LOAD-BALANCED ROUTING IN INTERCONNECTION NETWORKS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Arjun Singh
March 2005

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

William J. Dally
(Principal Adviser)

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Nick Mckeown

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Nicholas Bambos

Approved for the University Committee on Graduate Studies.

Abstract

Interconnection networks enable fast data communication between components of a digital system. Today, interconnection networks are used in a variety of applications such as switch and router fabrics, processor-memory interconnect, I/O interconnect, and on-chip networks, to name a few.

The design of an interconnection network has three aspects — the topology, the routing algorithm used, and the flow control mechanism employed. The topology is chosen to exploit the characteristics of the available packaging technology to meet the requirements (bandwidth, latency, scalability, etc.) of the application, at a minimum cost. Once the topology of the network is fixed, so are the bounds on its performance. For instance, the topology determines the maximum throughput (in bits/s) and zero-load latency (in hops) of the network. The routing algorithm and flow control must then strive to achieve these performance bounds.

The function of a routing algorithm is to select a path to route a packet from its source to its destination. In this thesis, we demonstrate the significance of the routing algorithm used in the network towards achieving the performance bounds set by the topology. Central to this thesis, is the idea of load-balancing the network channels. A naive routing algorithm that does not distribute load evenly over all channels, stands to suffer from sub-optimal worst-case performance. However, unnecessary load-balancing is overkill. Spreading traffic over all channels when there is no uneven distribution of traffic, leads to sub-optimal best-case and average-case performance. This thesis explores routing algorithms that strive to achieve high worst-case efficiency without sacrificing performance in the average or best-case.

While performance metrics such as average latency and worst-case throughput are key

parameters in evaluating a network, there are several other important measures such as amount of packet reordering, statistical guarantees on delay and network buffer occupancy, to name a few. In the last part of this thesis, we propose a method to analyze the performance of a class of load-balanced networks over these performance metrics.

Acknowledgements

I am grateful above all to my advisor, Bill Dally, whose enthusiasm and insight have made research a most enjoyable and fulfilling experience. His phenomenal breadth of knowledge and ability to discern the key points of any research problem, continue to inspire me. I am also grateful to my thesis and orals committee members, Professors McKeown, Bambos, Peumans, and Rosenblum for their invaluable comments and feedback on my work.

I would like to thank the members of the Concurrent VLSI Architecture (CVA) group, for their help and support throughout these years. I especially enjoyed the stimulating brainstorming sessions with the members of the interconnect subgroup (Amit, Brian, Jin, and John). I would also like to acknowledge Mattan, Nuwan, and Jung-Ho for being extremely supportive and understanding office mates.

For all the fun times at Stanford, I would like to thank all my friends — Indu, for her support over the last few years, Datta, Sriram, Sahoo, Niloy, and Amal, for having been there to help me in times of need. I also thank Sriram and Amit for proof reading parts of my dissertation. Finally, I thank my father, mother, and sister, who have been a pillar of strength for me and without whose encouragement none of this would have been possible.

This dissertation is dedicated to the loving memory of my departed mother who continues to guide me in spirit.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 A brief introduction to interconnection networks

> *An interconnection network is a programmable system that enables fast data communication between components of a digital system. The network is programmable in the sense that it enables different connections at different points in time. The network is a system because it is composed of many components: buffers, channels, switches, and controls that work together to deliver data.*
>
> *— Principles and Practices of Interconnection Networks (Chapter 1.1), Dally & Towles [12]*

The functional view of an interconnection network is illustrated in Figure 1.1. Eight terminal nodes are connected to the network with bidirectional channels. When a source terminal (say T_0) wants to communicate with a destination terminal (say T_4), it sends data in the form of a *message* into the network and the network delivers the message to T_4. Using the same resources, the network can deliver the above message in one cycle, and a different message in the next cycle.

The interconnection network of Figure 1.1 may be realized in several ways. One approach is to provision the system such that there is a possible point-to-point connection between every pair of terminals. As illustrated in Figure 1.2, one way of implementing

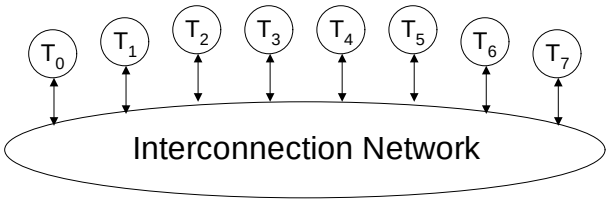


Figure 1.1: The functional view of an interconnection network. Terminals T_0 through T_7 are connected to the network with bidirectional channels.

such an arrangement is by connecting the terminals to a crossbar switch. At every cycle, the crossbar connects each source terminal with one distinct destination terminal.

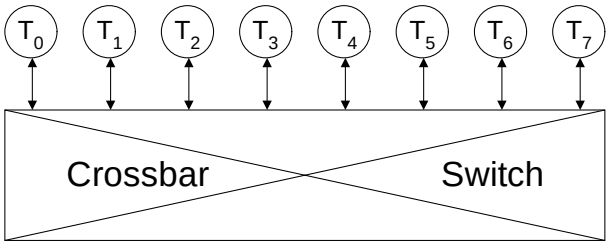


Figure 1.2: Realizing the 8-node network using a crossbar switch

An alternative way to implement the network may be to connect each terminal, T_i, to a *router* node, R_i, and connect the router nodes in a ring using bidirectional channels (Figure 1.3). While this implementation connects the terminals with much less wiring than may be required in the crossbar realization, the drawback is that all terminals no longer have point-to-point connections. In our example, T_0 can now send data to T_4 either through nodes R_0, R_1, R_2, R_3, and R_4 or through R_0, R_7, R_6, R_5, and R_4. For this reason, this implementation is called a *multi-hop* network.

Irrespective of the way the interconnection network is realized, the network itself plays a vital role in determining the performance of the system as a whole. We next examine some digital systems wherein we might encounter interconnection networks and explain why the network's performance is a key component of the performance of the system as a whole.

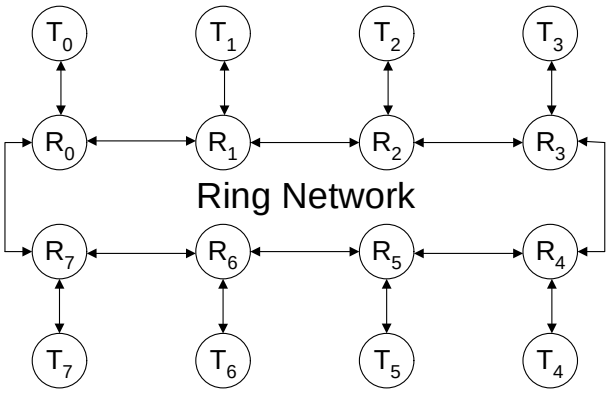


Figure 1.3: Connecting the 8 nodes in a ring

1.2 The importance of interconnection networks

Today, interconnection networks are used in almost all digital systems that have two or more components to connect. In a computer system, the "terminal nodes" from the previous section could be processors and memories, or I/O devices and controllers, communicating with each other. They could also be input and output ports in the case of communication switches and network routers. Interconnection networks may also connect sensors and actuators to processors in control systems, host and disk nodes in I/O networks and on-chip cores in chip multiprocessors.

The performance of most digital systems today is limited by their communication or interconnection, not by their logic or memory. Hence, it is imperative that the underlying interconnection network perform efficiently to improve the efficacy of the entire system. For instance, in a computer system, the interconnection network between processor and memory determines key performance factors such as the memory latency and memory bandwidth. The performance of the interconnection network in a communication switch largely determines the capacity (data rate and number of ports) of the switch.

The performance of an interconnection network can be measured using a rich set of metrics. The most common metrics are throughput and latency. Other important metrics include reliability, graceful degradation in the presence of faults, in-order delivery of data

packets, and delay guarantees in communicating data. To meet the performance specifications of a particular application, the *topology*, *routing*, and *flow control* of the network must be implemented. The topology of the network refers to the arrangement of the shared set of nodes and channels. Once a topology has been chosen, the routing algorithm determines a path (sequence of nodes and channels) a message takes through the network to reach its destination. Finally, flow control manages the allocation of resources to packets as they progress along their route. The topology sets limits on the performance of the network while the routing and flow-control strive to realize these performance limits.

In this thesis, we focus on the routing of packets through the network. In other words, given the topology, and assuming ideal flow control, we attempt to route packets through the network to give high performance on several metrics.

1.3 The need for load-balanced routing

The routing method employed by a network determines the path taken by a packet from a source terminal node to a destination terminal node. Networks with high *path diversity* offer many alternative paths between a source and destination. *Oblivious* routing algorithms choose between these paths based solely on the identity of the source and destination of the message while *adaptive* algorithms may base their decision on the state of the network.

A good routing algorithm makes its route selection in a manner that exploits locality to provide low latency and high throughput on *benign* traffic. Many applications also require the interconnection network to provide high throughput on *adversarial* traffic patterns. In an Internet router, for example, there is no backpressure on input channels. Hence, the interconnection network used for the router fabric must handle any traffic pattern, even the worst-case, at the line rate, or else packets will be dropped. To meet their specifications, I/O networks must provide guaranteed throughput on all traffic patterns between host and disk nodes. Some multicomputer applications are characterized by *random permutation traffic*[1]. This arises when operating on an irregular graph structure or on a regular structure

[1] *Random permutation traffic* in which each node sends all messages to a single, randomly-selected node should not be confused with *random traffic* in which each message is sent to a different randomly selected node.

that is randomly mapped to the nodes of the machine. Performance on these applications is limited by the throughput of the network on adversarial patterns.

A routing algorithm must strike a balance between the conflicting goals of exploiting locality on benign traffic while load-balancing on adversarial traffic. To achieve high performance on benign traffic, *Minimal Routing* (MR) — that chooses a shortest path for each packet — is favored. MR, however, performs poorly on worst-case traffic due to load imbalance. With MR, an adversarial traffic pattern can load some links very heavily while leaving others idle. To improve performance under worst-case traffic, a routing algorithm must balance load by sending some fraction of traffic over non-minimal paths hence, destroying some of the locality.

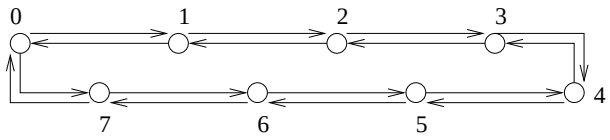


Figure 1.4: The 8 node ring with unidirectional channels

In order to better understand the tradeoffs inherent in minimally routing a benign and a difficult traffic pattern, we revisit our example of the 8 node ring from Figure 1.3. For brevity, we drop the terminal nodes from our discussion and concentrate solely on the router nodes and the channels. For the sake of simplicity, we also consider each bidirectional channel in the figure as two unidirectional channels, one for each direction. Moreover, each channel has a bandwidth of b bits/s. Figure 1.4 shows the simplified version of the 8 node ring. We consider minimally routing a benign and a hard traffic pattern on this ring.

For the benign pattern, nearest neighbor (NN), all traffic from node i is distributed equally amongst its neighboring nodes (half to node $i+1$[2] and half to node $i-1$). Since the channels have bandwidth b, each node can inject traffic at an optimal rate of $2b$ (throughput, $\theta = 2b$) if traffic is simply routed along minimal paths. We call such a traffic *benign* as all the channels are naturally load balanced if MR is used to route packets.

For the difficult pattern, we consider the worst-case traffic for MR on a ring, tornado (TOR) traffic. In TOR, all traffic from node i is sent nearly half-way-around the ring, to

[2] All operations on node ids are done modulo the total number of nodes.

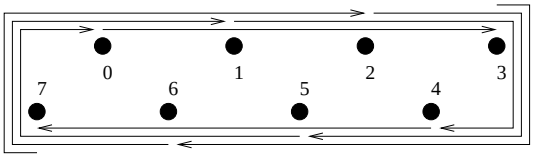


Figure 1.5: Minimally routed tornado traffic. Clockwise link load is 3. Counter clockwise link load is 0.

node $i + 3$. Figure 1.5 shows how minimally routing TOR leads to high load on the clockwise channels keeping the counter clockwise channels completely idle. This results in considerable load imbalance leading to a poor throughput of $\theta = b/3$ (62.3% of the optimal throughput as we shall demonstrate in the subsequent chapters[3]). Achieving good performance on adversarial patterns such as TOR requires routing some traffic non-minimally (the long way around the ring) to balance the load.

We could balance TOR traffic by completely randomizing the routing (VAL) as suggested by Valiant [50], sending from node i to a random intermediate node j and then from j to $i + 3$. Each of these two phases is a perfectly random route and hence uses $8/4 = 2$ links on average for a total of 4 links traversed per packet. These links are evenly divided between the clockwise and counterclockwise rings, two each. Thus, even though we are traversing one more link on average than for minimal routing, the per-node throughput for VAL is higher on TOR, $\theta = b/2$. The problem with purely randomized routing is that it destroys locality. For the NN traffic pattern, throughput is still $b/2$ while MR gives $\theta = 2b$.

Through the most part of this thesis, we shall propose routing algorithms that strive to achieve good worst-case performance without sacrificing the locality inherent in benign traffic.

1.4 Load-balancing in networking

Load balancing has been of interest in the networking community over the last few years. As the line rates continue to increase, load balancing plays a key role in building high

[3]This throughput asymptotically degrades to $b/4$ (50% of optimal) as the number of nodes becomes large.

speed interconnection networks within Internet routers [4] and also guaranteeing high performance in the network backbone [54].

In Internet routers, the worst-case throughput largely determines the capacity of the router [12, 49] since the worst-case throughput effectively determines the maximum load that the fabric is guaranteed to handle. Load-balancing has proved to be a useful technique to ensure high guaranteed throughput delivered by the router fabric.

Load-balanced switches were first introduced as Birkhoff-von Neumann (BN) switches by Chang et al [4, 5]. These switches consist of two back-to-back crossbar stages. Similar to Valiant's load balancing scheme, the first stage load balances the incoming traffic pattern by spreading traffic uniformly and at random among the first stage's output ports. The second stage then delivers traffic to the actual destination port. Based on the BN switch, a lot of work has been done to incorporate efficient load balancing within Internet switches [20, 21].

Recently, Keslassy et al [21] showed that using a fully connected graph topology with Valiant's two phase routing ensures optimal guaranteed throughput. From the point of view of worst-case throughput, this is the optimal load balancing scheme. However, this scheme does not optimize either the throughput on benign traffic patterns or the latency of packet delivery. The load balancing routing algorithms in this thesis strive to achieve high worst-case throughput without sacrificing performance in the average or best-case, while delivering packets with low latency.

1.5 Contributions

This thesis includes several key contributions to the study of load-balanced routing on interconnection networks.

We first concentrate on a popular interconnection topology, torus or k-ary n cube networks and design efficient routing algorithms for them:

- *RLB: Locality-preserving Randomized Oblivious routing* [43]. We propose efficient oblivious routing algorithms that provide high worst-case and average-case performance while sacrificing modest performance in the best case. However, they do not give optimal worst-case performance for generic torus networks.

- *GOAL: Globally Oblivious Adaptive Locally routing* [41]. GOAL improves the worst case performance of RLB by introducing adaptivity in its path selection. GOAL gives optimal worst-case performance but suffers from sub-optimal best-case behavior.

- *GAL: Globally Adaptive Load-balanced routing* [44, 42]. GAL senses global congestion adaptively and alleviates GOAL's problem of sacrificing performance in the best case. GAL is the only known algorithm that gives optimal worst-case and best-case performance compared to all known routing algorithms for torus networks.

From torus networks, we extend our focus to generic regular topologies.

- *UGAL: Universal Globally Adaptive Load-balanced routing* [40]. We extend the concepts of GAL to propose a universal routing scheme on any regular topology. UGAL gives provably optimal worst case performance without sacrificing any performance in the best-case.

Finally, our focus shifts to other performance metrics such as delay and buffer occupancy guarantees and packet reordering in load-balanced, high radix interconnection networks.

- *Delay and buffer bounds in high radix interconnection networks* [39]. We analyze the buffer occupancy, end-to-end delay, and packet reordering in load-balanced high radix interconnection networks. Our results suggest that modest speedups and buffer depths enable reliable networks without flow control to be constructed.

