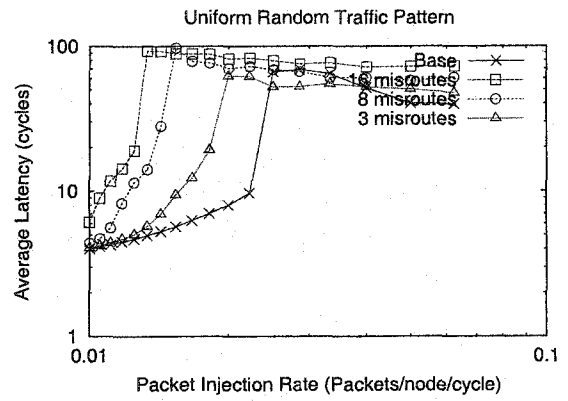


(b) Average Latency vs. Offered Load

## Deadlock Avoidance



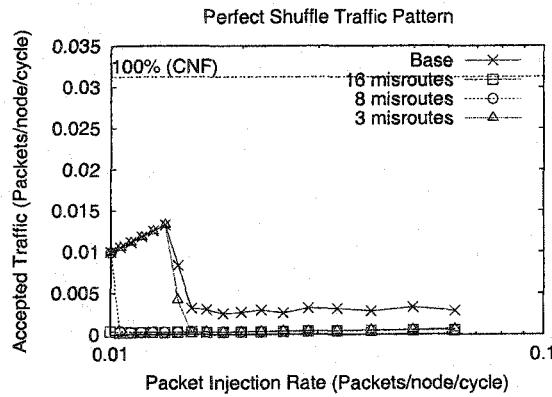
(c) Delivered Throughput vs. Offered Load



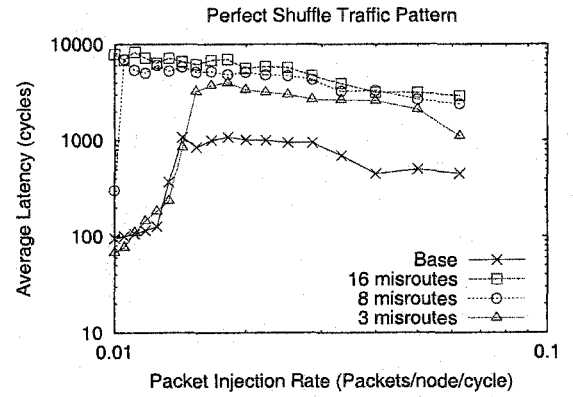
(d) Average Latency vs. Offered Load

**Figure 5.14:** *M*-misroute, Adaptive Router, Uniform Random Traffic

## Deadlock Recovery

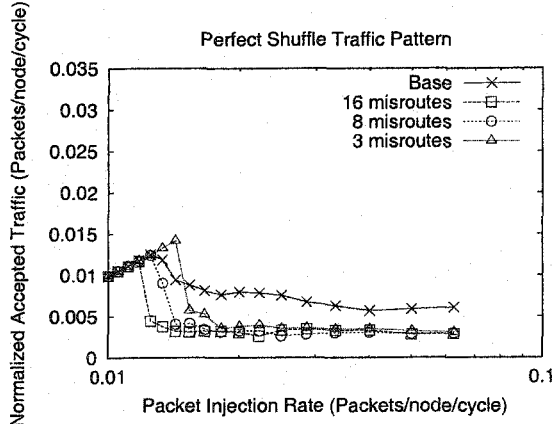


(a) Delivered Throughput vs. Offered Load

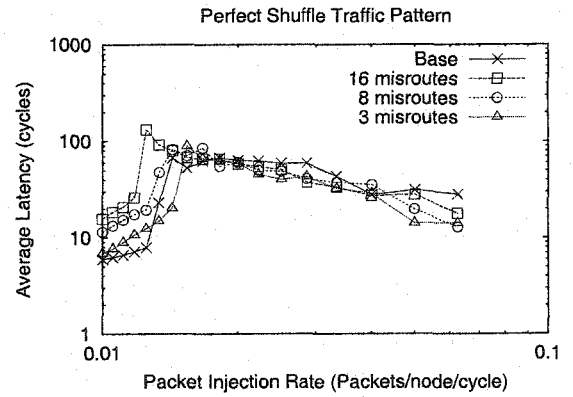


(b) Average Latency vs. Offered Load

## Deadlock Avoidance



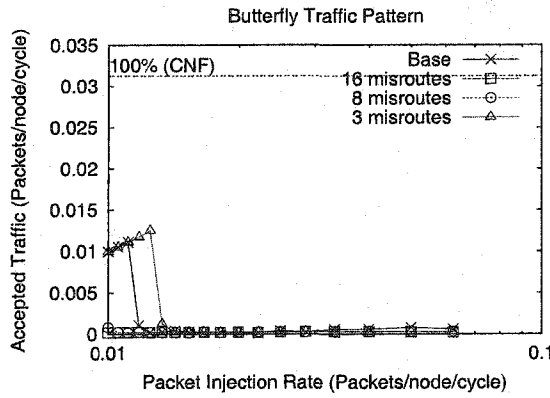
(c) Delivered Throughput vs. Offered Load



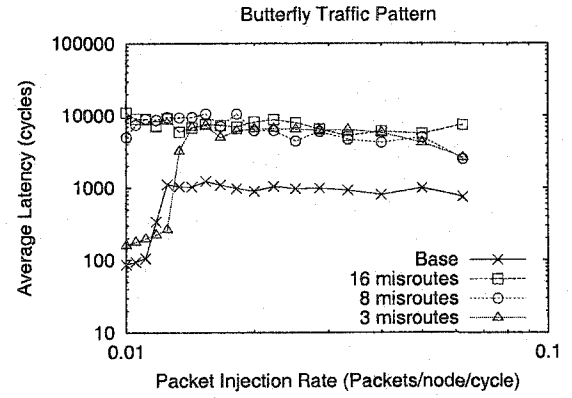
(d) Average Latency vs. Offered Load

**Figure 5.15:** *M*-misroute, Adaptive Router, Perfect Shuffle Traffic

## Deadlock Recovery

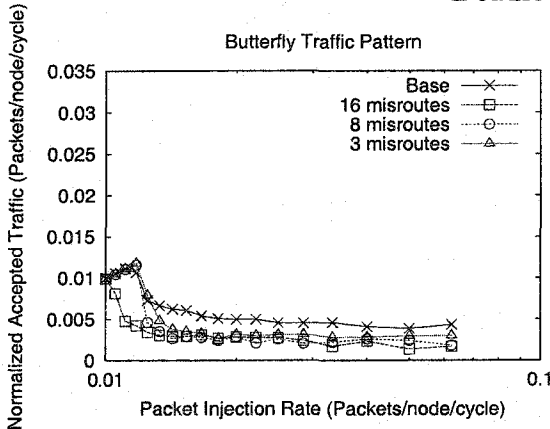


(a) Delivered Throughput vs. Offered Load

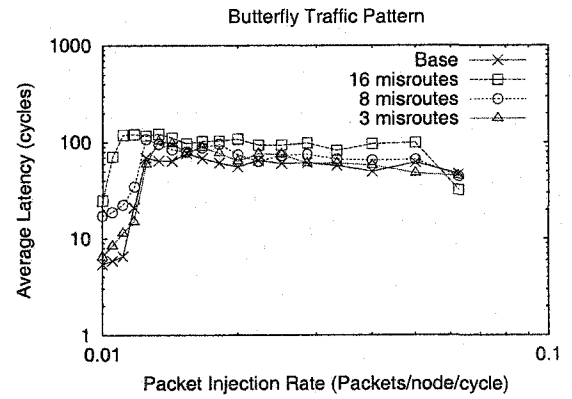


(b) Average Latency vs. Offered Load

## Deadlock Avoidance



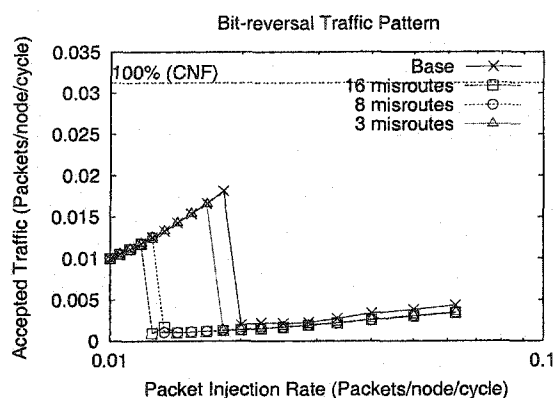
(c) Delivered Throughput vs. Offered Load