1.6 Outline

For the remainder of this thesis, we first focus on efficient oblivious and adaptive routing on torus or k-ary n cube networks. Chapter 2 introduces preliminary notations that will be used throughout the thesis. Chapters 3, 4, and 5 explore efficient algorithms designed to load balance torus networks. Chapter 6 builds on the concepts used for routing on tori to propose a universal routing algorithm on a generic regular topology that load balances the network without sacrificing locality. Finally, Chapter 7 addresses the problem of bounding the buffer

occupancy and end-to-end delay in load-balanced, high radix interconnection networks. Conclusions and future work are presented in Chapter 8. A method for finding worst-case traffic patterns in presented in Appendix A. Appendix B proves that GOAL is deadlock free. Throughput results for a 16-ary 2-cube network are presented in Appendix C. An expression for the probability of a queue being empty is derived in Appendix D.

Chapter 2

Preliminaries

Before we address the problem of designing load-balanced routing algorithms, we list our assumptions and provide a description of the network model. First, we abstract a topology as a graph — a collection of router nodes connected by directed edges (channels). The set of nodes is denoted as $\mathcal{N}$ and the size of this set is $N = |\mathcal{N}|$. Likewise, the set of channels is $\mathcal{C}$ and its size is $C = |\mathcal{C}|$. Each channel has a bandwidth of b bits/s. The *bisection bandwidth*, B, of the topology is the sum of the bandwidth of the channels that cross a minimum bisection (a cut that partitions the entire graph nearly in half) of the graph. Without loss of generality, each of the router nodes has an associated terminal node. We do not explicitly include these terminal nodes in our figures to avoid unnecessary clutter. Data is injected by the terminal nodes in *packets* which are further divided into fixed-sized flow control units called *flits*. Since this thesis is about routing, we assume ideal flow control and concentrate on single-flit sized packets.

In the rest of this chapter, we first formalize a few terms that will be used throughout this thesis. We shall then put these terms in perspective by briefly explaining how the performance of interconnection networks is typically measured. We then present a bound on the worst-case throughput for a network. Finally, we introduce torus networks and give a brief description of some traffic patterns we shall use on these networks.

2.1 Definitions

Injection rate (Offered load) (α): The injection rate, α, is the amount of data that each terminal node injects into the network on average.

Traffic matrix (Traffic pattern) (Λ): The destination of the traffic injected by each terminal node is given by a $N \times N$ traffic matrix, Λ. Each entry, λ_{ij}, of Λ corresponds to the fraction of traffic injected from terminal node i destined for terminal node j. It follows that the entries in each row add to 1. A node is said to be *oversubscribed* if the sum of its column entries in Λ is greater than 1. A traffic matrix is *admissible* if none of the nodes is oversubscribed.

Permutation matrix (Π): A traffic matrix Λ with a single 1 entry in each row and column. All other entries are 0. Each source node sends traffic to a single distinct destination node.

Uniform Random traffic (UR): A traffic matrix in which every node sends data uniformly and at random to every destination. All entries in Λ equal $1/N$.

Routing algorithm (R): A routing algorithm maps a source-destination pair to a path through the network from the source to the destination. *Oblivious* algorithms select the path using only the identity of the source and destination nodes. *Adaptive* algorithms may also base routing decisions on the state of the network. Both oblivious and adaptive algorithms may use randomization to select among alternative paths. *Minimal* algorithms route all packets along some shortest path from source to destination, while *non-minimal* ones may route packets along longer paths.

Channel load ($\gamma_c(R, \alpha, \Lambda)$): The amount of data crossing channel c on average when routing traffic matrix Λ at an injection rate α, with routing algorithm R.

Throughput ($\Theta(R, \Lambda)$): The maximum α that the terminal nodes can inject such that no channel is *saturated*, or is required to deliver more data than its bandwidth supports.

Capacity ($\Theta(UR)$): The ideal throughput for UR traffic, given by $2B/N$. Throughout this thesis, we shall normalize both the throughput and the offered load to the *capacity* of

the network.

Worst-case throughput ($\Theta_{wc}(R)$): The worst-case throughput for routing algorithm R over the space of all *admissible* traffic matrices,

$$\Theta_{wc}(R) = \min_{\Lambda} \Theta(R, \Lambda)$$

2.2 Performance measures

We place the terms defined above in perspective and briefly explain the basics of evaluating the performance of a routing algorithm. For a given traffic pattern, Λ, and routing algorithm, R, we can describe the performance of an interconnection network with two graphs as shown in Figures 2.1 and 2.2. The steady-state performance of the network at offered loads below the saturation point is described by Figure 2.1 which shows the average latency per packet plotted against the offered load, α. The zero-load latency or hop-count, H, is the y-intercept of this curve and the throughput, Θ, is the x-coordinate of the asymptote. At offered loads greater than Θ, average steady-state latency is infinite.

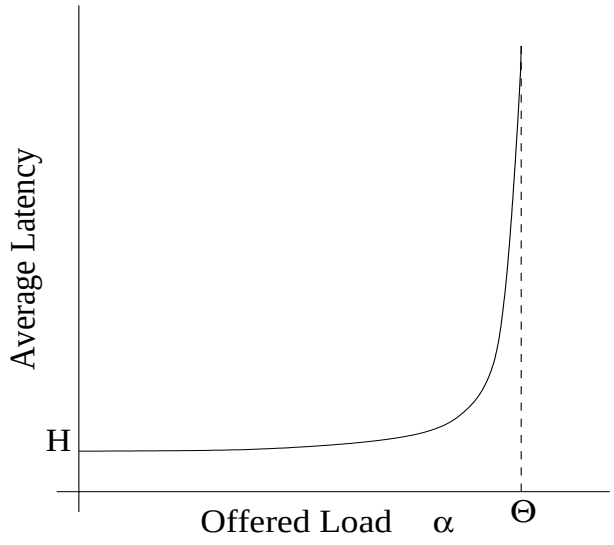


Figure 2.1: Notational latency vs. offered load graph

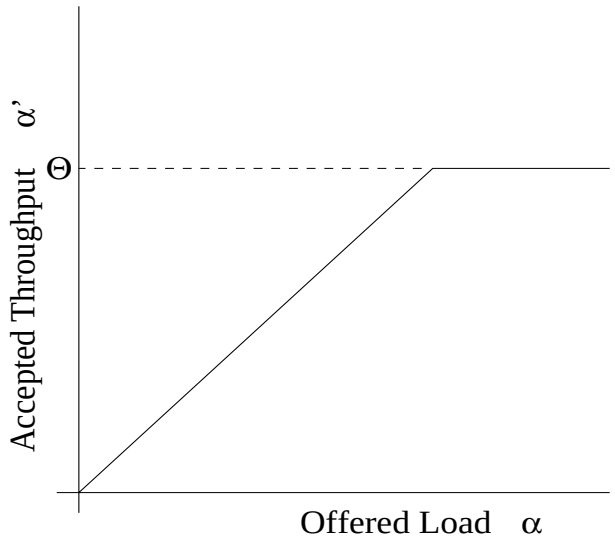


Figure 2.2: Accepted throughput vs. offered load

Figure 2.2, which shows a plot of accepted traffic, α', as a function of offered load, α, describes network performance after the saturation point, when $\alpha > \Theta$. We report the minimum accepted traffic over all source-destination pairs to reflect the throughput achieved for the specified traffic matrix Λ. Under heavy load, the source-destination pairs with less contention deliver more packets than other pairs. In effect, these pairs *get ahead* of the other pairs. However, the amount of the desired destination matrix, Λ, that is delivered is governed by the *slowest* pair (the one with the least accepted throughput).

To illustrate the need to compute α' as the minimum across all source-destination pairs instead of the average accepted load, consider the 8 node ring network routing three flows as shown in Figure 2.3. There are three source nodes — 0, 1 and 5 — sending packets to destination nodes 3, 2 and 6 respectively. All source nodes inject the same load, α, into the network. As we increase the injected load for each node from zero up to the saturation point, $\alpha = 0.5$, none of the links in the network are saturated. Until this point, the average accepted load, α^*, and the minimum accepted load, α', across the flows are both the same, i.e. $\alpha^* = \alpha' = \alpha$. Suppose we now increase α to 1.0. Link $1 \rightarrow 2$ becomes saturated and allocates half of its capacity to each of the two flows $0 \rightarrow 3$ and $1 \rightarrow 2$. However, link $5 \rightarrow 6$ offers its full capacity to flow $5 \rightarrow 6$. The accepted loads for nodes 2, 3 and

6 are therefore 0.5, 0.5 and 1.0 respectively. Hence, $\alpha^* = 2/3 = 0.67$ while $\alpha' = 0.5$. The minimum number, α', reflects the amount of the original traffic matrix, Λ, that is being delivered. The extra traffic on $5 \rightarrow 6$ represents additional traffic that is not part of the specified destination matrix.

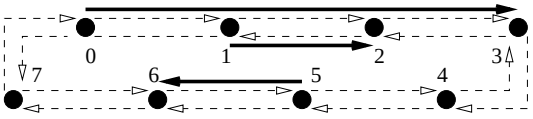


Figure 2.3: Minimally routed pattern $0 \rightarrow 3$, $1 \rightarrow 2$, $5 \rightarrow 6$

Returning to Figure 2.2, at offered loads less than the saturation point, all traffic is delivered so $\alpha'(\alpha) = \alpha$. Beyond the saturation point, accepted traffic is flat for a *stable* routing algorithm, i.e. $\alpha'(\alpha) = \Theta$ for $\alpha \geq \Theta$. For *unstable* algorithms, throughput degrades beyond saturation. This occurs for some non-minimal algorithms wherein, due to congestion, average path length increases with offered traffic. Instability may also occur when *global fairness* is not maintained, and hence the throughput of the slowest source-destination pair is reduced after saturation because more of a critical shared resource is being granted to a faster source-destination pair.

2.3 Performance bounds

Before we examine routing algorithms with high worst-case throughput, it is instructive to set an upper bound on the worst-case throughput of any network and prove that there is an algorithm that achieves this bound.

Theorem 1. *The worst-case throughput is no greater than half the network capacity.*

Proof. Consider a network of N nodes with bisection bandwidth, B. For simplicity, assume N is even. There are $N/2$ nodes on either side of the bisection. With UR traffic, each node sends half its traffic across the channels in the bisection. Hence, the maximum traffic each node can send for UR (network capacity) is $\Theta(UR) = 2B/N$.

Next, consider the traffic pattern where each node sends *all* its traffic across the bisection. We call this traffic pattern *diameter traffic* (DIA). The maximum traffic each node can

send for DIA without saturating the bisection is B/N. Hence, for any routing algorithm, R_*, $\Theta(R_*, DIA) = B/N = 50\%$ of capacity. It follows that $\Theta_{wc}(R_*) \leq 0.5$.

□

Previous work has attempted to address the issue of providing high worst-case performance. As briefly discussed in Chapter 1, Valiant's randomized algorithm (VAL) is a two-phase randomized routing algorithm. In the first phase, packets are routed from the source to a randomly chosen intermediate node using minimal routing. The second phase routes minimally from the intermediate node to the destination. We now prove that VAL gives optimal worst-case performance on regular networks. However, this is at the expense of completely destroying locality, and hence giving very poor performance on local traffic.

Theorem 2. *VAL gives optimal worst-case throughput but performs identically on every admissible traffic matrix on regular topologies.*

Proof. Each phase of VAL is equivalent to minimally routing UR traffic across the bisection. If each node injects α, then the load on the bisection is $N\alpha$ ($N\alpha/2$ due to each of the 2 phases occurring simultaneously). Since VAL balances this load equally over all channels of the bisection, the maximum α required to saturate the bandwidth is B/N (50% of capacity). Moreover, the analysis is identical for every traffic giving $\Theta(\mathrm{VAL}) = 0.5$. □

In the next three chapters, we propose load-balanced oblivious and adaptive routing algorithms on a popular topology called *torus* networks. Before we present our algorithms, we introduce torus networks and discuss previous work done in routing on such networks.

2.4 Torus networks

In Chapter 1, we had briefly introduced a ring network. Torus networks can be thought of as rings of rings in multiple dimensions. Formally, torus networks are referred to as k-ary n-cube networks (see Chapter 5 of [12]). A k-ary 1-cube is simply a ring (1-dimensional torus) of k nodes. In general, an n-dimensional, radix-k torus, or k-ary n-cube, consists of $N = k^n$ nodes arranged in an n-dimensional cube with k nodes along each dimension. Each of these N nodes serves simultaneously as an input terminal, output terminal, and

a switching node of the network. Channels connect nodes whose addresses differ by ± 1 in one dimension modulo k giving a total of $2nN$ channels in the network. Figure 2.4 shows a 4-ary 2-cube or a 4×4 torus network. In general, an arbitrary k-ary n-cube can be constructed by adding dimensions iteratively. As illustrated in Figure 2.5, a k-ary n-cube can be formed by combining k k-ary $(n-1)$-cubes. From this figure, it is easy to see that if channel bandwidth in each direction is b, the bisection bandwidth of a k-ary n-cube is $B = 4k^{n-1}b$.

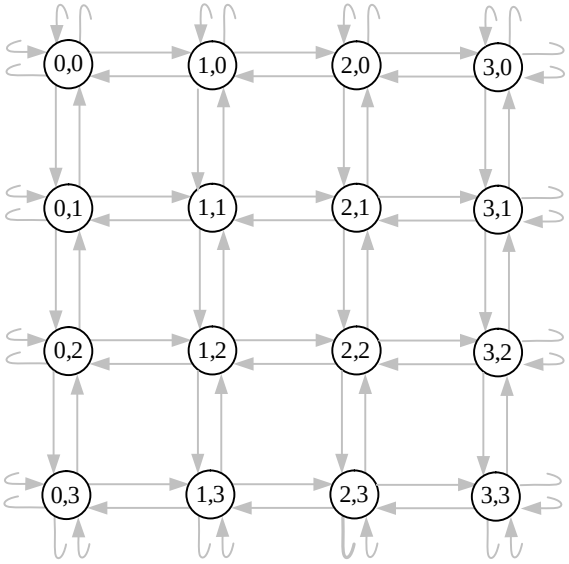


Figure 2.4: A 4-ary 2-cube (4×4 torus) network

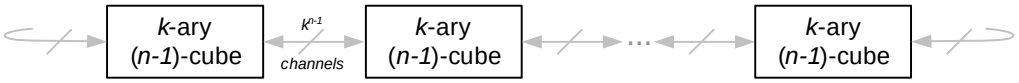


Figure 2.5: Constructing a k-ary n-cube from k k-ary $(n-1)$-cubes

Torus networks span a range of networks from rings ($n = 1$) to hypercubes ($k = 2$). These networks are attractive for several reasons. Their regular physical arrangement is well suited to packaging constraints. At low dimensions, tori have uniformly short wires allowing high-speed operation without repeaters. Since logically minimal paths in tori are almost always physically minimal as well, such networks can exploit physical locality

between communicating nodes. Moreover, tori are regular (all nodes have the same degree), offer high path diversity, and are also edge-symmetric, which helps to improve load balance across the channels. For these reasons, interconnection networks based on k-ary n-cube topologies are widely used as switch and router fabrics [8], for processor-memory interconnect [37], and for I/O interconnect [34].

2.5 Previous work on routing on torus networks

Several routing algorithms, both oblivious and adaptive, have been proposed in literature for torus networks. Table 2.1 presents a brief description of five previously published algorithms that we shall compare our load-balanced algorithms with.

Table 2.1: Previous routing algorithms for torus networks.

Name	Description
DOR	Dimension-order routing: deterministically route along a fixed minimal path, routing completely in the most significant unmatched dimension before moving on to the next dimension [45].
VAL	Valiant's algorithm: route to a random node, q (phase 1), anywhere in the network, then to the destination (phase 2) [50].
ROMM	Two-phase ROMM: route to a random node, q, in the minimal quadrant, then to destination [31].
MIN AD	Minimal Adaptive (or the $*$-channels algorithm): always route in the minimal quadrant, routing adaptively within it [16].
CHAOS	The CHAOS routing algorithm [25].