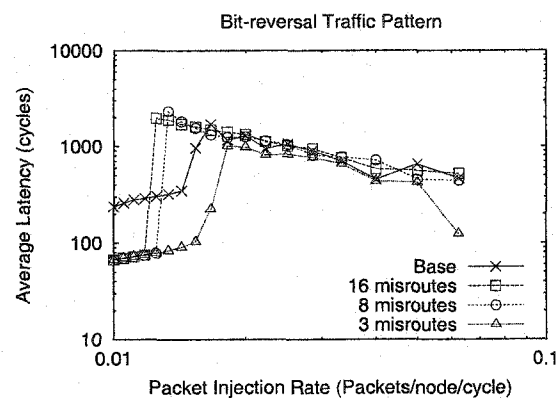
(d) Average Latency vs. Offered Load

Figure 5.16: *M*-misroute, Adaptive Router, Complement Traffic

## Deadlock Recovery

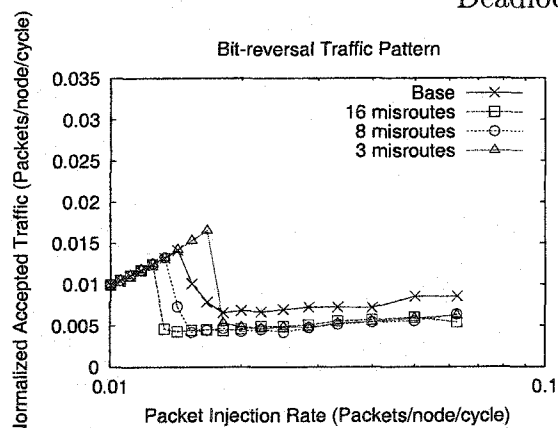


(a) Delivered Throughput vs. Offered Load

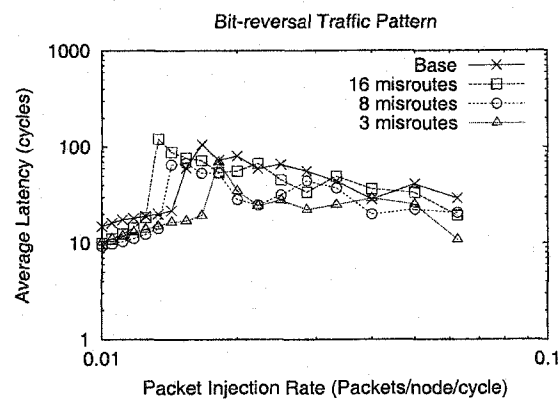


(b) Average Latency vs. Offered Load

## Deadlock Avoidance



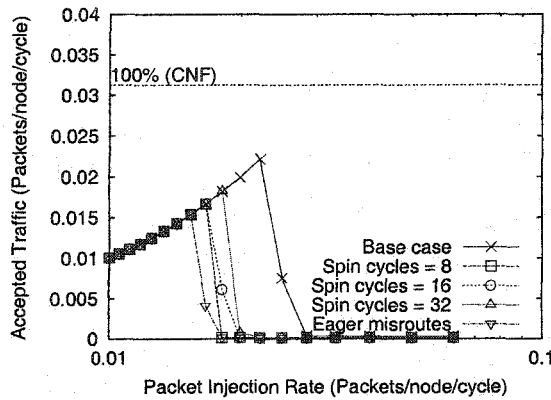
(c) Delivered Throughput vs. Offered Load



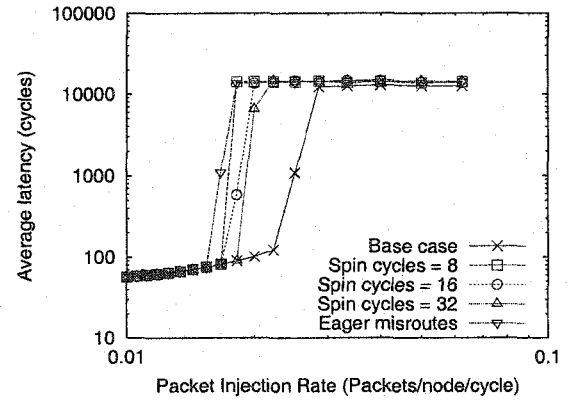
(d) Average Latency vs. Offered Load

**Figure 5.17:** *M*-misroute, Adaptive Router, Bit Reversal Traffic

### Uniform Random

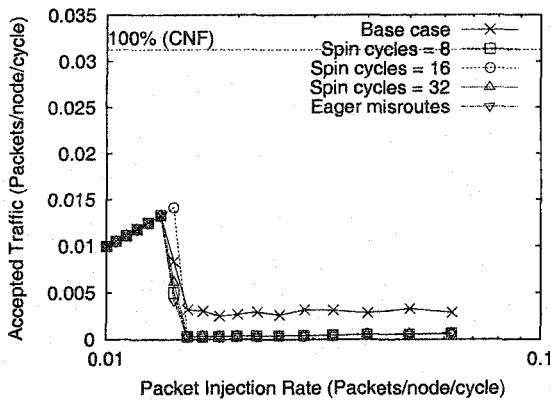


(a) Delivered Throughput vs. Offered Load

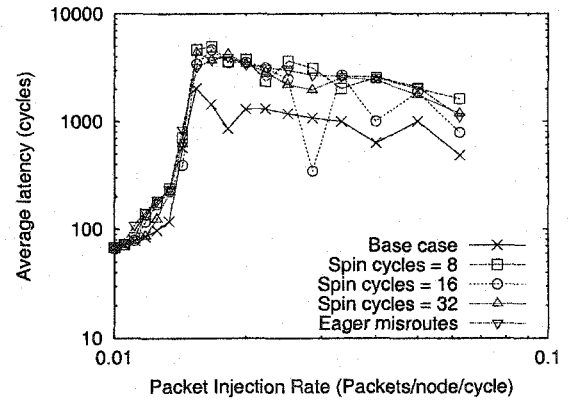


(b) Average Latency vs. Offered Load

### Perfect Shuffle



(c) Delivered Throughput vs. Offered Load

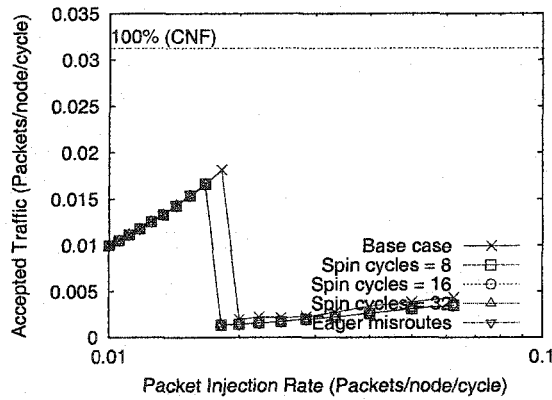


(d) Average Latency vs. Offered Load

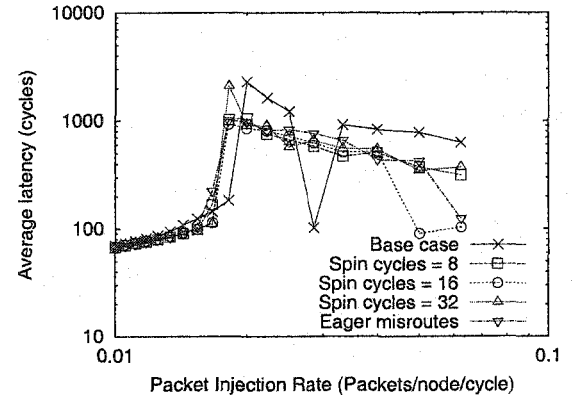
Figure 5.18: *M*-misroute, Adaptive Router with *lazy* Misroutes