The first three rows of the table describe oblivious algorithms — DOR, VAL, and ROMM. Dimension-order routing (DOR), sometimes called e-cube routing, was first reported by Sullivan and Bashkow [45]. In DOR, each packet first traverses the dimensions one at a time, arriving at the correct coordinate in each dimension before proceeding to the next. Because of its simplicity, it has been used in a large number of interconnection networks [11, 37]. The poor performance of DOR on adversarial traffic patterns motivated much work on adaptive routing.

Valiant first described how to use randomization to provide guaranteed throughput for an arbitrary traffic pattern [50]. As discussed before, his method, VAL, perfectly balances load by routing to a randomly selected intermediate node (phase 1) before proceeding to the destination (phase 2). DOR is used during both phases. While effective in giving high guaranteed performance on worst-case patterns, this algorithm destroys locality — giving poor performance on local or even average traffic.

In order to preserve locality while gaining the advantages of randomization, Nesson and Johnson suggested ROMM [31] — Randomized, Oblivious, Multi-phase Minimal routing. Like VAL, ROMM routes each packet via an intermediate node chosen randomly from the minimal *quadrant*[1], ensuring that the resulting path is strictly minimal. While [31] reports good results on a few permutations, we demonstrate in Chapter 3 that ROMM actually has lower worst-case throughput than DOR. The problem is that with minimal routing, it is impossible to achieve good load balance on adversarial patterns.

The last two rows of Table 2.1 describe two adaptive algorithms — MIN AD and CHAOS. In order to address the issue of providing high worst-case performance while preserving locality, Minimal adaptive routing algorithms (MIN AD) [16, 27] route minimally to preserve locality but adaptively within quadrants to avoid local congestion. However, since they are minimal, they cannot avoid sub-optimal throughput on traffic that causes global load imbalance.

Non-minimal adaptive routing algorithms attempt to avoid throughput degradation on adversarial traffic patterns by incorporating non-minimal paths. However, the decision to misroute a packet is local and does not ameliorate the global imbalance created by adversarial traffic. For example, in Chaotic routing [3] (CHAOS), packets always choose a minimal direction, if available. Otherwise, they are momentarily buffered in a shared queue and eventually misrouted in a non-minimal direction (further from their destination). However, the decision to misroute does not incorporate any concept of global load balance and therefore, CHAOS, like MIN AD, suffers from low throughput on adversarial traffic patterns.

[1] A *quadrant* consists of all paths from a given source node that travel in a single direction in each dimension. For example, in a 2-D torus, the $(+,-)$ quadrant consists of all paths that travel in the $+x$ and $-y$ directions.

2.6 Traffic patterns

Throughout this thesis, we shall evaluate routing algorithms on a suite of benign and adversarial traffic patterns. The patterns for k-ary 2-cubes are shown in Table 2.2. The first two patterns are *benign* in the sense that they naturally balance load and hence give good throughput with simple routing algorithms. The next three patterns are *adversarial* patterns that cause load imbalance. These patterns have been used in the past [49, 25, 12, 16, 31] to stress and evaluate routing algorithms. Finally, the worst-case pattern is the traffic permutation (selected over all possible permutations) that gives the lowest throughput. In general, the worst-case pattern may be different for different routing algorithms.

Table 2.2: Traffic patterns for evaluation of routing algorithms on k-ary 2-cubes

Name	Description
NN	Nearest Neighbor: each node sends to one of its four neighbors with probability 0.25 each.
UR	Uniform Random: each node sends to a randomly selected node.
BC	Bit Complement: (x, y) sends to $(k-x-1, k-y-1)$.
TP	Transpose: (x, y) sends to (y, x).
TOR	Tornado: (x, y) sends to $(x + \frac{k}{2} - 1, y)$
WC	Worst-case: the permutation that gives the lowest throughput by achieving the maximum load on a single link (See Appendix A.1.3).

In the following three chapters, we propose oblivious and adaptive load-balanced routing algorithms for torus networks. These algorithms load balance traffic across the channels of the topology to yield high throughput on adversarial traffic patterns while simultaneously exploiting the locality of benign traffic patterns. They also gracefully handle momentary overloads to provide stability post saturation.

Chapter 3

Oblivious Load Balancing

In this chapter, we focus on oblivious routing algorithms. As mentioned before, oblivious algorithms select a path from a source to a destination using only the identity of the source and destination nodes. In other words, the routing decision is made "oblivious" of the state of the network. Oblivious algorithms may use randomization to choose between possible paths. They may also be categorized as minimal or non-minimal, depending on the length of the routes. In the following discussion, we propose randomized, non-minimal, oblivious routing algorithms — RLB and RLBth — for torus networks, and evaluate their performance compared to other oblivious routing algorithms. Before we do that, we present some motivation for our choice of non-minimal algorithms.

3.1 Balancing a 1-Dimensional ring: RLB and RLBth

In Chapter 1, we saw how a minimal routing algorithm (MIN) degrades the throughput of an 8 node ring to $b/3$ bits/s. The capacity of any k-ary n-cube is $2B/N = 8b/k$. Hence, for an 8 node ring ($k = 8$), $\Theta(R_{min}, \Pi_{tor}) = 0.33$ of capacity. In general, the performance of MIN degrades asymptotically to 25% of capacity for any dimensional torus network, implying a very poor worst-case performance:

Theorem 3. *The worst-case throughput for MIN is at most 25% of capacity, for tori with large radix k.*

Proof. We consider the tornado pattern, Π_{tor}, on a k-ary n-cube. In this traffic, every source sends a packet to a destination that is one less than half-way across the ring in the first dimension. For instance, in a k-ary 2-cube, every source (i, j) sends to $(i + k/2 - 1, j)$[1]. Then, with an injection load of α bits/s from each node, the load on the clockwise links is $\alpha(k/2 - 1)$ (the load on the 8 node ring was 3α in Figure 1.5). Hence, the throughput is $\Theta(R_{min}, \Pi_{tor}) = b/(k/2 - 1)$ (bits/s). Expressed as a fraction of capacity, the throughput is $\frac{k/8}{k/2-1} \approx 0.25$ for large values of k. It follows that $\Theta_{wc}(R_{min}) \leq 0.25$. □

The reason MIN gives such poor performance in the worst case is that a hard traffic pattern can cause severe load imbalance in the links of the network. To balance load over all links, a routing algorithm must send some traffic along non-minimal paths for adversarial traffic. Consider the tornado traffic pattern on the 8 node ring, but with a non-minimal routing algorithm that sends $5/8$ of all messages in the short direction around the ring — three hops clockwise — and the remaining $3/8$ of all messages in the long, counterclockwise direction (see Figure 3.1). Each link in the clockwise direction carries $5\alpha/8$ load from 3 nodes for a total load of $15\alpha/8$. Similarly, each link in the counterclockwise direction carries $3\alpha/8$ from 5 nodes and hence, also carries a total load of $15\alpha/8$. Thus, the traffic is perfectly balanced — each link has identical load. As a result of this load balance, the per-node throughput is increased by 60% to $\theta = 8b/15 = 0.53$ of capacity compared to that of a minimal scheme. Since this scheme routes non-minimally based on the *locality* of the destination, we call it Randomized Local Balanced routing (RLB) [43].

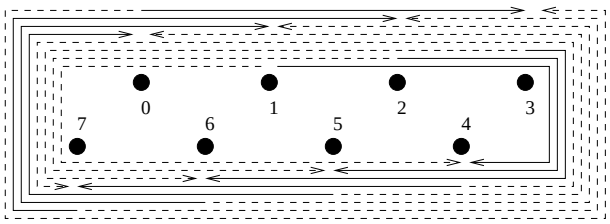


Figure 3.1: Non-minimally routing tornado traffic based on locality. The dashed lines contribute a link load of $\frac{3}{8}$ while the solid lines contribute a link load of $\frac{5}{8}$. All links equally loaded with load = $\frac{15}{8}$.

[1] All operations on node ids are done mod k.

With RLB routing, if a source node, s, sends traffic to destination node, d, then the distance in the short direction around the loop is $\Delta = \min(|s-d|, k-|s-d|)$, and the direction of the short path is $r = +1$ if the short path is clockwise, and $r = -1$ if the short path is counterclockwise. To exactly balance the load due to symmetric traffic, we send each packet in the short direction, r, with probability $P_r = \frac{k-\Delta}{k}$ and in the long direction, $-r$, with probability $P_{-r} = \frac{\Delta}{k}$. This loads $(k-\Delta)$ channels in the long direction with load P_{-r} and Δ channels in the short direction with load P_r for a total load of $\frac{\Delta(k-\Delta)}{k}$ in each direction.

With nearest-neighbor traffic (each node i sends half of its traffic to $i+1$ and half to $i-1$), for example, $\Delta = 1$, giving $P_r = \frac{k-1}{k}$. Thus, for $k = 8$, $7/8$ of the traffic traverses a single link and $1/8$ traverses seven links. On average, each packet traverses $14/8 = 1.75$ channels — evenly distributed in the two directions — and hence, throughput is $\theta = 2b/1.75 = 1.14b = 1.14$ times capacity.

This simple comparison in one dimension shows the capability of RLB to give good performance on an adversarial traffic pattern without sacrificing all the locality in benign traffic. It achieves a throughput of 0.53 on tornado traffic, a vast improvement over the 0.33 of a minimal algorithm, and it achieves 1.14 on nearest neighbor traffic, not as good as the 2 of a minimal algorithm, but much better than the 0.5 of fully random routing (VAL). In fact, we can prove that for a 1-dimensional torus, RLB, like VAL, gives optimal worst-case performance while performing much more efficiently on benign traffic:

Theorem 4. *RLB gives optimal worst-case throughput on a ring.*

Proof. Since the proof requires some background on how to construct the worst-case traffic for an oblivious routing algorithm, we present it in Appendix A.1.2. □

In order to improve RLB's performance on local traffic like nearest neighbor, we can modify the probability function of picking the short or long paths so that for very local traffic, RLB always routes minimally. Specifically, if $\Delta < \frac{k}{4}$ (the average hops in a k node ring), then the message must be routed minimally. Hence, $P_r = 1$ and $\mathrm{P}_{-r} = 0$ if $\Delta < \frac{k}{4}$, else P_r is the same as that in RLB. We call this modified version RLB threshold or RLBth. With this modification, RLBth achieves a throughput of 2 on nearest neighbor traffic while retaining a throughput of 0.53 on tornado traffic pattern.

3.2 RLB routing on higher dimensional tori

In multiple dimensions RLB works, as in the one dimensional case, by balancing load across multiple paths while favoring shorter paths. Unlike the one dimensional case, however, where there are just two possible paths for each packet — one short and one long, there are many possible paths for a packet in a multi-dimensional network. RLB exploits this path diversity to balance load.

To extend RLB to multiple dimensions, we start by independently choosing a direction for each dimension just as we did for the one-dimensional case above. Choosing the directions selects the *quadrant* in which a packet will be routed in a manner that balances load among the quadrants. To distribute traffic over a large number of paths within each quadrant, we route first from the source node, s, to a randomly selected intermediate node, q, within the selected quadrant, and then from q to the destination, d. For each of these two phases we route in *dimension order*, traversing along one dimension before starting on the next dimension, but randomly selecting the order in which the dimensions are traversed.

We select the quadrant to route in by choosing a direction for each of the n dimensions in a k-ary n-cube. Suppose the source node is $s = \{s_1, s_2, \ldots, s_n\}$ and the destination node is $d = \{d_1, d_2, \ldots, d_n\}$, where x_i is the coordinate of node x in dimension i. We compute a distance vector $\Delta = \{\Delta_1, \Delta_2, \ldots, \Delta_n\}$ where $\Delta_i = \min(|s_i - d_i|, k - |s_i - d_i|)$. From the distance vector, we compute a minimal direction vector $r = \{r_1, r_2, \ldots, r_n\}$, where for each dimension i, we choose r_i to be $+1$ if the short direction is clockwise (increasing node index) and -1 if the short direction is counterclockwise (decreasing node index). Finally we compute an RLB direction vector r' where for each dimension, i, we choose $r'_i = r_i$ with probability $P_{ri} = \frac{k-\Delta_i}{k}$ and $r'_i = -r_i$ with probability $1 - P_{ri} = \frac{\Delta_i}{k}$.

For example, suppose we are routing from $s = (0, 0)$ to $d = (2, 3)$ in an 8-ary 2-cube network (8×8 2-D torus). The distance vector is $\Delta = (2, 3)$, the minimal direction vector is $r = (+1, +1)$, and the probability vector is $P = (0.75, 0.625)$. We have four choices for r', $(+1, +1)$, $(+1, -1)$, $(-1, +1)$, and $(-1, -1)$ which we choose with probabilities 0.469, 0.281, 0.156, and 0.094 respectively. Each of these four directions describes a *quadrant* of the 2-D torus as shown in Figure 3.2. The weighting of directions routes more traffic in the minimal quadrant 1, $r' = (+1, +1)$ and less in quadrant 4 which takes the long path in both

dimensions $r' = (-1, -1)$. Moreover, this weighting of directions will exactly balance the load for any traffic pattern in which node $s = (x, y)$ sends to node $d = (x + \Delta_x, y + \Delta_y)$ — a 2-D generalization of tornado traffic.

Once we have selected the quadrant we need to select a path within the quadrant in a manner that balances the load across the quadrant's channels. There are a large number of unique paths across a quadrant which is given by:

$$N_p = \prod_{i=0}^{n-2} \begin{pmatrix} \sum_{j=i}^{n-1} \Delta_j \\ \Delta_i \end{pmatrix}$$

However, we do not need to randomly select among all of these paths. To balance the load across the channels, it suffices to randomly select an intermediate node, q, within the quadrant and then to route first from s to q and then from q to d. We then pick a random order of dimensions, o, for our route where o_i is the step during which the i^{th} dimension is traversed. We select this random ordering independently for both phases of routing. This is similar to the two-phase approach taken by a completely randomized algorithm. However, in this case the randomization is restricted to the selected quadrant.

It is important that the packet not backtrack during the second phase of the route, during which it is sent from q to d. If minimal routing were employed for the second phase, this could happen since the short path from q to d in one or more dimensions may not be in the direction specified by r'. To avoid backtracking, which unevenly distributes the load, we restrict the routing to travel in the directions specified by r' during both routing phases: from s to q and from q to d. These directions are fixed based on the quadrant the intermediate node q lies in as shown in Figure 3.2.

We need to randomly order the traversal of the dimensions to avoid load imbalance between quadrant links, in particular the links out of the source node and into the destination. Figure 3.3 shows how traversing dimensions in a fixed order leads to a large imbalance between certain links. If one dimension (say x) is always traversed before the other (say y), all the links are not evenly balanced. In the figure, if q is in local quadrant 1, then the downward link $(0, 0)$ - $(0, 1)$ will only be used if q is one of nodes $(0, 1)$, $(0, 2)$, or $(0, 3)$ while the right-going link $(0, 0)$ - $(1, 0)$ is used if q is any of the other nodes in the local quadrant. This, increases the likelihood of using the x dimension link over the y dimension

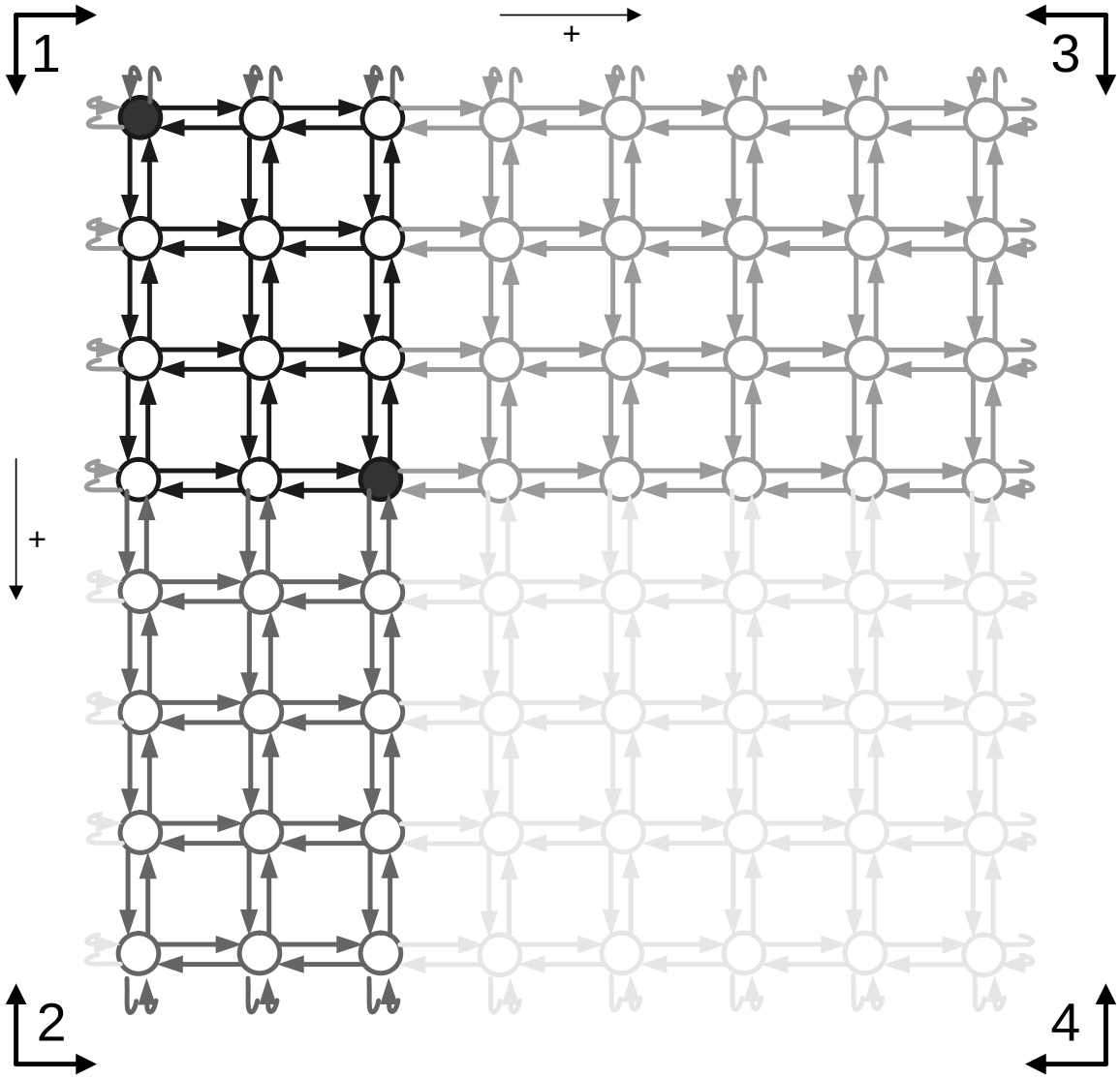


Figure 3.2: Probability distribution of the location of the intermediate node in RLB. All nodes in a similarly shaded region (quadrant) have equal probability of being picked.

link thereby unnecessarily overloading it.

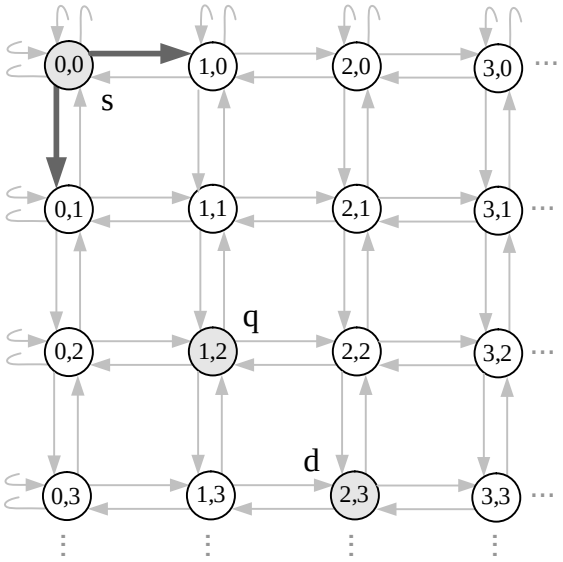


Figure 3.3: Traversing dimensions in fixed order causes load imbalance

Suppose in our example above, routing from (0,0) to (2,3) in an 8-ary 2-cube, we select the quadrant $r' = (-1, +1)$. Thus, we are going in the negative direction in x and the positive direction in y. We then randomly select q_x from $[3, 4, 5, 6, 7, 0]$ and q_y from $[0, 1, 2]$. Suppose this selection yields intermediate point $q = (7, 1)$. Finally, we randomly select an order $o = (1, 2)$ for the 1^{st} phase and also $o = (1, 2)$ for the 2^{nd} phase (note that the two orderings could have been different) implying that we will route in x first and then in y in both phases. Putting our choice of direction, intermediate node, and dimension order together gives the final route as shown in Figure 3.4. Note that if backtracking were permitted, a minimal router would choose the $+x$ direction after the first step since its only three hops in the $+x$ direction from q to d and five hops in the $-x$ direction.

Figure 3.5 shows how backtracking is avoided if directions are fixed for both the phases. The dotted path shows the path taken if Dimension Order Routing (traverse x dimension greedily, i.e., choosing the shortest path in that dimension and then traverse y dimension greedily) is followed in each phase when going from *s* to *q* to *d*. Fixing the direction sets based on the quadrant *q* is in, avoids the undesirable backtracking, as shown by the bold path.

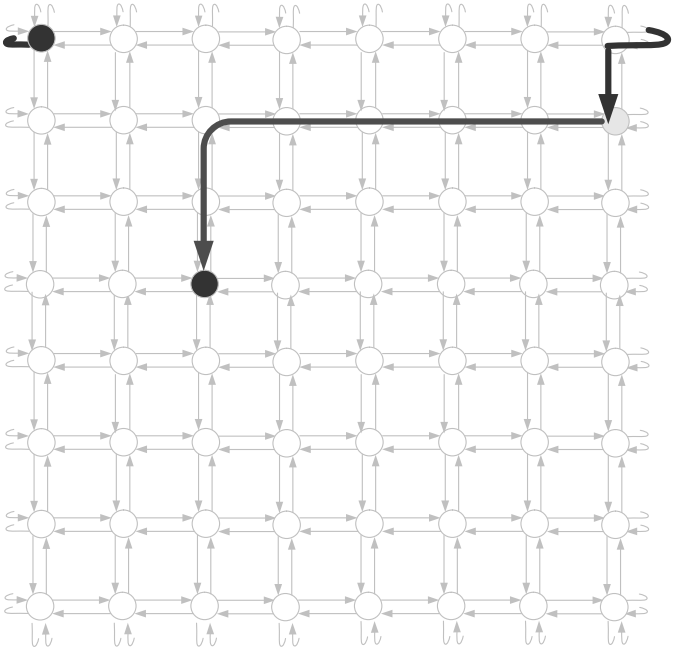


Figure 3.4: An example of routing using RLB

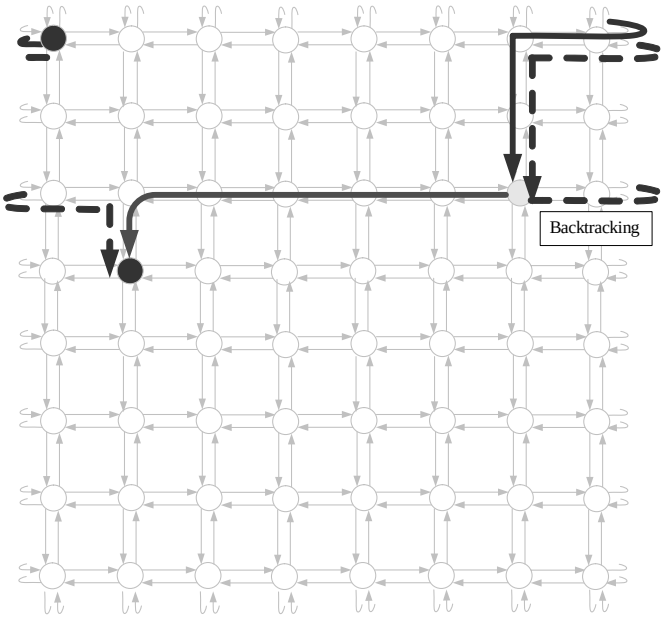


Figure 3.5: Avoiding backtracking in the RLB scheme. When the directions are fixed for both phases, routing is done along the bold path instead of the dotted path.

3.2.1 RLB threshold (RLBth) on higher dimensional tori

As in the one dimension case, RLBth works the same as RLB even for higher dimensions with a modification in the probability function for choosing the quadrants. Specifically, if $\Delta_i < \frac{k}{4}$, then $P_{r_i} = 1$ and $P_{-r_i} = 0$, else $P_{r_i} = \frac{k-\Delta_i}{k}$ and $P_{-r_i} = \frac{\Delta_i}{k}$. The threshold value of $\frac{k}{4}$ comes from the fact that it is the average hop distance for a k node ring in each dimension.

3.3 Performance Evaluation

3.3.1 Experimental setup

Measurements in this section have been made on a cycle-accurate network simulator for a k-ary 2-cube network that models the pipeline of each router as described in [32]. Any contention is resolved using age-based arbitration, always giving priority to a packet with an older time-stamp since injection. All latency numbers presented are measured since the time of birth of the packets and include the time spent by the packets in the source queues. We further assume that the network uses ideal flow control with each node having buffers of infinite length. Using this idealized model of flow control allows us to isolate the effect of the routing algorithm from flow control issues. The RLB algorithms can be applied to other flow control methods such as virtual channel flow control. The saturation throughput, Θ, is always normalized to the capacity of the network.

All simulations were instrumented to measure steady-state performance with a high degree of confidence. The simulator was warmed up under load without taking measurements until none of the queue sizes changed by more than 1% over a period of 100 cycles. Once the simulator was warmed up, a sample of injected packets was labeled for measurement. This sample size was chosen to ensure that measurements are accurate to within 3% with 99% confidence. The simulation was run until all labeled packets reached their destinations. We have simulated two topologies, an 8-ary 2-cube and a 16-ary 2-cube, but present only the results for the 8-ary 2-cube topology. The results obtained for the 16-ary 2-cube topology follow the same trends and are presented in Appendix C.

3.3.2 Latency-load curves for RLB

The latency-load curve for each traffic pattern of Table 2.2 (except NN) applied to an 8-ary 2-cube network using RLB is shown in Figure 3.6[2]. Each curve starts at the y-axis at the zero load latency for that traffic pattern which is determined entirely by the number of hops required for the average packet and the packet length. As offered traffic is increased, latency increases because of queueing due to contention for channels. Ultimately, a point is reached where the latency increases without bound. The point where this occurs is the *saturation throughput* for the traffic pattern, the maximum bandwidth that can be input to each node of the network in steady state. The numerical values of this saturation throughput for each traffic pattern are given in Table 3.1.

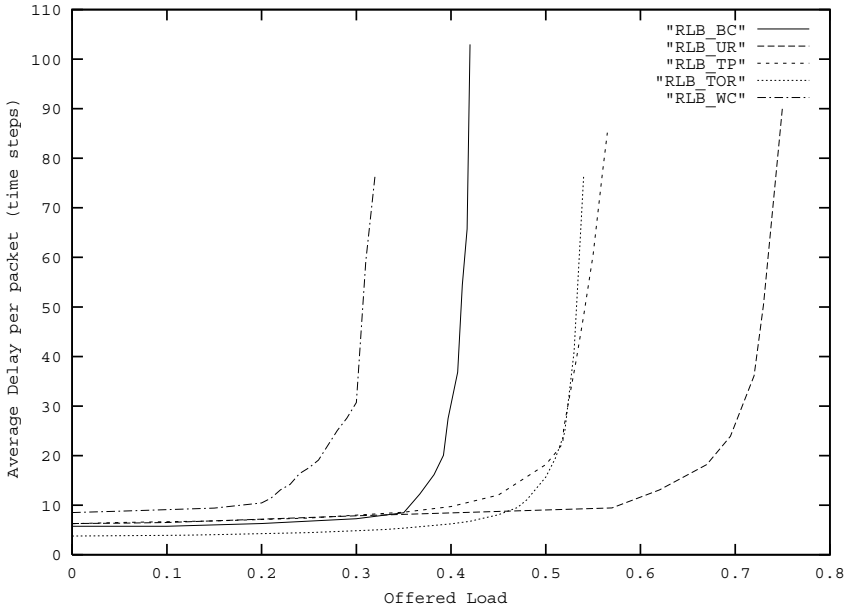


Figure 3.6: RLB delay-load curves for various traffic patterns

[2]The NN curve is omitted to allow the throughput scale to be compressed improving clarity.

Table 3.1: Saturation throughput of RLB and its backtracking variation

Traffic	RLB	Backtrack
NN	2.33	2.9
UR	0.76	0.846
BC	0.421	0.421
TP	0.565	0.50
TOR	0.533	0.4
WC	0.313	0.27

3.3.3 Effect of backtracking

In describing RLB in Section 3.2, we qualitatively discussed the importance of avoiding backtracking during the second phase of routing. Table 3.1 shows quantitatively how backtracking affects performance. The first column shows the saturation throughput of RLB on each of the six traffic patterns — the asymptotes of the curves in Figure 3.6. The second column shows throughput on each traffic pattern using a variation of RLB in which backtracking is permitted. With this algorithm, after routing to intermediate node q, the packet is routed over the shortest path to the destination, not necessarily going in the same direction, as indicated by the dashed path in Figure 3.5.

The table shows that backtracking improves performance for the two benign cases but gives lower performance on tornado and worst-case traffic. The improvement on benign traffic occurs because RLB with backtracking is closer to minimal routing — it traverses fewer hops than RLB without backtracking. The penalty paid for this is poorer performance on traffic patterns like TOR that require non-minimal routing to balance load. We discuss some other variations on RLB in Section 3.4.

3.3.4 Throughput on specific traffic patterns

We now compare the performance of RLB with that of the three oblivious routing algorithms of Table 2.1. Table 3.2 shows the saturation throughput of each oblivious algorithm

on each traffic pattern[3]. The minimal algorithms, DOR and ROMM, offer the best performance on benign traffic patterns but have very poor worst-case performance. VAL gives the best worst-case performance but converts every traffic pattern to this worst case, giving very poor performance on the benign patterns. RLB strikes a balance between these two extremes, achieving a throughput of 0.313 on worst-case traffic (50% better than ROMM and 25% better than DOR) while maintaining a throughput of 2.33 on NN (366% better than VAL) and 0.76 on UR (52% better than VAL). RLBth improves the locality of RLB — matching the throughput of minimal algorithms in the best case and improving the UR throughput of RLB (64% better than VAL). In doing so, however, it marginally deteriorates RLB's worst case performance by 4%.

Table 3.2: Comparison of saturation throughput of RLB, RLBth and three other routing algorithms on an 8-ary 2-cube for six traffic patterns

Traf	DOR	VAL	ROMM	RLB	RLBth
NN	4	0.5	4	2.33	4
UR	1	0.5	1	0.76	0.82
BC	0.50	0.5	0.4	0.421	0.41
TP	0.25	0.5	0.54	0.565	0.56
TOR	0.33	0.5	0.33	0.533	0.533
WC	0.25	0.5	0.208	0.313	0.30

Figure 3.7 shows the latency-throughput curve for each of the five algorithms on nearest-neighbor (NN) traffic. RLBth, ROMM, and DOR share the same curve on this plot since they all choose a minimal route on this traffic pattern. The VAL curve starts at a much higher zero load latency because it destroys the locality in the pattern.

The latency-throughput curves for each algorithm on bit complement (BC) traffic are shown in Figure 3.8. At almost all values of offered load, VAL has significantly higher latency. However, VAL has a higher saturation throughput than RLB.

The worst case row of Table 3.2 reflects the lowest throughput for each algorithm over all possible traffic patterns. The worst case throughput and traffic pattern (permutation) for each routing algorithm is computed using the method described in Appendix A. Using

[3]The worst-case pattern is different for each algorithm (See Appendix A).