### Bit Reversal

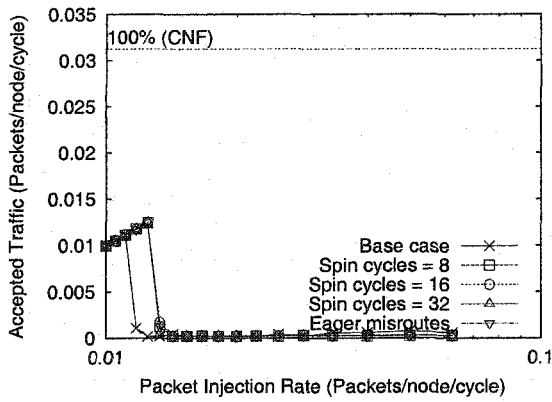


(a) Delivered Throughput vs. Offered Load

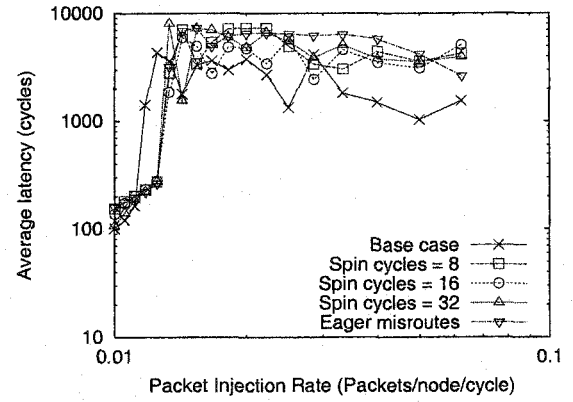


(b) Average Latency vs. Offered Load

### Complement



(c) Delivered Throughput vs. Offered Load



(d) Average Latency vs. Offered Load

Figure 5.19: *M*-misroute, Adaptive Router with *lazy* Misroutes

### 5.5.5 Varying Packet Size and Network Size

So far, I have presented results for 16-ary, 2-cubes with 16 flit packets. It is important to verify that the results we saw are representative for the design space of realistic distributed shared memory multiprocessors. Evaluating BLAM for every point in this huge design space<sup>3</sup> is not feasible. In this section, I present results for some network sizes and packet sizes selected on the basis of technology trends.

#### Other packet sizes

In the context of cache-coherent, distributed shared memory multiprocessors, network packets carry blocks of data corresponding to the cache blocksize of the cache most distant from the processor. In current day systems, this typically means the level 2 (L2) cache. The L2 cache block size in recent processors varies between 64 Bytes (UltraSPARC 2, Alpha 21164, Alpha 21264) and 128 bytes (Pentium 4). Using these as typical cache block sizes and using the Alpha 21364 flit size (4 Bytes) as the typical flit width, the packet sizes of interest range between 8 flits and 32 flits. Taking technology trends into account, I also consider larger 64 flit packets.

From my simulation results of a 16-ary, 2-cube network with workload using 32 flit and 64 flit packets (Figures 5.20–5.23), we observe from that the base network saturates at a lower packet injection rate when packet size is increased to 32 flits and 64 flits. This is expected because the load, as measured in flit injection rate, increases when the packet size increases. Apart from this difference, the relative performance of the base minimal adaptive router, 6VC BLAM and 3VC BLAM remain qualitatively similar.

---

<sup>3</sup>where, design space = {(all packet sizes) X (all network sizes)}

## Other network sizes

The 16-ary, 2-cube network I considered in earlier sections had 256 nodes. In this section, I evaluate BLAM with 512 node and 1024 node networks as well. For the 512 node system, I consider an 8-ary, 3-cube topology to verify BLAM operation at higher dimensions as well. For the 1024 node system, I use a 32-ary, 2-cube. Figures 5.24, 5.25, 5.26 and 5.27 show the performance of BLAM for an 8-ary, 3-cube (512 nodes) network ((a) and (b)) and a 32-ary, 2-cube (1024 nodes) network ((c) and (d)) with 16 flit packets for *uniform random*, *bit reversal*, *complement* and *perfect shuffle* traffic patterns respectively. For these two network sizes, the BLAM misroute limit is set to 32 for the 8-ary, 3-cube and to 64 for the 32-ary, 2-cube, because a limit 16 misroutes was insufficient to prevent the heavy use of deadlock-free paths resulting in poor performance.

## 5.6 Summary

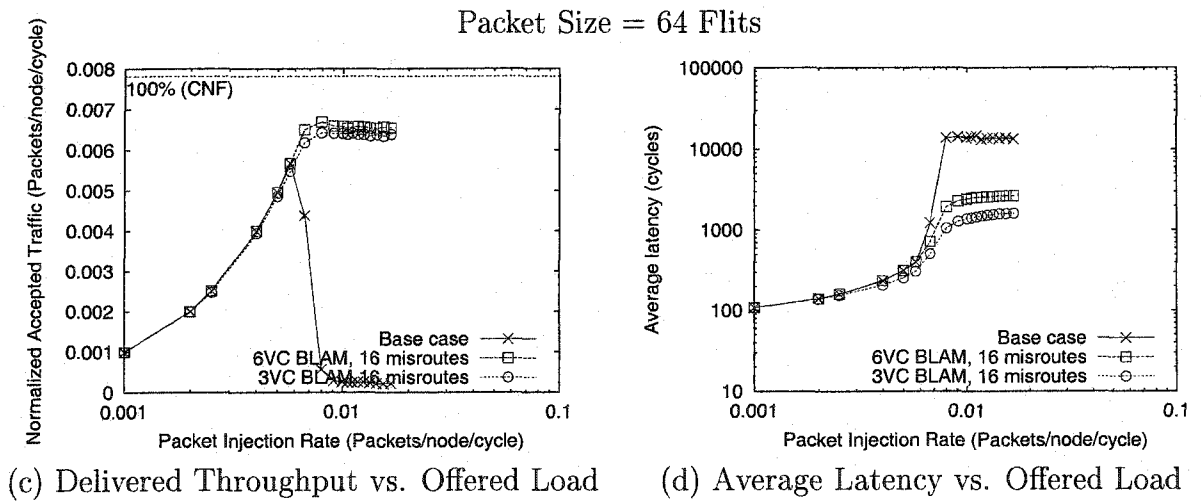
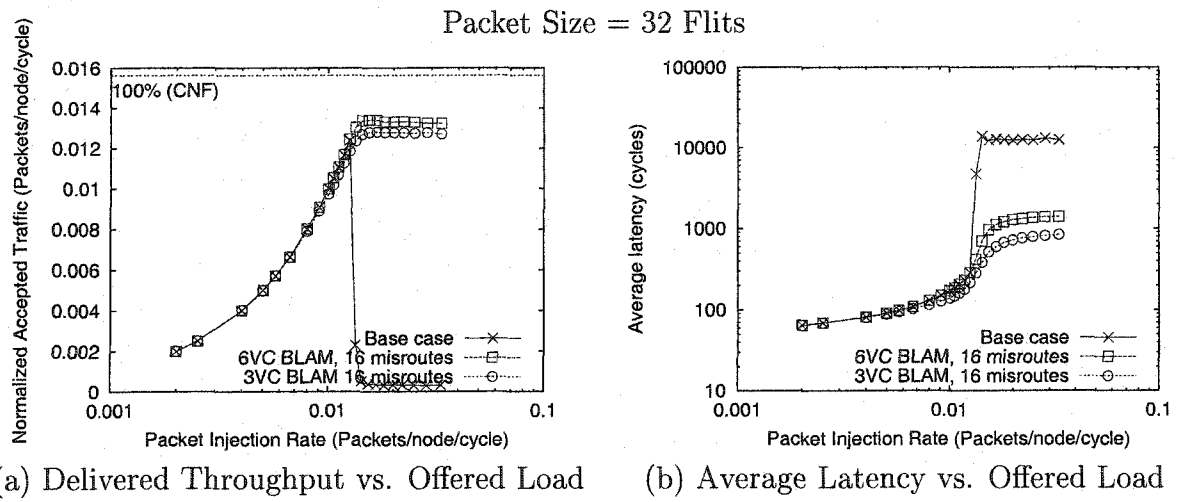
High performance, deadlock-freedom, and livelock-freedom are all important for multiprocessor interconnection networks. Unfortunately, existing routing algorithms trade off one property for others. Minimal, adaptive routing algorithms compromise performance at high loads to guarantee deadlock-freedom and livelock-freedom. In contrast, Chaotic routing algorithms accept weaker, probabilistic livelock-freedom guarantees to achieve high performance and deadlock-freedom. The challenge is to design a routing algorithm that combines the best of minimal adaptive routing and Chaotic routing to provide high performance without sacrificing deadlock- or livelock-freedom.

This chapter proposes a new routing algorithm—called BLAM (Bypass buffers with Limited Adaptive lazy Misrouting)—which achieves that goal. Like minimal

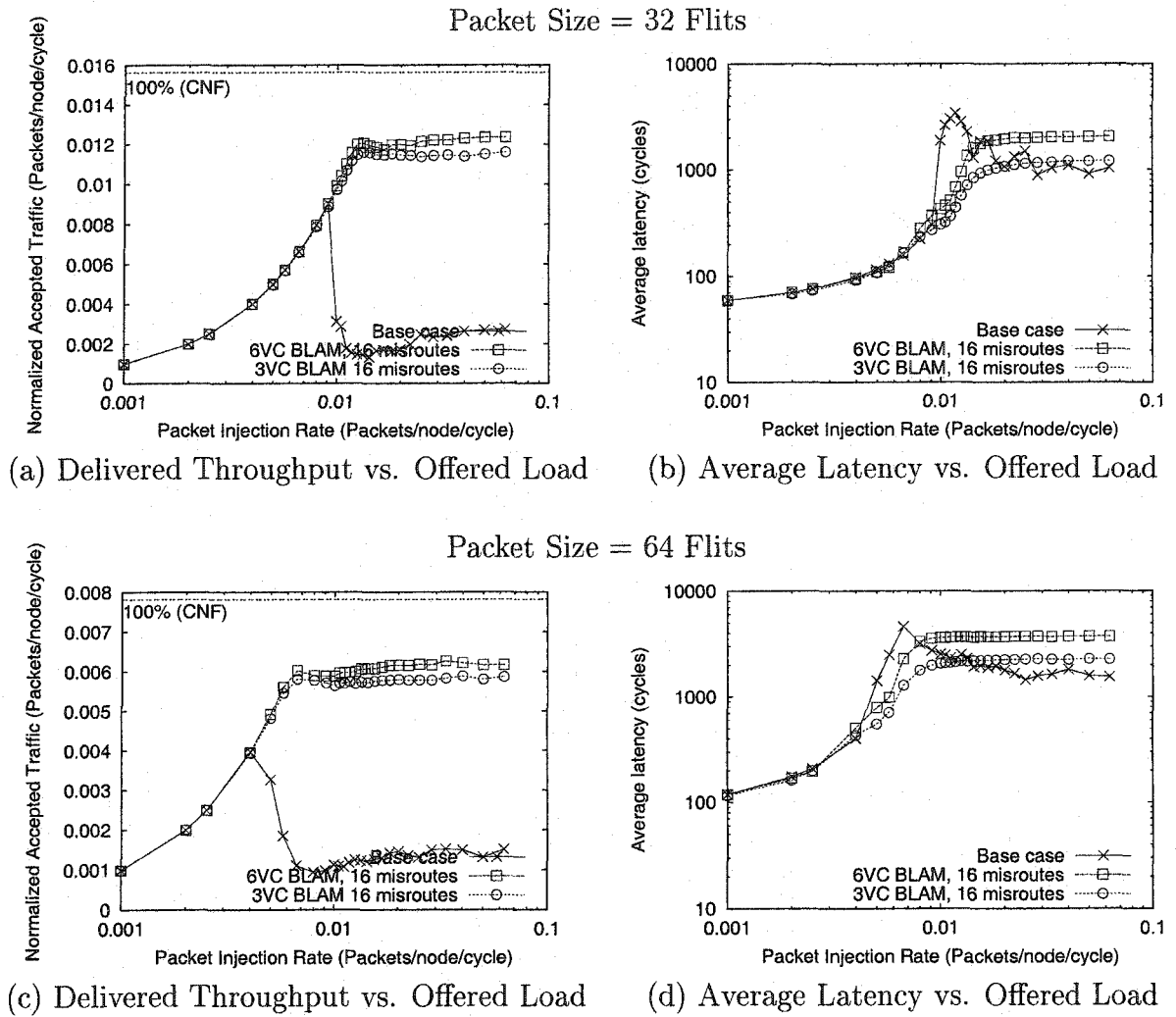
adaptive routing algorithms, BLAM provides deadlock-freedom by the use of a deadlock-free escape paths. Like Chaos, it provides high performance via the use of lazy misrouting and the *packet exchange protocol*. BLAM implements lazy misrouting via bypass buffers at the input ports. However, unlike Chaos, BLAM limits the number of misroutes, thereby, providing livelock-freedom.

Using simulation of a variety of configurations and communication patterns, I demonstrate that BLAM achieves very high network performance, which is comparable to what Chaos can achieve. Additionally, I demonstrate that components of BLAM—bypass buffers and lazy misrouting—may not necessarily improve performance individually. However, when combined in BLAM, these techniques can provide very high network throughput.

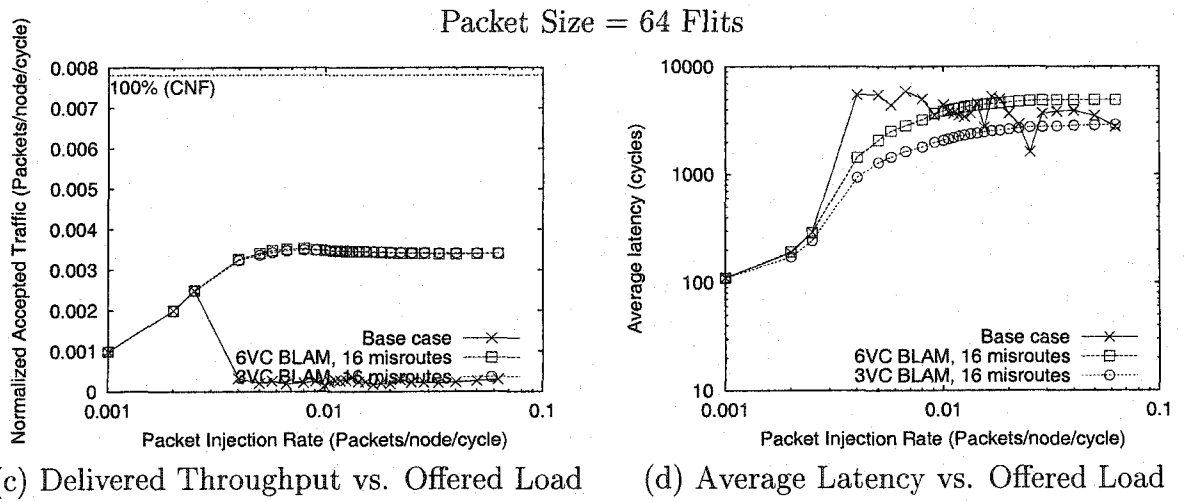
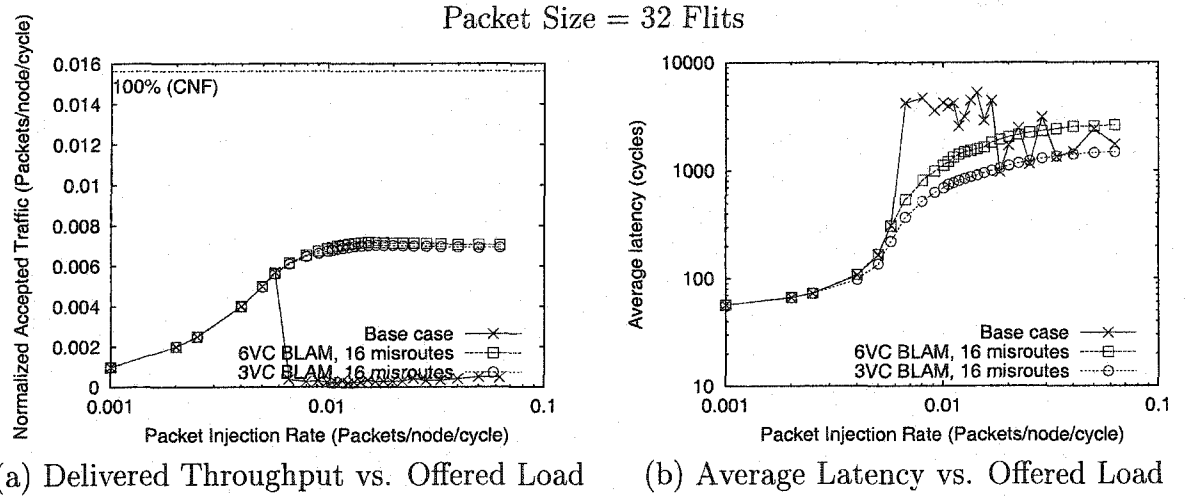
Unfortunately, inspite of their high performance, no commercial product (to the best of my knowledge) has adopted Chaotic routing algorithms, even though they have existed for more than a decade. The presence of livelocks—however, low the probability may be—causes network designers to shy away from such algorithms. BLAM removes this barrier by providing performance similar to Chaotic routing algorithms, but with guaranteed livelock- and deadlock-freedom.