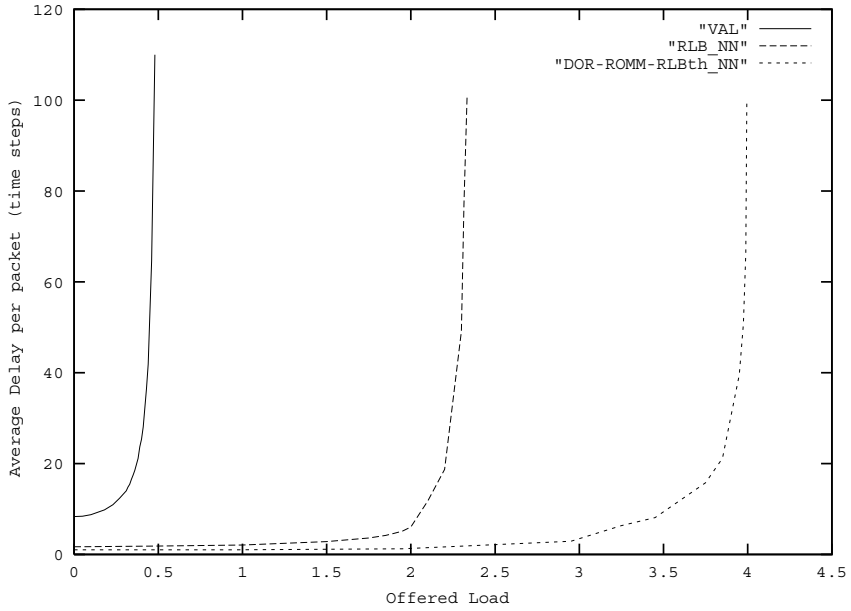


Figure 3.7: Performance of different algorithms on NN (Nearest neighbor) traffic

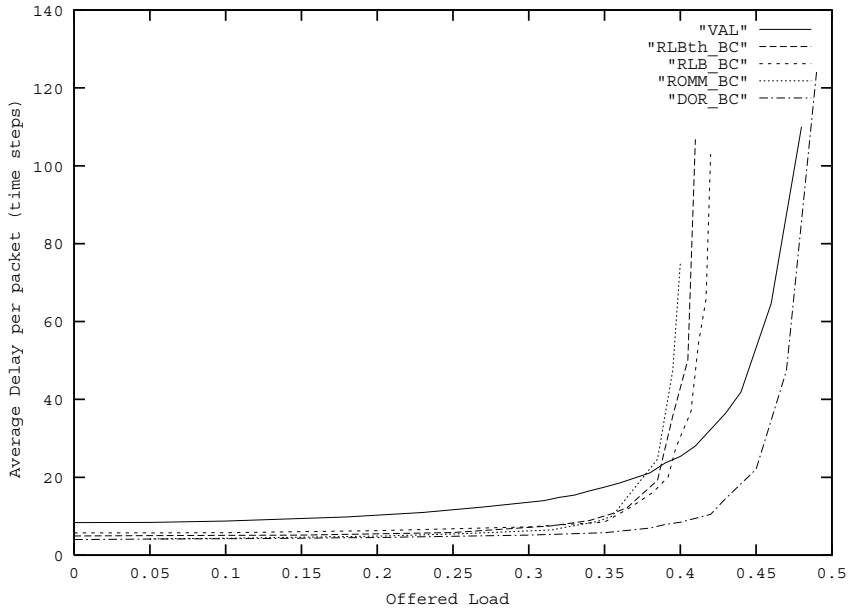


Figure 3.8: Performance of different algorithms on BC (Bit Complement) traffic

worst-case permutations for this evaluation is more accurate than picking some arbitrary adversarial traffic pattern (like BC, TP, or TOR) since the worst-case pattern for an algorithm is often quite subtle.

3.3.5 Throughput on random permutations

One might ask how often permutations as bad as the adversarial patterns in Table 3.2 occur in practice. To address this question, we compare the throughput of RLB and RLBth with VAL, ROMM, and DOR on 10^6 randomly selected permutations[4]. Since simulating 10^6 permutations on a cycle-accurate simulator is not feasible, we analytically compute the throughput for each permutation using the technique presented in Appendix A.1.1.

Histograms of the saturation throughput across the permutations are shown in Figure 3.9. RLB has a smooth bell-shaped histogram centered at 0.51 throughput. RLBth's histogram (not shown) is almost identical to that of RLB but centered at 0.512. VAL achieves the same throughput on all traffic permutations. Hence, its histogram is a delta function at 0.5. The histogram for ROMM is noisier and has an average saturation throughput of 0.453 — 12% lower than RLB's throughput. DOR's histogram has three spikes at 0.25, 0.33 and 0.5 corresponding to a worst case link load of 4, 3 and 2 in any permutation. DOR's average saturation throughput is 0.314, 39% lower compared to RLBth. The average saturation throughput values are summarized in Table 3.3. RLB algorithms have higher average throughput on random permutations than VAL, ROMM, or DOR.

Table 3.3: Average Saturation Throughput for 10^6 random traffic permutations

Algorithm	Average throughput
RLBth	0.512
RLB	0.510
VAL	0.500
ROMM	0.453
DOR	0.314

[4]These 10^6 permutations are selected from the $N! = 64!$ possible permutations on a 64-node 8-ary 2-cube.

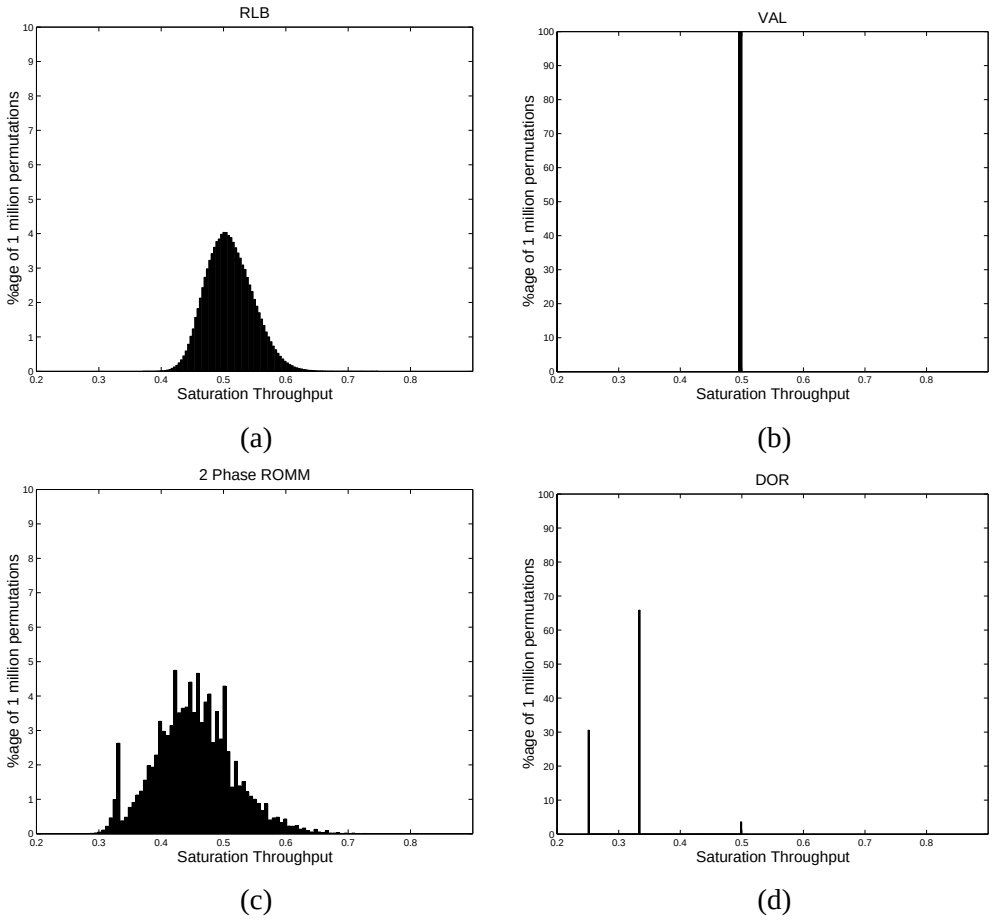


Figure 3.9: Histograms for the saturation throughput for 10^6 random permutations. (a) RLB, (b) VAL, (c) ROMM, (d) DOR.

3.3.6 Latency

RLB gives a lower packet latency than fully randomized routing (VAL). To quantify this latency reduction, we computed latency histograms between representative pairs of source and destination in a network loaded with uniform random traffic for RLB, RLBth, VAL, ROMM, and DOR.

The latency, T, incurred by a packet is the sum of two components, $T = H + Q$, where H is the hop count and Q is the queueing delay. The average value of H is constant with load while that of Q rises as the offered load is increased. For a minimal algorithm, H is equivalent to the *Manhattan* distance, D, from source to destination. For non-minimal algorithms, $H \geq D$.

In an 8-ary 2-cube, the Manhattan distance between a source and a destination node can range from 1 to 8. In our experiments, we chose to measure the latency incurred by packets from a source to 3 different destination nodes:

- A $(0, 0)$ to B $(1, 1)$: path length of 2 representing very local traffic.
- A $(0, 0)$ to C $(1, 3)$: path length of 4 representing semi-local traffic.
- A $(0, 0)$ to D $(4, 4)$: path length of 8 representing non-local traffic.

The histograms for semi-local paths (packets from A to C) are presented in Figure 3.10. The histograms are computed by measuring the latency of 10^4 packets for each of these three pairs. For all experiments, offered load was held constant at 0.2. The experiment was repeated for each of the five routing algorithms. The histogram for DOR is almost identical to that of ROMM and is not presented.

DOR and ROMM have a distribution that starts at 4 and drops off exponentially — reflecting the distribution of queueing wait times. This gives an average latency of 4.28 and 4.43, respectively. Since both these algorithms always route minimally, their H value is 4 and therefore, Q values are 0.28 and 0.43, respectively.

RLBth has a distribution that is the superposition of two exponentially decaying distributions: one with a H of 4 that corresponds to picking quadrant 1 of Figure 3.2 and a second distribution with lower magnitude starting at $H = 6$ that corresponds to picking

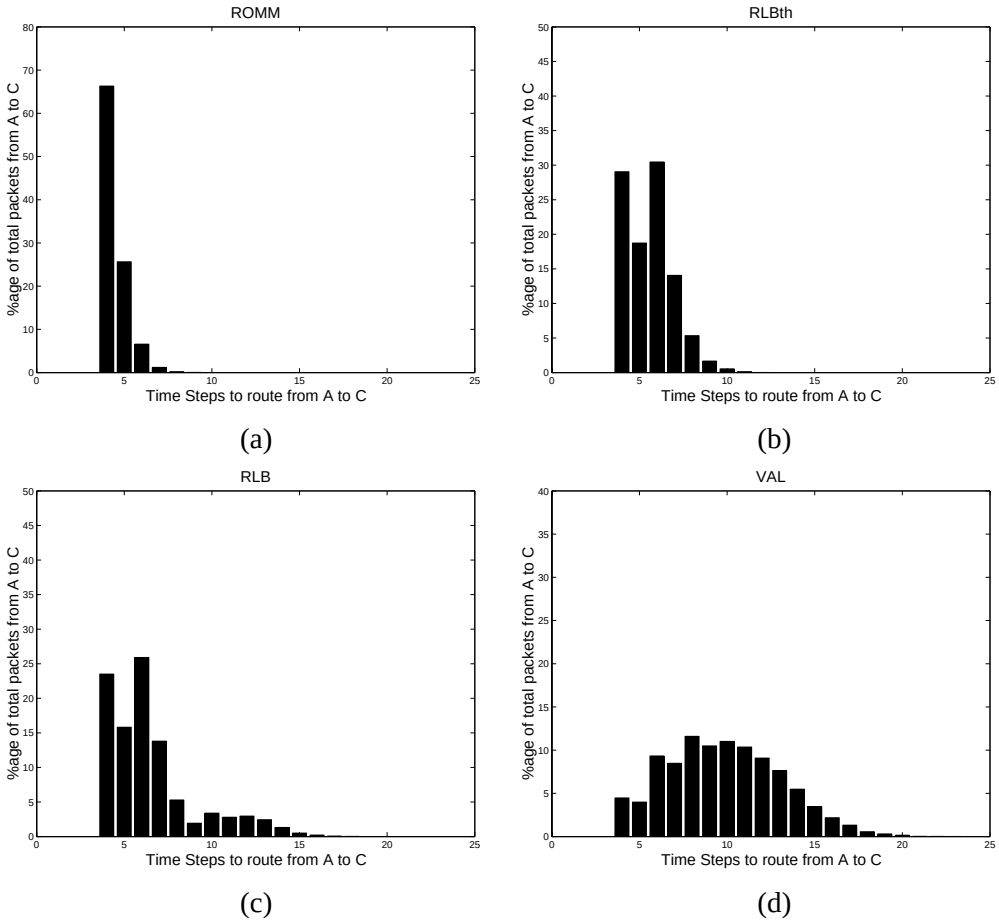


Figure 3.10: Histograms for 10^4 packets routed from node A(0,0) to node C(1,3). (a) ROMM, (b) RLBth, (c) RLB, (d) VAL. The network is subjected to UR pattern at 0.2 load.

quadrant 2. The bar at $T = 6$ appears higher than the bar at $T = 4$ because it includes both the packets with $H = 6$ and $Q = 0$ and packets with $H = 4$ and $Q = 2$. The average H for RLBth is 4.75, giving an average Q of 0.81.

The distribution for RLB includes the two exponentially decaying distributions of RLBth corresponding to $H = 4$ and $H = 6$ and adds to this two additional distributions corresponding to $H = 10$ and $H = 12$ corresponding to quadrants 3 and 4 of Figure 3.2. The probability of picking quadrants 3 and 4 is low, yielding low magnitudes for the distributions starting at 10 and 12. The average H for RLB is 5.5, giving an average Q of 0.98.

VAL has a very high latency with a broad distribution centered at $T = 9.78$. This broad peak is the superposition of exponentially decaying distributions starting at all even numbers from 4 to 12. The average H component of this delay is 8 since each of the two phases is a route involving a fixed node and a completely random node (4 steps away on average). The average Q is 1.78.

The results for all the three representative paths are summarized in Table 3.4. VAL performs the worst out of all the algorithms. It has the same high H and Q latency for all paths. DOR and ROMM being minimal algorithms, do the best at this low load of 0.2. They win because their H latency is minimal and at a low load their Q latency is not too high. RLB algorithms perform much better than VAL — in both H and Q values. RLB is on average 2.2 times, 1.5 times and 1.1 times faster than VAL on local, semi-local and non-local paths, respectively. RLBth does even better by quickly delivering the very local messages — being 3.65 times, 1.76 times and 1.11 times faster than VAL on the same three paths as above.

3.4 Taxonomy of locality-preserving algorithms

RLB performs three randomizations to achieve its high degree of load balance: (1) it randomly chooses a quadrant, and hence a direction vector for routing, (2) it randomly chooses an order in which to traverse the dimensions, and (3) it randomly chooses an intermediate way-point node in the selected quadrant. We can generate eight non-backtracking, locality-preserving randomized routing algorithms by disabling one or more of these randomizations.

Table 3.4: Average total, hop and queueing latency (in time steps) for 10^4 packets for 3 sets of representative traffic paths at 0.2 load. $A-B$, $A-C$ and $A-D$ represent local, semi-local and non-local paths, respectively. A is node (0,0), B is (1,1), C is (1,3), and D is (4,4). All other nodes send packets in a uniformly random manner at the same load.