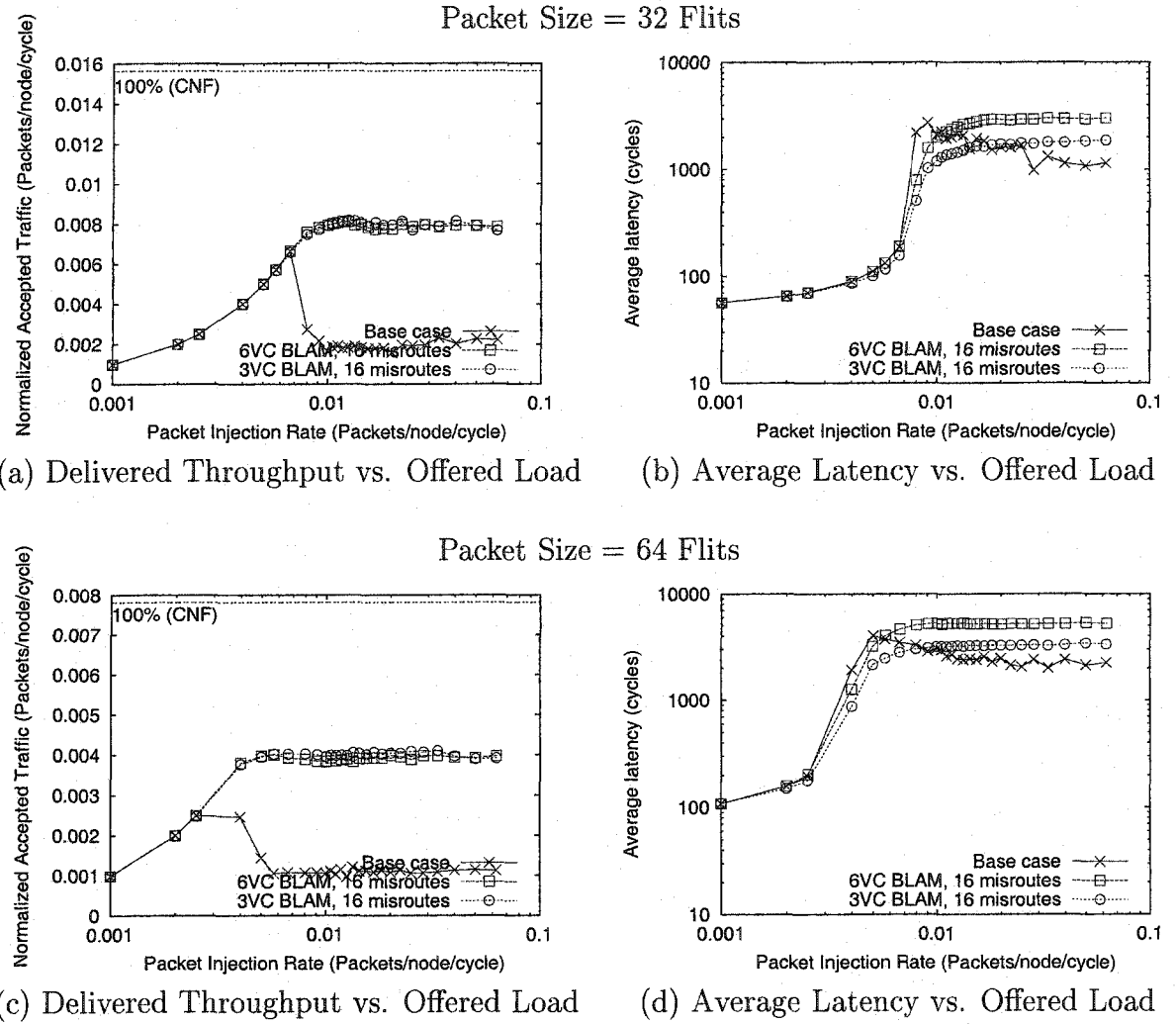
**Figure 5.20:** Overall Performance For Other Packet Sizes (Uniform Random Traffic, 16-ary, 2-cube Network)



**Figure 5.21:** Overall Performance For Other Packet Sizes (Bit-Reversal Traffic, 16-ary, 2-cube Network)



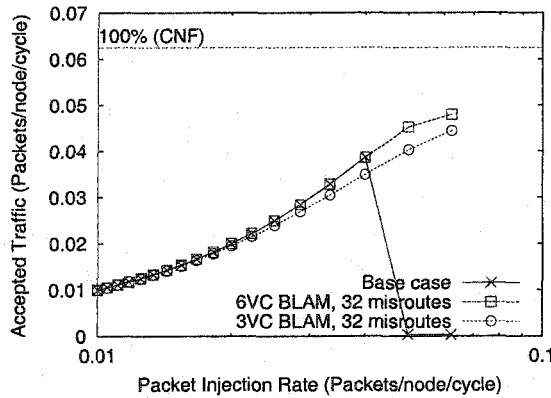
**Figure 5.22:** Overall Performance For Other Packet Sizes (Complement Traffic, 16-ary, 2-cube Network)



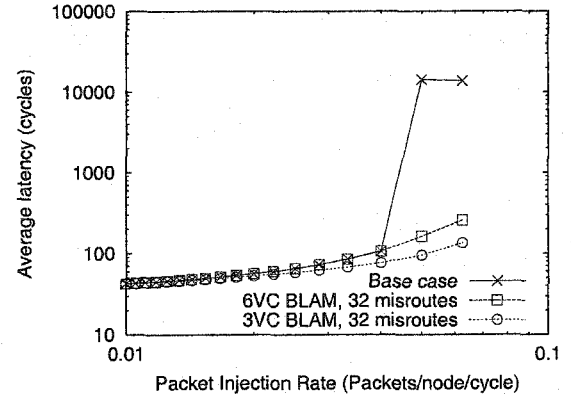
**Figure 5.23:** Overall Performance For Other Packet Sizes (Perfect Shuffle Traffic, 16-ary, 2-cube Network)



### 8-ary, 3-cube (512 node) Network

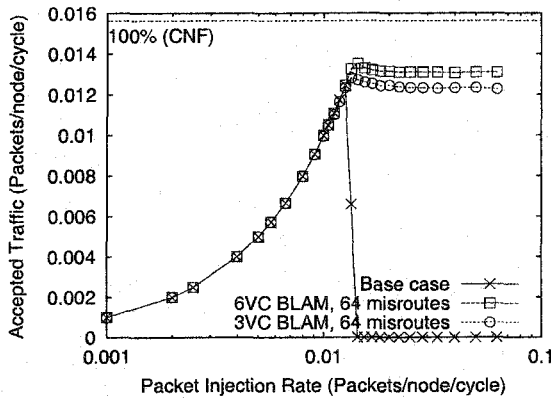


(a) Delivered Throughput vs. Offered Load

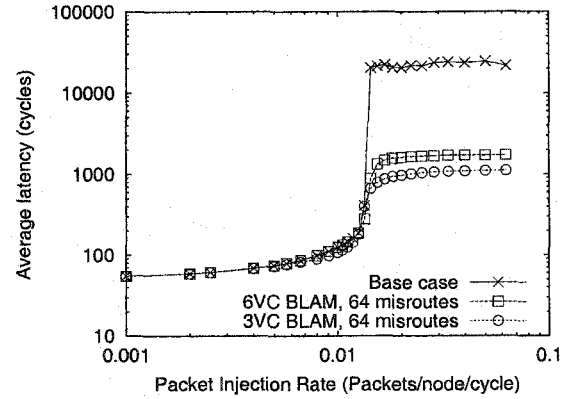


(b) Average Latency vs. Offered Load

### 32-ary, 2-cube (1024 node) Network



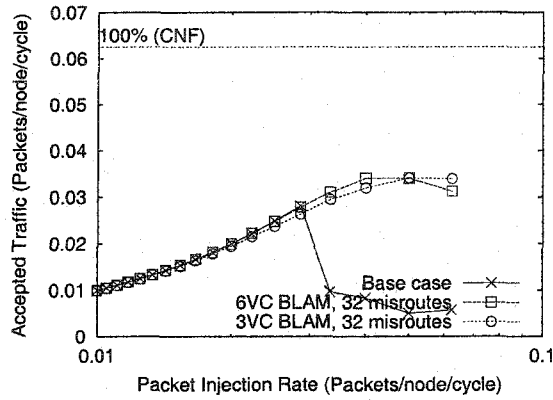
(c) Delivered Throughput vs. Offered Load



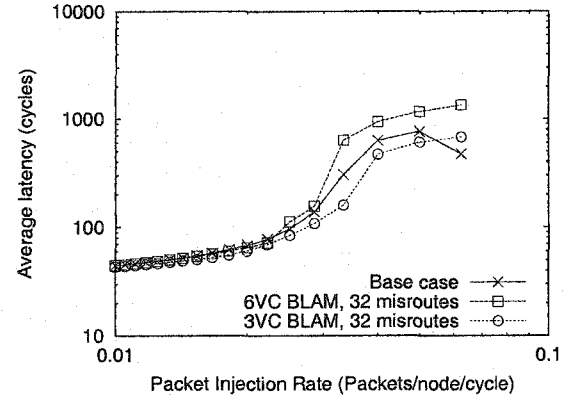
(d) Average Latency vs. Offered Load

**Figure 5.24:** Overall Performance For Other Network Sizes (Uniform Random Traffic, 16 flit packets)

### 8-ary, 3-cube (512 node) Network

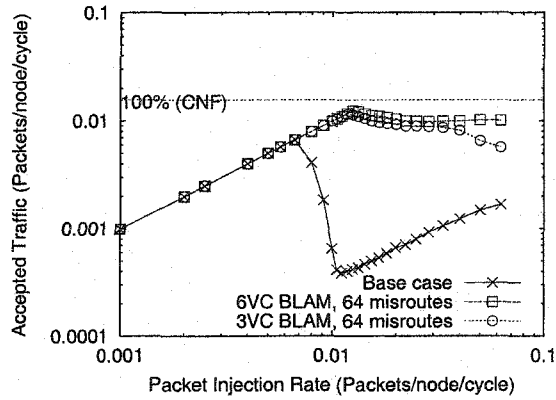


(a) Delivered Throughput vs. Offered Load

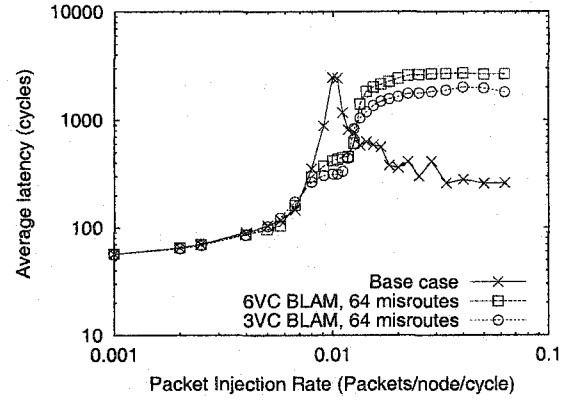


(b) Average Latency vs. Offered Load

### 32-ary, 2-cube (1024 node) Network



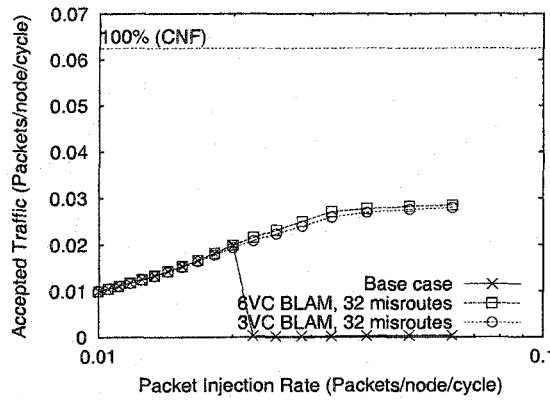
(c) Delivered Throughput vs. Offered Load



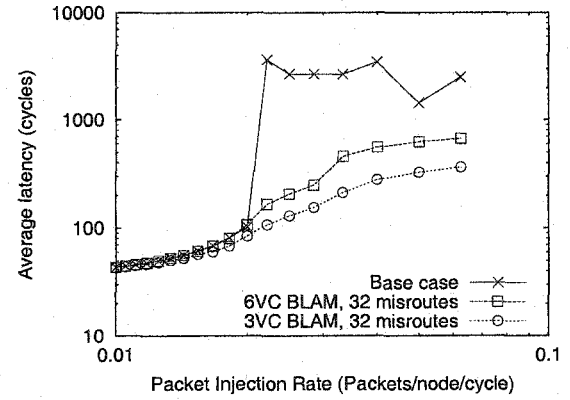
(d) Average Latency vs. Offered Load

**Figure 5.25:** Overall Performance For Other Network Sizes (Bit-Reversal Traffic, 16 flit packets)

### 8-ary, 3-cube (512 node) Network

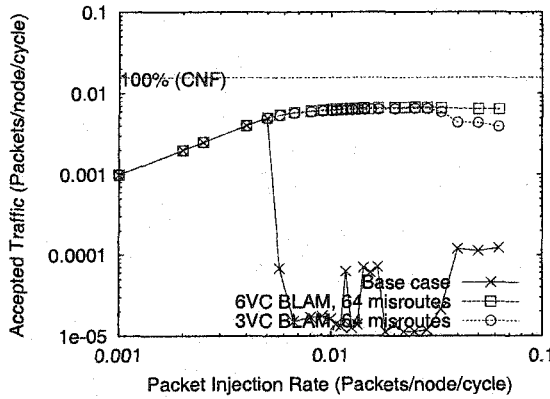


(a) Delivered Throughput vs. Offered Load

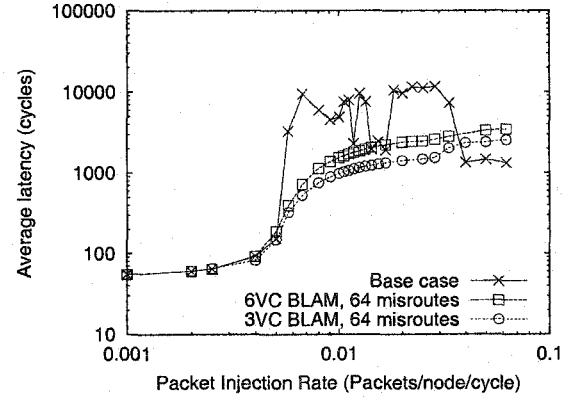


(b) Average Latency vs. Offered Load

### 32-ary, 2-cube (1024 node) Network



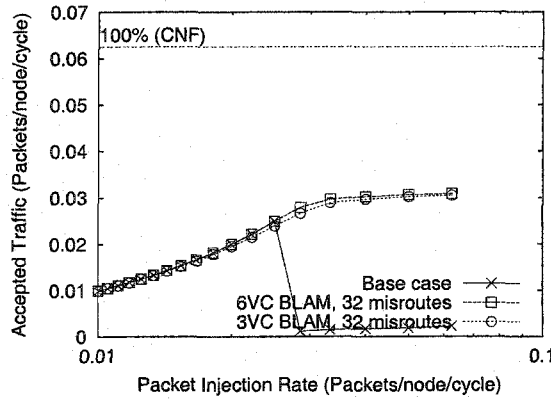
(c) Delivered Throughput vs. Offered Load