Algo	T_{AB}	H_{AB}	Q_{AB}	T_{AC}	H_{AC}	Q_{AC}	T_{AD}	H_{AD}	Q_{AD}
DOR	2.3	2	0.3	4.28	4	0.28	8.24	8	0.24
ROMM	2.34	2	0.34	4.43	4	0.43	8.42	8	0.42
RLBth	2.68	2	0.68	5.56	4.75	0.81	8.81	8	0.42
RLB	4.31	3.5	0.81	6.48	5.5	0.98	8.92	8	0.92
VAL	9.78	8	1.78	9.78	8	1.78	9.78	8	1.78

In this taxonomy of routing algorithms, each algorithm is characterized by a 3-bit vector. If the first bit is set, the quadrant is chosen randomly (weighted to favor locality). Otherwise, the minimal quadrant is always used. If this bit is clear the routing algorithm will be minimal. The dimensions are traversed in a random order, if the second bit is set and in a fixed order (x first, then y, etc...) if this bit is clear. Finally, the third bit, if set, causes the packet to be routed first to a random way-point in the selected quadrant and then to proceed to the destination without reversing direction in any dimension. For example a vector of 111 corresponds to RLB — all randomizations enabled and a vector of 000 corresponds to DOR — no randomization. By examining the points between these two extremes we can quantify the contribution to load balance of each of the three randomizations.

Table 3.5 describes the eight algorithms and gives their performance on our six traffic patterns. All four minimal algorithms have the same high performance on the benign traffic patterns (NN and UR) since they never misroute. The first randomization we consider is the order of dimensions. Vector 010 gives us dimension order routing with random dimension order — e.g., in 2-D we go x-first half the time and y-first half the time. This randomization eases the bottleneck on TP traffic, doubling performance on this pattern, but does not affect worst-case performance. So we can see that randomizing dimension order alone does not improve worst-case performance.

Next, let us consider the effect of a random way-point in isolation. Vector 001 gives us ROMM, in which we route to a random way-point in the minimal quadrant and then on to

the destination. This randomization, while it improves performance on TP traffic, actually reduces worst-case throughput and throughput on BC traffic. This is because the choice of a random way-point concentrates traffic in the center of a region for these patterns. Combining random directions with a random way-point (vector 011) improves throughput for TP. However, it does not affect performance on the other patterns. Thus, routing to a random way-point alone actually makes things worse, not better.

Table 3.5: Taxonomy of locality preserving randomized algorithms. Saturation throughput values are presented for an 8-ary 2-cube topology.

Vector	Description	NN	UR	BC	TP	TOR	WC
000	DOR-F: dimension-order routing	4	1	0.5	0.25	0.33	0.25
010	DOR-R: with randomized dimension order	4	1	0.5	0.5	0.33	0.25
001	ROMM-F: fixed dimension order — route first to a random node q in the minimal quadrant and then to the destination	4	1	0.4	0.438	0.33	0.208
011	ROMM-R: random dimension order — like 001 but the order in which dimensions are traversed is randomly selected for both phases.	4	1	0.4	0.54	0.33	0.208
100	RDR-F: randomly select a quadrant (weighted for locality) and then route in this quadrant using a fixed dimension order	2.28	0.762	0.5	0.286	0.533	0.286
110	RDR-R: with random dimension order	2.286	0.762	0.5	0.571	0.533	0.286
101	RLB-F: with fixed dimension order	2.286	0.762	0.421	0.49	0.533	0.310
111	RLB-R	2.33	0.76	0.421	0.565	0.533	0.313

Finally, we will consider the non-minimal algorithms. Vector 100 corresponds to random direction routing (RDR) in which we randomly select directions in each dimension, in effect selecting a quadrant, and then use dimension-order routing within that quadrant. As described in Section 3.2, this selection is weighted to favor locality. Randomly selecting the quadrant by itself gives us most of the benefits (and penalties) of RLB. We improve worst-case performance by 14% compared to the best minimal scheme, and we get the best performance of any non-minimal algorithm on BC. However, performance on TP suffers, it is equal to worst-case, due to the fixed dimension order. Randomizing the dimension order, vector 110, fixes the TP problem but does not affect the other numbers.

Routing first to a random way-point within a randomly-selected quadrant, vector 101, slightly improves worst-case performance (24% better than minimal and 8% better than RDR). However using a random way-point makes performance on TP and BC worse. Putting all three randomizations together, which yields RLB as described in Section 3.2, gives slightly better worst-case, TP, and NN performance.

Overall, the results show that randomization of quadrant selection has the greatest impact on worst-case performance. Non-minimal routing is essential to balance the load on adversarial traffic patterns. Once quadrant selection is randomized, the next most important randomization is selection of a random way-point. This exploits the considerable path diversity within the quadrant to further balance load. However, applying this randomization by itself actually reduces worst-case throughput. The randomization of dimension order is the least important of the three having little impact on worst-case throughput. However, if a random way-point is not used, randomizing dimension order doubles throughput on traffic patterns such as TP.

3.5 Discussion

3.5.1 The drawbacks of RLB

Having evaluated the performance of RLB on an 8-ary 2-cube, we realize that RLB suffers from suboptimal performance in two aspects: (1) Unlike VAL, it does not perform optimally in the worst-case and (2) Unlike MIN, it does not perform optimally on benign

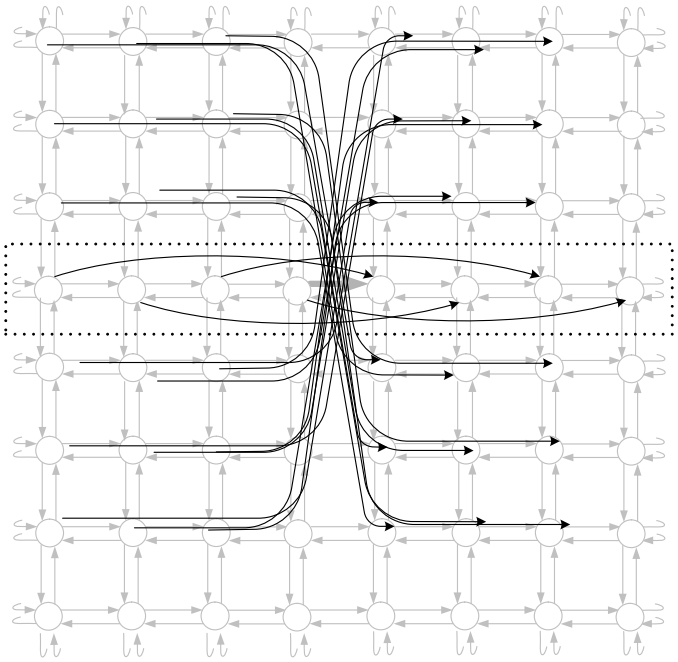


Figure 3.11: Adversarial traffic for RLB

traffic.

The reason for RLB's lower worst-case throughput is illustrated by constructing an adversarial traffic permutation as shown in Figure 3.11. This traffic is very similar to the worst-case traffic permutation derived in Appendix A. We use this traffic as it succinctly demonstrates how RLB can yield a throughput under 0.5. The pattern first focuses on one row of the network (outlined with a dashed box in the figure). In this row, each node sends traffic exactly half-way across the ring in the row. Since the y coordinates are matched, RLB routes all traffic along the same row, never routing outside the dashed box. As a result the load on the bold grey link in that row is 2α. This channel load can be increased further by selecting source-destination pairs outside the boxed row, which further load the channel in bold. By setting up a large number of these crossing patterns, as shown in Figure 3.11, the channel load can be increased to significantly higher than 2α, thereby reducing the throughput much below 0.5.

Adaptively deciding the next dimension to traverse is an effective way of reducing the congestion on the worst-case link described above. In the next two chapters, we shall

incorporate adaptivity in our load-balancing algorithms to alleviate both the sub-optimal worst-case performance as well as the benign performance of RLB.

3.5.2 Deadlock and livelock

Deadlock is a situation that occurs when a cycle of packets are waiting for one another to release resources, and hence are blocked indefinitely. *Livelock* is a condition whereby a packet keeps circulating within the network without ever reaching its destination. Freedom from such critical conditions must be guaranteed. All minimal algorithms such as DOR and ROMM guarantee livelock freedom with fair arbitration since each channel traversed by a packet reduces the distance to the destination. VAL is also deterministically livelock free since it is minimal in each of its phases. RLB algorithms, while non-minimal, are also inherently livelock free. Once a route has been selected for a packet, the packet monotonically makes progress along the route, reducing the number of hops to the destination at each step. Since there is no incremental misrouting, all packets reach their destinations after a predetermined, bounded number of hops.

As stated in Section 3.3.1, we assume ideal flow control with unbounded buffers for the results presented in this paper so deadlock due to channel or buffer dependency is not an issue. The results here can be extended to virtual channel flow control as we shall demonstrate in the next two chapters.

3.5.3 Packet reordering

The use of a randomized routing algorithm can and will cause out-of-order delivery of packets. While this may be acceptable for multiprocessor systems with a relaxed memory coherence model, memory systems with strict coherence and Internet routers require in-order delivery.

Several methods can be used to guarantee in order delivery of packets where needed. One approach is to ensure that packets that must remain ordered (e.g., memory requests to the same address or packets that belong to the same flow) follow the same route. This can be accomplished, for example, by using a packet group identifier (e.g., the memory address or the flow identifier) to select the quadrant, the intermediate node and the order of

traversing the dimensions for the route. Packet order can also be guaranteed by reordering packets at the destination node using the well known sliding window protocol [46].

3.6 Summary

Randomized Local Balance (RLB) is a non-minimal oblivious algorithm that balances load by randomizing three aspects of the route: the selection of the routing *quadrant*, the order of dimensions traversed, and the selection of an intermediate way-point node. RLB weights the selection of the routing quadrant to preserve locality. The probability of misrouting in a given dimension is proportional to the distance to be traversed in that dimension. This exactly balances traffic for symmetric traffic patterns like tornado traffic. RLBth is identical to RLB except that it routes minimally in a dimension if the distance in that dimension is less than a threshold value ($\frac{k}{4}$).

RLB strikes a balance between randomizing routes to achieve high guaranteed performance on worst-case traffic and preserving locality to maintain good performance on average or neighbor traffic. On worst-case traffic, RLB outperforms all minimal algorithms, achieving 25% more throughput than dimension-order routing and 50% more throughput than ROMM, a minimal oblivious algorithm. Unlike the 1-D case, RLB does not give optimal worst-case guarantees in higher dimensional tori. In an 8-ary 2-cube, for instance, the worst-case throughput of RLB is 37% lower than the throughput of a fully randomized routing algorithm. This degradation in worst-case throughput is balanced by a substantial increase in throughput on local traffic. RLB (RLBth) outperforms VAL by 4.6 (8) on NN traffic and 1.52 (1.69) on UR traffic. RLBth improves the locality of RLB, matching the performance of minimal algorithms on NN traffic, at the expense of a 4% degradation in worst-case throughput. Both RLB and RLBth give higher saturation throughput on average for 10^6 random traffic permutations. Moreover, RLB and RLBth provide much lower latency, upto 3.65 times less, than VAL.

By selectively disabling the three sources of randomization in RLB, we are able to identify the relative importance of each source. Our results show that the advantages of RLB are primarily due to the weighted random selection of the routing quadrant. Routing a fraction of the traffic the long way around each dimension effectively balances load for

many worst-case patterns. By itself, randomly choosing dimension order has little effect on worst-case performance and by itself, picking a random intermediate node actually reduces worst-case throughput.

In the next chapter, we incorporate adaptive decisions in RLB routing to alleviate its sub-optimal worst-case performance.

Chapter 4

GOAL Load Balancing

In the previous chapter, we showed how an adversary can load a link for an oblivious load-balancing algorithm resulting in sub-optimal worst-case performance. In order to improve the worst-case performance, we incorporate adaptivity into the RLB algorithm, i.e., we permit routing decisions to be made based on the congestion in the network. However, such decisions need to be made based solely on local congestion information. In order for such local decisions to be effective, we incorporate a realistic flow control in our routing scheme in which buffers are no longer infinite. This means that when buffers are full, they *backpressure* the preceding buffers thus, propagating congestion information. Combining the *global* load balance of RLB and the *local* balance of adaptive methods, we get *Globally Oblivious Adaptive Locally* (GOAL) [41].

GOAL, like RLB, obliviously chooses the direction of travel in each dimension weighting the *short* direction more heavily than the *long* direction in a manner that globally balances channel load while preserving some locality. Once the directions are selected, the packet is adaptively routed to the destination in the resulting quadrant. This adaptive routing avoids the local imbalance created by adversarial patterns. For instance, in the adversarial pattern shown in Figure 3.11, if routing were done adaptively within the quadrants, packets would be routed away from the congested link shown in bold grey, thereby increasing the throughput to 0.5.

We next formally define GOAL and evaluate its performance on various metrics.

4.1 GOAL

GOAL is equivalent to RLB in 1-dimension. In higher dimensional tori, GOAL routes a packet from a source node, $s = \{s_1, \ldots, s_n\}$, to the destination node, $d = \{d_1, \ldots, d_n\}$, by obliviously choosing the direction to travel in each of the n dimensions to exactly balance channel load (as is done by the RLB algorithm). Once the quadrant is selected, the packet is routed adaptively within the quadrant from s to d. A dimension i is termed *productive* if the coordinate of the current node x_i differs from d_i. Hence, it is productive to move in that dimension since the packet is not already at the destination coordinate. At each hop, the router advances the packet in the productive dimension that has the shortest output queue.

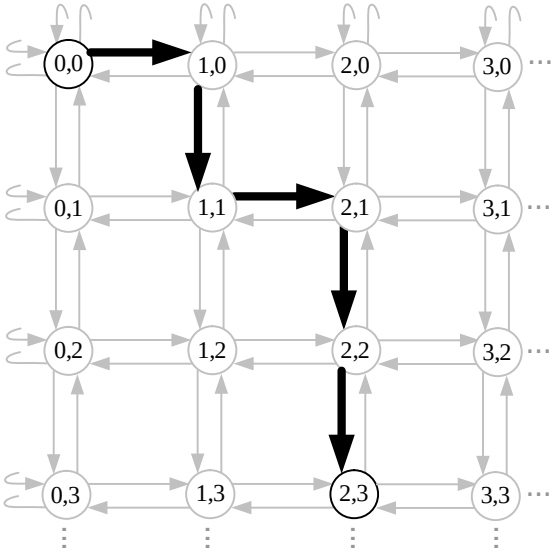


Figure 4.1: Example route from S (0,0) to D (2,3) through the minimal quadrant (+1,+1)

Revisiting our example of routing from $s = (0, 0)$ to $d = (2, 3)$ in an 8-ary 2-cube network (Chapter 3.2). Suppose GOAL obliviously chooses the minimal quadrant 1, $(+1, +1)$. One possible route of the packet is shown in Figure 4.1. On the first hop, the productive dimension vector is $p = (1, 1)$, i.e., both the x and y dimensions are productive. Suppose the queue in the x dimension is shorter so the packet proceeds to node $(1, 0)$. At $(1, 0)$, p is still $(1, 1)$ so the packet can still be routed in either x or y. At this point, suppose the queue in the y dimension is shorter, so the packet advances to node $(1, 1)$. At $(1, 1)$, p is still $(1, 1)$ and this time the route is in x to $(2, 1)$. At this point, the packet has reached the destination

coordinate in x so $p = (0, 1)$. Since the only productive dimension is y, the remaining hops are made in the y dimension regardless of queue length.

4.1.1 Virtual channels and deadlock

Since we use finite buffers, we incorporate *virtual channel flow control* [9] to avoid deadlock in the network. Our implementation of GOAL employs 3 virtual channels (VCs) per unidirectional physical channel (PC) to achieve deadlock freedom in the network. This is an extension of the scheme proposed in the $*$-channels algorithm [16] applied to the non-minimal GOAL algorithm. There are two types of virtual channels per PC, $*$ and *non*-$*$. Packets move through the $*$-channels *only* when traversing the most significant *productive* dimension. The *non*-$*$ channels are fully adaptive and can be used at any time. In order to make the $*$-channel subnetwork free from deadlock, we have two $*$-channels per PC - $*_0$ and $*_1$. $*_1$ ($*_0$) is used if the packet has (has not) crossed a wrap-around edge in the current dimension. With these constraints, it can be proved that the channel dependency graph, CDG[1], for the $*$-channels associated with GOAL is acyclic. Moreover, no $*$-channel can ever participate in a deadlock cycle. Hence, every packet that has not reached its destination always has a $*$-channel in the set of virtual channels it can possibly use to make forward progress towards its destination. Therefore, if VCs are assigned fairly, deadlock can never arise. The formal proof is presented in Appendix B.

4.1.2 Livelock

Minimal algorithms, such as MIN AD, guarantee livelock freedom with fair arbitration since each channel traversed by a packet reduces the distance to the destination. The CHAOS scheme uses randomization to misroute from a shared queue of packets in each node during congestion. This randomization only ensures that the algorithm is *probabilistically* livelock free. GOAL, like RLB, while non-minimal, provides deterministic freedom from livelock. Once a route has been selected for a packet, the packet monotonically makes

[1]The CDG for a network, G, and a routing algorithm, R, is a directed graph, $D(C, E)$. The vertices of D are the channels of G. The edges of D are the pairs of channels (c_i, c_j) such that there is a direct dependency from c_i to c_j. For a detailed explanation of channel dependency graphs, see Chapter 3 of [13].

progress along the route, reducing the number of hops to the destination at each step. Since there is no incremental misrouting, all packets reach their destinations after a predetermined, bounded number of hops.

4.2 Performance evaluation

In this section, we compare the performance of GOAL with the algorithms described in Table 2.1. Since we have incorporated virtual channel flow control, we reevaluate the oblivious algorithms from the previous chapter with realistic flow control to present a fair picture. The deadlock avoidance mechanism for each algorithm is summarized in Table 4.1. Each packet is again assumed to be one flit long. The total buffer resources are held constant across all algorithms, i.e., the product of the number of VCs and the VC channel buffer depth is kept constant. For the CHAOS algorithm (which does not use VCs), we increase the number of buffers in the shared queue of each node. The rest of the experimental set up is similar to the one in Chapter 3.

Table 4.1: Deadlock avoidance schemes for the different routing algorithms

Name	Deadlock avoiding mechanism
DOR	Uses 2 VCs — VC_1 (VC_0) is used if the packet has (has not) crossed a wrap-around edge in the current dimension [45].
VAL	Uses 2 subnetworks (for each phase) of 2 VCs each [50].
ROMM	Uses the same scheme as VAL.
RLB	Uses the same scheme as GOAL.
CHAOS	Uses deflection routing [25].
MIN AD	Uses 3 VCs. A "non-star" VC is provided to ensure full adaptivity. Two "star" VCs ensure a deadlock free network just like in DOR [16].
GOAL	Uses 3 VCs.

4.2.1 Throughput on specific patterns

Figures 4.2 and 4.3 show the saturation throughput for each algorithm on each traffic pattern of Table 2.2. The two benign traffic patterns are shown in Figure 4.2 while the four adversarial patterns are shown in Figure 4.3 with an expanded vertical scale. The figures show that GOAL achieves performance at least as good as VAL on the adversarial patterns while offering significantly more performance on the benign patterns — 52% higher throughput on random traffic and $4.6\times$ the throughput on nearest-neighbor. However, the throughput of GOAL does not match the performance of minimal algorithms on the local patterns. This is the price of oblivious load balancing.

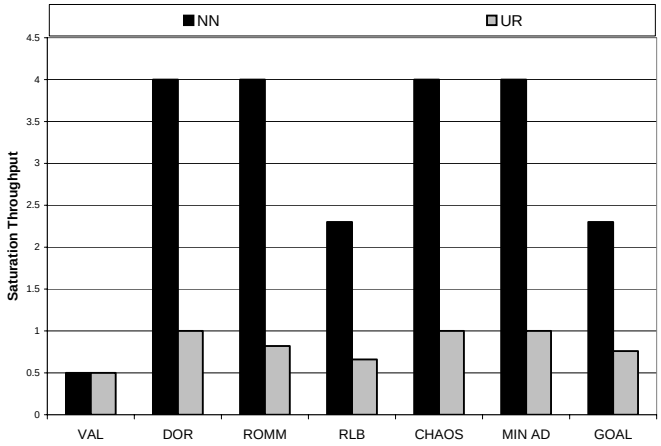


Figure 4.2: Comparison of saturation throughput of seven algorithms on an 8-ary 2-cube for two benign traffic patterns.

The figures also show that the minimal algorithms, DOR, ROMM and MIN AD, offer high performance on benign traffic patterns but have very poor worst-case performance. Because the misrouting in CHAOS is local in nature, its performance is comparable to that of MIN AD, a minimal algorithm.

The exact worst-case throughput for the oblivious algorithms — DOR, ROMM, VAL and RLB — is shown in Figure 4.3. Since there is no known method to evaluate the worst case pattern for adaptive algorithms, the worst case graphs shown for CHAOS, MIN AD and GOAL show the lowest throughput over all traffic patterns we have simulated. We know from Theorem 3 that MIN AD saturates at $\Theta = 0.33$ on TOR for $k = 8$. CHAOS

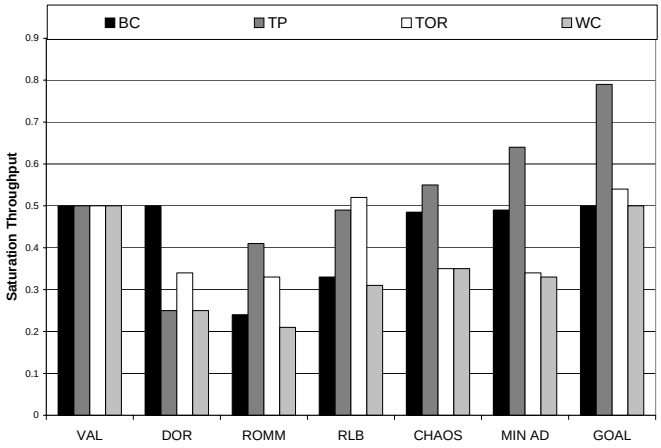


Figure 4.3: Comparison of saturation throughput of seven algorithms on an 8-ary 2-cube for four adversarial traffic patterns

does not perform appreciably better on TOR traffic saturating at $\Theta = 0.35$. The worst case pattern for GOAL that we know of is the DIA traffic pattern discussed in Theorem 1 on which it saturates at 0.5.

The latency-load curves for the benign UR pattern for all the algorithms are shown in Figure 4.4. On this benign pattern the minimal algorithms give the best performance, VAL gives the worst performance, and GOAL falls midway between the two. CHAOS, MIN AD, and DOR all offer minimal zero-load latency and unit throughput because they always take the shortest path. Because GOAL occasionally routes the long way around, its latency is increased and its throughput is reduced compared to the minimal algorithms. However, it offers substantially better performance than VAL, while yielding the same performance as VAL on worst-case traffic.

Figure 4.5 shows the latency-load curves for each algorithm on the adversarial TOR pattern for each of the algorithms. Here the balanced algorithms RLB and GOAL offer the best performance because they efficiently balance load across the two directions in the x dimension. VAL does nearly as well but has more than twice the zero load latency as the balanced algorithms because of its two-phase nature and because it takes gratuitous hops in the y dimension. The minimal algorithms — DOR, MIN AD, and ROMM — perform poorly (37% lower throughput than GOAL) on TOR because they route all of the traffic in

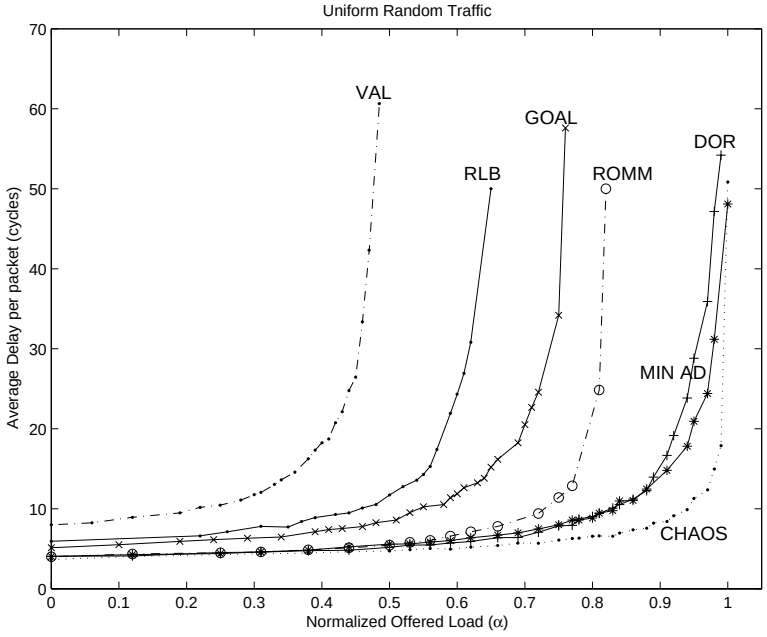


Figure 4.4: Performance of different algorithms on UR (Uniform Random) traffic

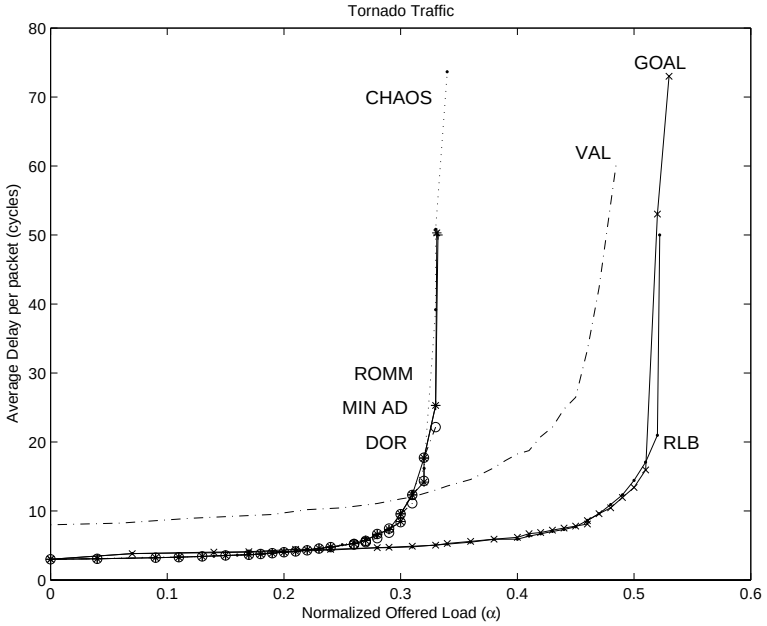


Figure 4.5: Performance of different algorithms on TOR (Tornado) traffic

the shorter direction, leaving the channels in the other direction idle. While CHAOS is not a minimal algorithm, its misrouting is local in nature and thus is not able to globally balance load across directions. Thus, the performance of CHAOS closely matches the performance of MIN AD on all adversarial patterns, including TOR.

4.2.2 Throughput on Random Permutations

As in Chapter 3, our next experiment is to measure the performance of each algorithm on random permutations. Since no analytical method is known to compute the throughput of adaptive algorithms, we simulate 1000 permutations[2] on our simulator for this experiment. Histograms of saturation throughput over these permutations for six of the algorithms are shown in Figure 4.6. No histogram is shown for VAL because its throughput is always 0.5 for all permutations. All the other routing algorithms have bell-shaped histograms. The highest, average and worst throughput in this experiment for each of the algorithms are presented in Figure 4.7.

The figures show that over the $1,000$ permutations, GOAL is the only algorithm with a worst-case throughput that matches or exceeds that of VAL. The minimal algorithms and CHAOS do substantially worse. GOAL outperforms the best of these algorithms, MIN AD, by 31%. The figures also show that despite the fact that it obliviously routes a fraction of traffic the long way around, GOAL has the highest average case throughput of all of the algorithms, outperforming MIN AD by 5%. This shows clearly that even for an *average* permutation, global load balance enhances performance and it is worth obliviously misrouting to achieve this balance.

The figure also shows the importance of using adaptive routing to achieve local balance[3]. GOAL has 49% higher average throughput and 75% higher worst-case throughput than RLB which also uses quadrant selection to achieve global balance but attempts to obliviously achieve local balance. For the same reason, the unbalanced adaptive algorithms MIN AD and CHAOS outperform the minimal oblivious algorithms, DOR and ROMM. MIN AD slightly outperforms CHAOS in terms of both average and worst-case throughput. This suggests that for most permutations, local misrouting is not advantageous.

[2] In Chapter 3, we had *analytically* computed the throughput for 10^6 permutations.

[3] This advantage of adaptive routing has also been noted in [3].

The reason is that, with local misrouting, a packet may be misrouted several times, alternating its path between both the $+$ and $-$ directions in a dimension. While techniques can be employed to limit this excessive misrouting, such as in the BLAM algorithm [47], the fundamental shortcoming of locally adaptive misrouting still exists — it incorporates no concept of global load balance into its routing decisions.

4.2.3 Latency

The latency experiment from Chapter 3 is repeated with the algorithms of Table 2.1. The results for all the three representative paths are presented in Figure 4.8. Under benign traffic at low load, the three minimal algorithms, DOR, ROMM, and MIN AD, and CHAOS (which behaves minimally at low load) give the lowest latency. All of these algorithms have a minimal hop count, $H = 4$, and the queueing delay Q is exponentially distributed with means ranging from 0.44 cycles for MIN AD to 0.76 cycles for ROMM. This difference in queueing delay further shows the advantage of adaptivity.

The balanced algorithms, RLB and GOAL, have higher latency (40% higher than minimal) than the minimal algorithms because they route a fraction of the traffic in the non-minimal quadrants.

4.2.4 Stability

In this subsection, we evaluate the stability (throughput with offered traffic in excess of the saturation throughput) of each routing algorithm. As described in Chapter 2, for a given destination matrix, Λ, and rate of offered traffic, α, we measure the accepted traffic, α', as the *minimum* accepted load over all source-destination pairs sending packets.

Figure 4.9 shows α^* (upper line), the *average* accepted throughput and α' (lower line), the *minimum* accepted throughput vs. α for the BC permutation on GOAL and CHAOS routing algorithms. The figure shows that CHAOS is unstable on this adversarial traffic pattern due to injection-queue starvation. Since CHAOS employs deflection routing to avoid deadlock, it accepts traffic from the node (source queue) only if resources are available after serving the input channels and the shared queue. Thus, at high, non-uniform loads, the source queues on nodes in high-traffic areas are starved indefinitely leading to a

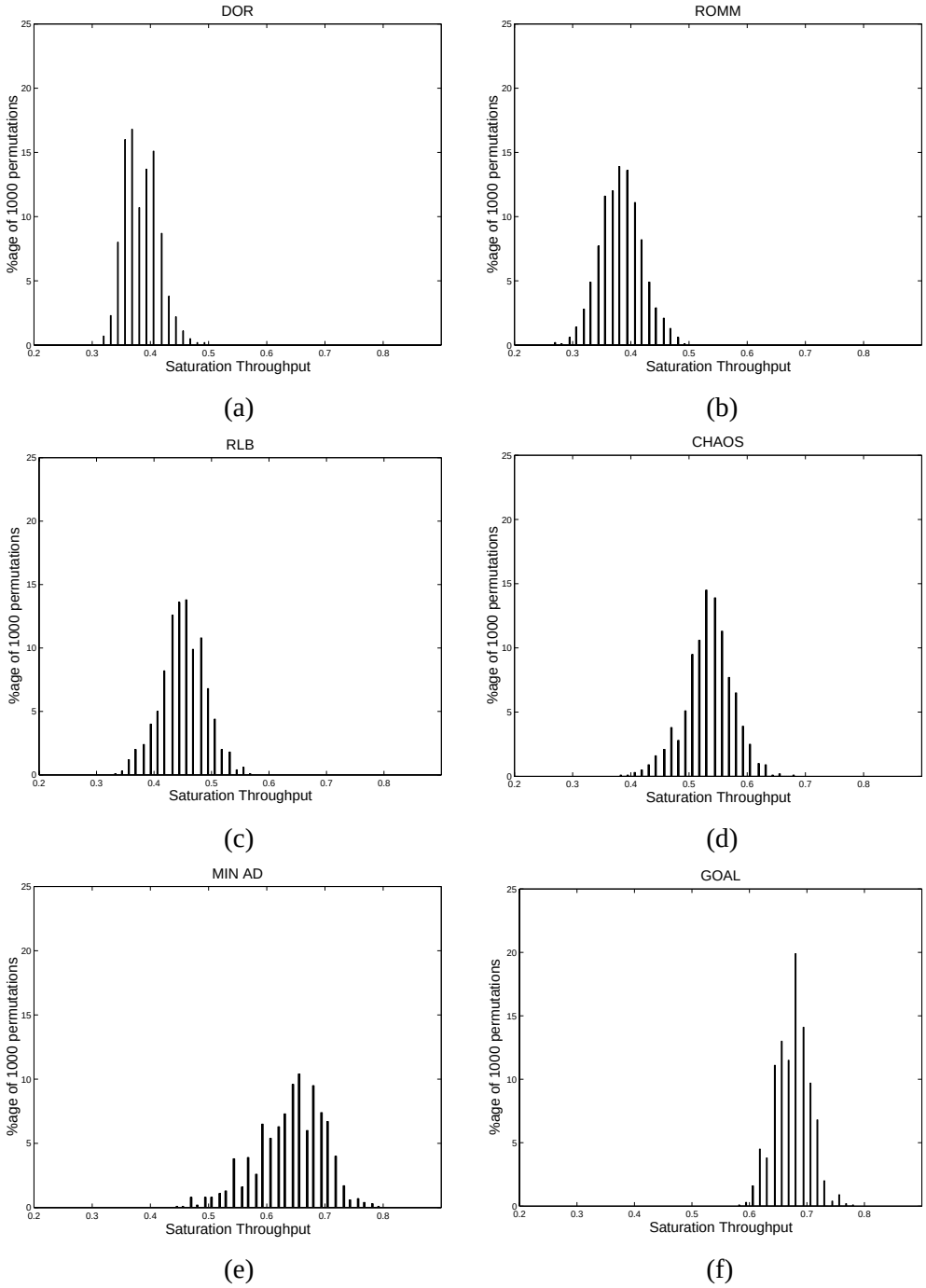


Figure 4.6: Histograms for the saturation throughput for 10^3 random permutations. (a) DOR, (b) ROMM, (c) RLB, (d) CHAOS (e) MIN AD (f) GOAL.