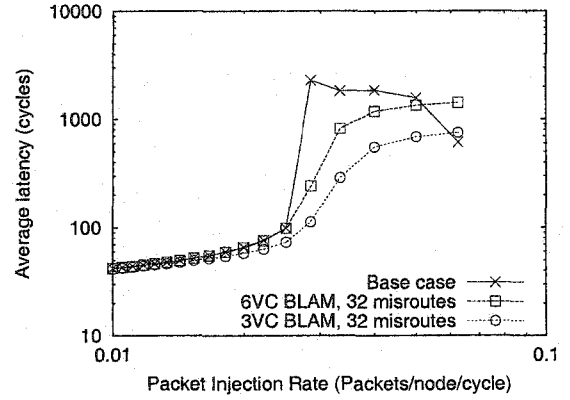
(d) Average Latency vs. Offered Load

**Figure 5.26:** Overall Performance For Other Network Sizes (Complement Traffic, 16 flit packets)

### 8-ary, 3-cube (512 node) Network

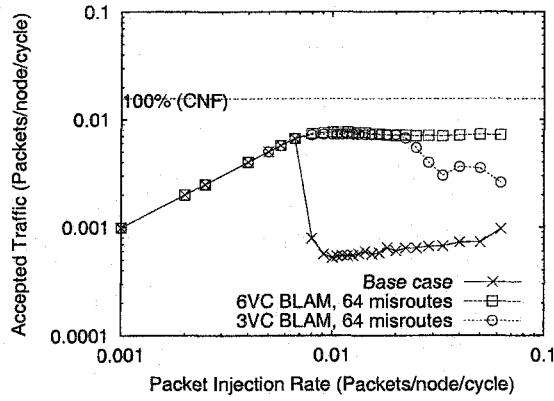


(a) Delivered Throughput vs. Offered Load

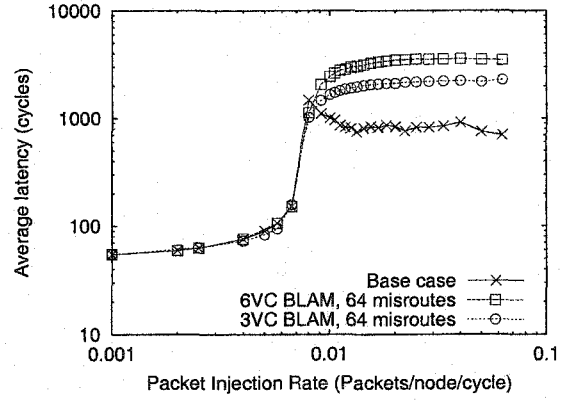


(b) Average Latency vs. Offered Load

### 32-ary, 2-cube (1024 node) Network



(c) Delivered Throughput vs. Offered Load



(d) Average Latency vs. Offered Load

**Figure 5.27:** Overall Performance For Other Network Sizes (Perfect Shuffle Traffic, 16 flit packets)

## Chapter 6:

### Conclusion

Large-scale cache coherent shared memory machines constitute an important and growing portion of the server market. The trend towards more powerful processing power per node results in increased loads on the interconnection networks.

Interconnection network performance is an important component of overall system performance in such systems and demanding server applications expect the interconnection network to provide high-bandwidth, low latency communication. Unfortunately, interconnection networks suffer from known performance problems at high loads due to network saturation.

Previously studied techniques to solve this problem can be broadly classified into two approaches: (a) the congestion control approach and (b) the load balancing approach. The congestion control approach tries to control the offered load to keep the network below saturation. The load balancing approach tries to avoid creation of network “hot-spots” that can quickly propagate due to tree saturation. This thesis proposes and evaluates three techniques to overcome the performance problems at high loads. The first technique falls under the congestion control category and the next two fall under the load balancing category.

#### 6.1 Summary

The first technique—*Tune*—relies on faster feedback and self-tuned source-throttling to keep the network operating. Using proactively propagated global congestion information, *Tune* limits the injection of new packets into the network when it determines that saturation is imminent. This keeps the network operating in its high-throughput, low-latency region of operation. Further, *Tune* also uses global throughput informa-

tion to implement a robust self-tuning mechanism that enables two key features: (a) it is able to recover even if it goes into saturation and (b) it is able to adapt to changes communication patterns.

*Tune* uses an exclusive sideband signaling mechanism to gather global buffer occupancy and throughput information. The buffer occupancy information conveys faster congestion feedback and the throughput information is used to drive the self-tuning mechanism.

Exhaustive simulations with various communications patterns, steady loads and bursty loads show that *Tune* is able to effectively prevent the performance degradation at saturation. Comparing *Tune* to a previously proposed source-throttling mechanism—ALO [4]—that uses local congestion information and fixed thresholds shows that *Tune* is better at preventing performance degradation at saturation. Evaluation of meta packets as an information distribution mechanism shows that the delays in gathering information make it unsuitable for *Tune*.

The second technique—*congestion aware via-routing*—attempts to improve performance by achieving better load balancing in the minimum rectangle by using global congestion aware routing. The evaluation of various flavors of the *via-routing* scheme shows that *via-routing* in the minimum rectangle offers no significant advantage over the load balancing inherent in fully adaptive (minimal) routing. Experiments which assume perfect knowledge of the least congested paths in the minimum rectangle still show significant performance degradation in performance at saturation.

Finally, the third technique is a non-minimal routing algorithm—BLAM—that uses limited, lazy misrouting and bypassing to achieve near-chaos performance with deterministic guarantees of deadlock- and livelock-freedom for virtual cut-through networks.

Experiments show that each component of BLAM (limited misrouting with escape

paths, distributed bypass buffers and lazy misrouting), when used individually, offers little improvement at best and can actually hurt performance in some cases. But the three components acting in concert (as in the BLAM routing algorithm), not only sustain high-bandwidth, low latency network operation but also extend the load at which saturation occurs. The limit on misroutes guarantees livelock-freedom in spite of non-minimal routing and the escape paths guarantee deadlock freedom when the misroutes are exhausted. Lazy misrouting prevents premature misrouting and distributed bypass buffers enables stalling packets to make way for other packets that follow.

## Bibliography

- [1] D. G. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. Morris. The case for resilient overlay networks. In *Proceedings of the 8th Annual Workshop on Hot Topics in Operating Systems*, May 2001.
- [2] D. G. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. Morris. Resilient overlay networks. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, October 2001.
- [3] D. Basak and D.K. Panda. Alleviating Consumption Channel Bottleneck in Wormhole-Routed k-ary n-cube Systems. *IEEE Transactions on Parallel and Distributed Systems*, 9(5):481–496, May 1998.
- [4] E. Baydal, P. Lopez, and J. Duato. A Simple and Efficient Mechanism to Prevent Saturation in Wormhole Networks. In *Proceedings. 14th International Parallel and Distributed Processing Symposium*, pages 617–622, 2000.
- [5] K. Bolding and L. Snyder. Mesh and torus chaotic routing. Technical Report TR-91-04-04, Department of Computer Science, University of Washington, Seattle, 1991.
- [6] L. S. Brakmo and L. L. Peterson. TCP Vegas: End to End Congestion Avoidance on a Global Internet. *Journal of Selected Areas in Communications*, 13(8):1465–1480, October 1995.
- [7] The Chaotic Routing Project, Computer Science and Engineering Department, University of Washington. *Standard for Presentation of Results: Chaos Normal Form and Burton Normal Form*. <http://www.cs.washington.edu/research/projects/lis/chaos/www/presentation.html>.
- [8] The Chaotic Routing Project, Computer Science and Engineering Department, University of Washington, Seattle. *The Chaos Router Simulator*. <http://www.cs.washington.edu/research/projects/lis/chaos/www/simulator.html>.
- [9] A. Charlesworth. The Sun Fireplane Interconnect. *IEEE Micro*, 22(1):36–45, January/February 2002.



- [10] A. A. Chien and M. Konstantinidou. Workloads and Performance Metrics for Evaluating Parallel Interconnects. *IEEE Computer Architecture Technical Committee Newsletter*, pages 23–27, Summer-Fall 1994.
- [11] B. Coates, A. Davis, and K. Stevens. The Post Office Experience: Designing a Large Asynchronous Chip. In *Proceedings of the HICSS*, 1993.
- [12] R. Cutler and S. Atkins. *IBM e-server pSeries 680 Handbook*. IBM, Armonk, N.Y., December 2000. <http://www.redbooks.ibm.com/pubs/pdfs/redbooks/sg246023.pdf>.
- [13] W. J. Dally. Virtual-Channel Flow Control. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):194–205, March 1992.
- [14] W. J. Dally and C. L. Seitz. The TORUS routing chip. *Journal of Distributed Computing*, 1(3):187–196, October 1986.
- [15] W. J. Dally and C. L. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Transactions on Computers*, C-36(5):547–553, May 1987.
- [16] K. Diefendorff. Power4 Focuses on Memory Bandwidth. *Microprocessor Report*, 13(13), October 1999.
- [17] E. W. Dijkstra. A Note on Two Problems in Connection with Graphs. *Numerische Mathematik*, 1:269–271, 1959.
- [18] J. Duato. A New Theory of Deadlock-Free Adaptive Routing in Wormhole Networks. *IEEE Transactions on Parallel and Distributed Systems*, 4(12):1320–1331, December 1993.
- [19] J. Duato, S. Yalamanchili, and L. Ni. *Interconnection Networks: An Engineering Approach*, page 116. IEEE Computer Society Press, 1997.
- [20] J. Duato, S. Yalamanchili, and L. Ni. *Interconnection Networks: An Engineering Approach*, page 405. IEEE Computer Society Press, 1997.

- [21] J. S. Emer. Simultaneous Multithreading: Multiplying Alpha Performance. *Microprocessor Forum*, October 1999.
- [22] J. M. Tendler et al. *IBM e-server POWER4 System Microarchitecture*. IBM, Armonk, N.Y., October 2001. <http://www-1.ibm.com/servers/eserver/pseries/hardware/whitepapers/power4.pdf>.
- [23] 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, April 1991.
- [24] J. Flich, M. P. Malumbres, P. Lopez, and J. Duato. Performance Evaluation of a New Routing Strategy for Irregular Networks with Source Routing. In *Proceedings of the 14th international conference on Supercomputing*, pages 34–43, 2000.
- [25] S. Floyd. TCP and Explicit Congestion Notification. *ACM Computer Communications Review*, 24(5):10–23, October 1994.
- [26] S. Floyd and V. Jacobson. Random Early Detection gateways for Congestion Avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, August 1993.
- [27] M. Fulgham and A. Snyder. A Study of Chaotic Routing with Nonuniform Traffic. Technical Report UW-CSE-93-06-01, University of Washington, June 1993.
- [28] P. T. Gaughan and S. Yalamanchili. Adaptive Routing Protocols for hypercube Interconnection Networks. *IEEE Computer*, 46(2):12–22, 1997.
- [29] K. Gharachorloo, M. Sharma, S. Steely, and S. Van Doren. Architecture and Design of the Alphaserver GS320. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 13–24, November 2000.
- [30] C. J. Glass and L. M. Ni. The Turn Model for Adaptive Routing. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 278–287, May 1992.

- [31] A. G. Greenberg and B. Hajek. Deflection routing in hypercube networks. *IEEE Transactions on Communications*, COM-40(6):1070–1081, June 1992.
- [32] Hewlett-Packard. *Meet the HP Superdome Servers*. [http://www.hp.com/products1/servers/scalableservers/superdome/infolibrary/whitepapers/technical\\_wp.pdf](http://www.hp.com/products1/servers/scalableservers/superdome/infolibrary/whitepapers/technical_wp.pdf).
- [33] V. Jacobson. Congestion Avoidance and Control. In *Proceedings of ACM SIGCOMM '88 Symposium*, pages 314–329, August 1988.
- [34] R. Jain. Congestion Control and Traffic Management in ATM networks: Recent Advances and a Survey. *Computer Networks and ISDN Systems*, October 1996.
- [35] P. Kermani and L. Kleinrock. Virtual Cut-Through : A New Computer Communication Switching technique. *Computer Networks*, 3:267–286, 1979.
- [36] J. H. Kim, Z. Liu, and A. A. Chien. Compressionless Routing: A Framework for Adaptive and Fault-Tolerant Routing. In *Proceedings of the 21st International Symposium on Computer Architecture*, April 1994.
- [37] S. Konstantinidou and L. Snyder. The Chaos Router. *IEEE Transactions on Computers*, 43(12):1386–1397, December 1994.
- [38] Anjan K.V. and T.M. Pinkston. An Efficient, Fully Adaptive Deadlock Recovery Scheme : Disha. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 201–210, June 1995.
- [39] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 241–251, June 1997.
- [40] P. Lopez, J. M. Martinez, and J. Duato. DRIL : Dynamically Reduced Message Injection Limitation Mechanism for Wormhole Networks. In *International Conference on Parallel Processing*, pages 535–542, August 1998.
- [41] P. Lopez, J. M. Martinez, J. Duato, and F. Petrini. On the Reduction of Deadlock Frequency by Limiting Message Injection in Wormhole Networks. In *Proceedings of Parallel Computer Routing and Communication Workshop*, June 1997.

- [42] 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, 1989.
- [43] S. S. Mukherjee, P. Bannon, S. Lang, A. Spink, and D. Webb. The 21364 Network Architecture. *IEEE Micro*, 22(1):26–35, January/February 2002.
- [44] Ted Nesson and S. Lennart Johnsson. ROMM routing on mesh and torus networks. In *Proc. 7th Annual ACM Symposium on Parallel Algorithms and Architectures SPAA '95*, pages 275–287, Santa Barbara, California, 1995.
- [45] J. Y. Ngai and C. L. Seitz. A Framework for Adaptive Routing in Multicomputer Networks. In *Proceedings of the 1st Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 1–9, June 1989.
- [46] A. G. Nowatzky, M. C. Browne, E. J. Kelly, and M. Parkin. S-Connect: from Networks of Workstations to Supercomputer Performance. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 71–82, June 1995.
- [47] L.-S. Peh and W.J. Dally. Flit-Reservation Flow Control. In *Proceedings of the Sixth International Symposium on High Performance Computer Architecture*, pages 73–84, January 2000.
- [48] G. F. Pfister and V. A. Norton. Hot-Spot Contention and Combining in Multistage Interconnection Networks. *IEEE Transactions on Computers*, C-34(10):943–948, October 1985.
- [49] V. Puente, R. Beivide, J. Gregorio, J. Prellezo, J. Duato, and C. Izu. Adaptive Bubble Router: A Design to Improve Performance in Torus Networks. In *Proceedings of the International Conference on Parallel Processing*, pages 58–67, 1999.
- [50] K.K. Ramakrishnan and R. Jain. A Binary Feedback Scheme for Congestion Avoidance in Computer Networks. *ACM Transactions on Computer Systems*, 8(2):158–181, 1990.
- [51] P. Ranganathan, K. Gharachorloo, S. V. Adve, and L. A. Barroso. Performance of Database Workloads on Shared Memory Systems with Out-of-Order Proces-

- sors. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 307–318, October 1998.
- [52] S. Scott and G. Sohi. The Use of Feedback in Multiprocessors and its Application to Tree Saturation Control. *IEEE Transactions on Parallel and Distributed Systems*, 1(4):385–398, October 1990.
  - [53] S. L. Scott. Synchronization and Communication in the T3E Multiprocessor. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 26–36, October 1996.
  - [54] Silicon Graphics. *SGI 3000 Family Reference Guide*. [http://www.sgi.com/origin/3000/3000\\_ref.pdf](http://www.sgi.com/origin/3000/3000_ref.pdf).
  - [55] A. Singh, W. J. Dally and B. Towles, and A. K. Gupta. Locality-preserving randomized oblivious routing on torus networks. In *Proceedings of the 12th Annual ACM Symposium on Parallel Algorithms and Architectures*, August 2002.
  - [56] A. Smai and L. Thorelli. Global Reactive Congestion Control in Multicomputer Networks. In *5th International Conference on High Performance Computing*, pages 179–186, 1998.
  - [57] B. J. Smith. Architecture and Applications of the HEP Multiprocessor Computer System. In *Proceedings of SPIE*, pages 241–248, 1981.
  - [58] 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.
  - [59] C. B. Stunkel, D. G. Shea, and B. Abali et al. The SP2 High Performance Switch. *IBM Systems Journal*, 34(2):185–204, 1995.
  - [60] The Superior Multiprocessor ARchiTecture (SMART) Interconnects Group, Electrical Engineering - Systems Department, University of Southern California. *FlexSim*. <http://www.usc.edu/dept/ceng/pinkston/tools.html>.
  - [61] M. Thottethodi, A. R. Lebeck, and S. S. Mukherjee. Self-Tuned Congestion Control for Multiprocessor Networks. In *Proceedings of the Seventh International*

*Symposium on High Performance Computer Architecture (HPCA-7)*, pages 107–118, January 2001.

- [62] B. Towles and W. J. Dally. Worst-case Traffic for Oblivious Routing Functions. *Computer Architecture Letters*, 1, February 2002.
- [63] J. Upadhyay, V. Varavithya, and P. Mohapatra. A Traffic Balanced Adaptive wormhole routing scheme for Two-Dimensional Meshes. *IEEE Transactions on Computers*, pages 190–197, May 1997.
- [64] L. G. Valiant. A scheme for fast parallel communication. *SIAM Journal on Computing*, 11(2):350–361, May 1982.

## Biography

Mithuna Thottethodi was born in Bangalore, India on June 13, 1974. He received his Bachelor of Technology (Honours) degree in June 1996 in the field of Computer Science and Engineering from the Indian Institute of Technology (IIT), Kharagpur, India. He entered the graduate program at Duke University in August 1996 and received his Ph.D. degree in Computer Science in December, 2002. He is the primary author of “Self-Tuned Congestion Control for Multiprocessor Networks” which appeared in the proceedings of the Seventh International Symposium on High Performance Computer Architecture (HPCA-7, 2001) and “Tuning Strassen’s Matrix Multiplication For Memory Efficiency” which appeared in the proceedings of the Supercomputing ’98 conference. He is also a co-author of “Nonlinear Array Layouts for Hierarchical Memory Systems” (ICS ’99), “Recursive Array Layouts and Fast Matrix Multiplication” (SPAA ’99), “Annotated Memory References: A Mechanism for Informed Cache Management” (Europar ’99) and “Recursive Array Layouts and Fast Matrix Multiplication” (IEEE Transactions on Parallel and Distributed Systems, November 2002).