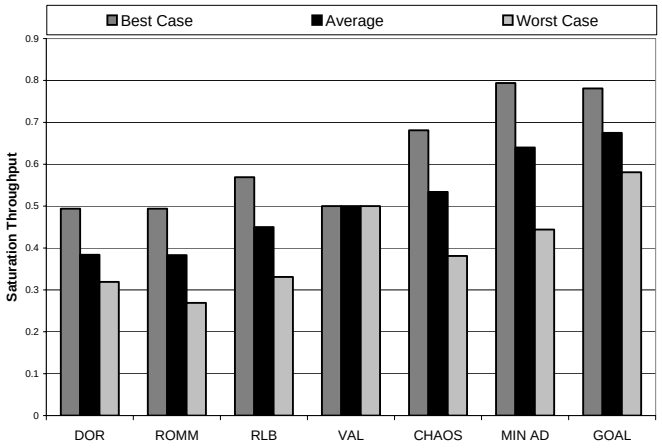


Figure 4.7: Best-case, Average and Worst-case Saturation Throughput for 10^3 random traffic permutations

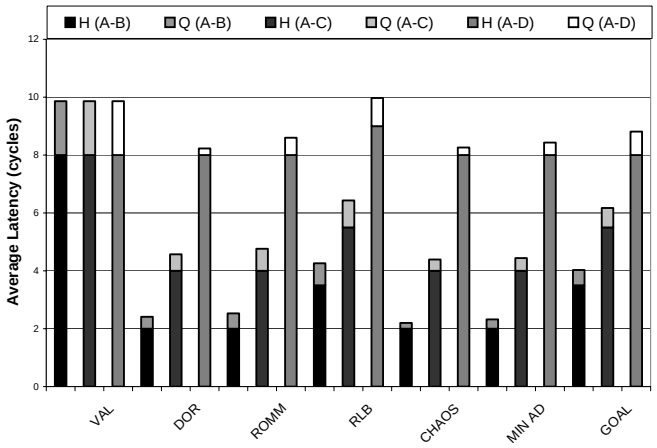
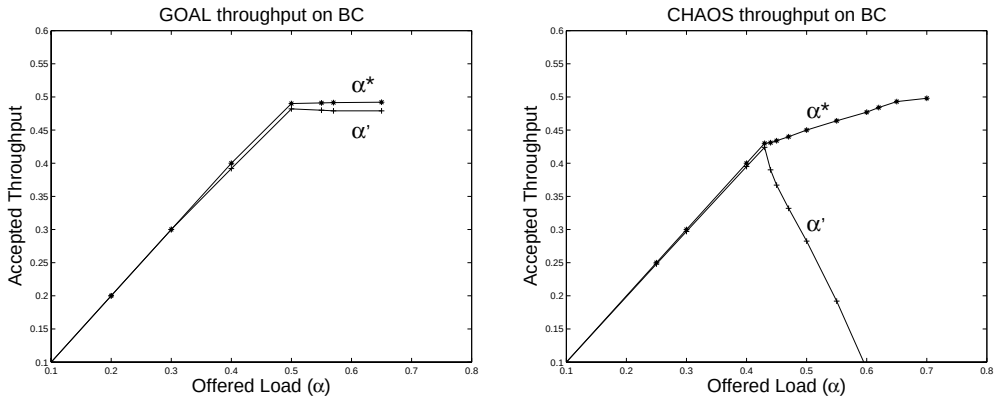


Figure 4.8: Average total — hop (H) and queueing (Q) — latency for 10^4 packets for 3 sets of representative traffic paths at 0.2 load



(a) (b)

Figure 4.9: Accepted Throughput for BC traffic on (a) GOAL and (b) CHAOS

nearly zero α'. However, GOAL is stable with accepted traffic, α', flat after saturation. The other five algorithms are also stable post saturation with age-based arbitration as shown in Figure 4.10.

It is worth noting that some algorithms presented in the literature such as those in Chapter 9 of [13] show that the accepted throughput degrades after saturation. This is because these schemes either use deadlock recovery techniques [1] or *strict* escape paths which drain the packets that may be involved in a deadlock. Since deadlock occurs frequently post saturation, the throughput of the network degrades to the bandwidth of the deadlock free lanes or escape channels. The four oblivious algorithms — VAL, DOR, ROMM and RLB — use deadlock avoidance, i.e., they achieve deadlock freedom by ensuring that the channel dependency graph (see Chapter 3 of [13]) of all the virtual channels used is acyclic. Hence, they are stable after saturation. MIN AD and GOAL use the $*$-channels as the deadlock free escape paths for packets that maybe involved in a deadlock in the fully adaptive *non*-$*$ channels. However, these escape paths are not *strictly* meant for packets involved in a potential deadlock in the *non*-$*$ channels, i.e. packets entering the $*$-channels can always go back to the *non*-$*$ ones and vice versa. Thus, none of the adaptivity is lost and the throughput is sustained post saturation.

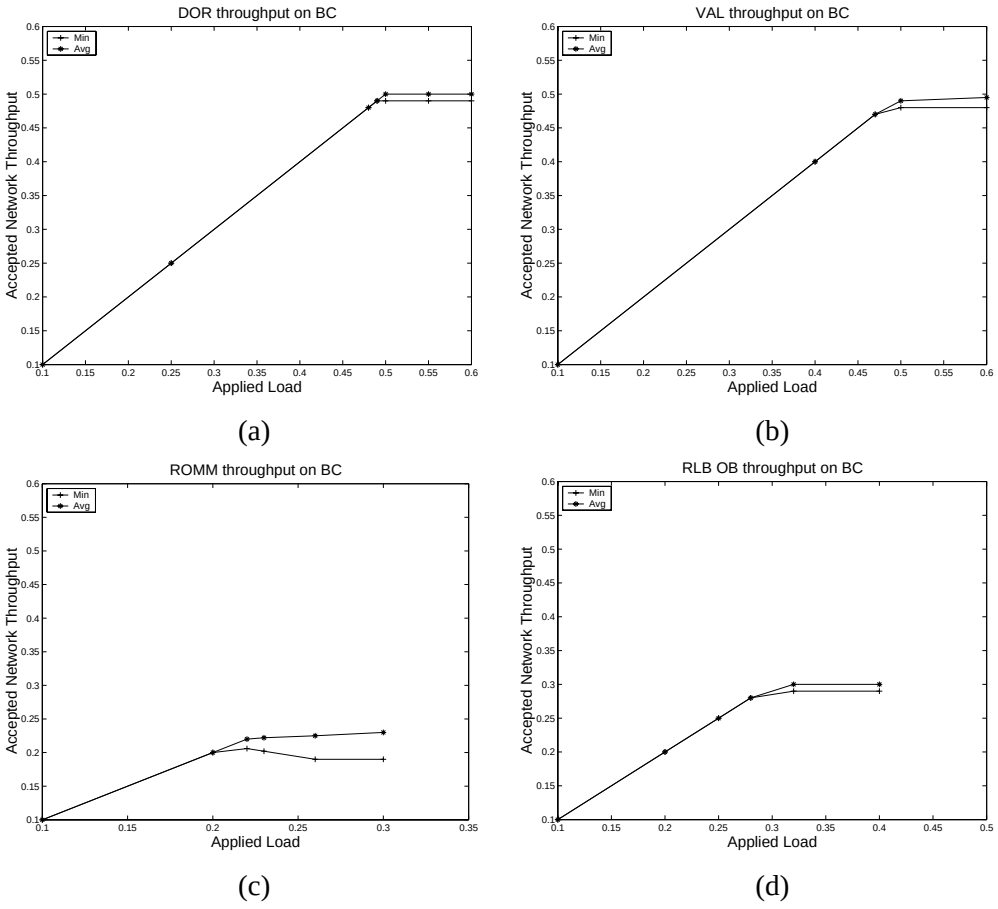


Figure 4.10: Accepted Throughput for BC traffic on (a) DOR (b) VAL (c) ROMM (d) Oblivious RLB algorithms — RLB & RLBth. The minimum throughput (Min or α') over all source-destination pairs remains flat post-saturation just like the average throughput (Avg or α^*).

4.2.5 Performance on Hot-Spot traffic

Occasionally a destination node in an interconnection network may become oversubscribed. This may occur in a switch or router due to a transient misconfiguration of routing tables. In a parallel computer such a *hot spot* occurs when several processors simultaneously reference data on the same node.

We evaluate the performance of five algorithms on hot-spot traffic by using a *hot-spot pattern* similar to that used in [3]. We first select a background traffic pattern, bit complement (BC), on which most of the algorithms give similar performance. On top of BC, five nodes[4] are selected which are five times more likely to be chosen as destinations than the other nodes. In the resulting matrix, Λ_{HS}, all rows sum to one, but the five columns corresponding to the five *hot-spot nodes* sum to five. Since the three adaptive algorithms — CHAOS, MIN AD and GOAL — and two oblivious algorithms — VAL and DOR — give similar performance on Λ_{BC}, we present results for these five on the resulting Λ_{HS} traffic.

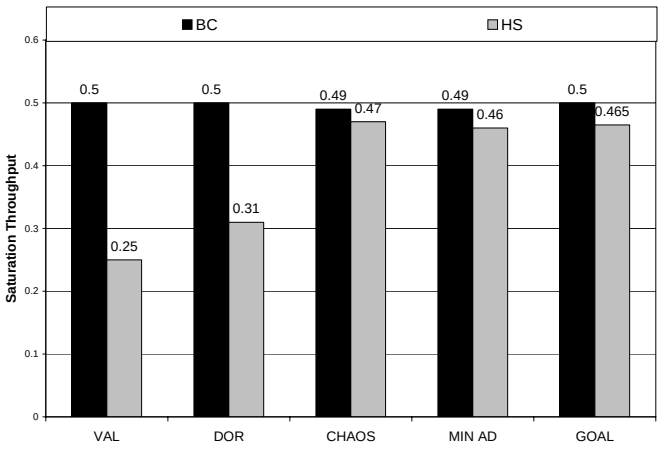


Figure 4.11: Saturation Throughput for the Hot-Spot traffic pattern and the background Bit Complement pattern.

Figure 4.11 shows the performance of each routing algorithm on the hot-spot pattern and the background BC pattern. The adaptive algorithms, CHAOS, MIN AD, and GOAL have similar performance on hot-spot traffic. They clearly outperform the oblivious algorithms because adaptivity is required to route around the congestion resulting from hot

[4]These five hot-spot nodes are chosen very close to each other to stress the adaptivity of the algorithms.

nodes. VAL gives throughput lower than 0.5 on hot-spot traffic because the traffic matrix is no longer admissible.

4.3 Summary

In this Chapter, we have introduced a load-balanced, non-minimal adaptive routing algorithm for torus networks, GOAL, that achieves high throughput on adversarial traffic patterns while preserving locality on benign patterns. GOAL matches or exceeds the throughput of Valiant's algorithm on adversarial patterns and exceeds the worst-case performance of CHAOS, RLB, and minimal routing by more than 40%. Like RLB, GOAL exploits locality to give $4.6\times$ the throughput of Valiant on local traffic and more than 30% lower zero-load latency than Valiant on uniform traffic.

GOAL globally balances network load by obliviously choosing the direction of travel in each dimension, in effect randomly picking a *quadrant* in which to transport the packet. The random choice of directions is made using distance-based weights that exactly balance load in each dimension. Once the quadrant is selected, GOAL locally balances load by routing adaptively within that quadrant. GOAL employs a new algorithm for deadlock freedom based on an extension of the $*$-channels approach [16] (which is used for minimal routing) to handle the non-minimal case. This provides deadlock freedom with just three virtual channels per physical channel. Unlike CHAOS, GOAL is deterministically livelock free since within the selected quadrant distance to the destination is monotonically decreased with each hop.

We compare GOAL to the four oblivious algorithms from Chapter 3 — VAL, DOR, ROMM, and RLB — and two *state-of-the-art* adaptive routing methods — CHAOS and MIN AD — and present a comparison in terms of throughput, latency, stability, and hot-spot performance. This evaluation includes throughput histograms on random permutations and latency histograms for CHAOS and MIN AD that have not been previously reported. GOAL provides the highest throughput of the seven algorithms on four adversarial patterns and on the average and worst-case of $1,000$ random permutations. The cost of this high worst-case throughput is a degradation on local traffic. GOAL achieves only 58% and 76% of the throughput of minimal algorithms on nearest-neighbor traffic and uniform traffic,

respectively. Due to oblivious misrouting, GOAL also has 40% higher latency on random traffic than the minimal algorithms; however it has 38% lower latency than VAL. Finally, we analyze network performance beyond saturation throughput and show for the first time that due to fairness issues CHAOS is unstable in this regime for certain permutations.

Sub-optimal performance on benign traffic remains a thorn in the flesh for any kind of oblivious misrouting. In the next chapter we explore how adaptive decisions to misroute can alleviate this problem for GOAL.

Chapter 5

Globally Adaptive Load-balancing

We have demonstrated the superior performance on *average* of adaptive algorithms over the oblivious ones. In this chapter, we focus on adaptive algorithms. We introduce two new methods of globally adaptive load-balanced routing that we refer to as GAL [44] and CQR [42]. Unlike previous adaptive routing algorithms that make routing decisions based on local information, the globally adaptive load balanced algorithms sense congestion globally using either segmented injection queues or channel queues to decide the directions to route in each dimension. They further load balance the network by routing in the selected directions adaptively. The use of approximate global information enables GAL and CQR to achieve the performance (latency and throughput) of minimal adaptive routing on benign traffic patterns, while performing as well as the obliviously load-balanced GOAL on adversarial traffic.

5.1 GAL: Globally Adaptive Load-balanced routing

In the previous chapters, we have seen that no adaptive algorithm discussed thus far yields optimal throughput on both benign patterns (such as UR, NN) and adversarial patterns (such as TOR). An ideal algorithm routes benign traffic minimally, and load balances adversarial traffic across the channels of the network. In this section, we present a routing algorithm, GAL, that, using approximate global information, changes its routing decision from minimal to non-minimal. We first consider this algorithm for routing a benign traffic,

NN, and an adversarial traffic, TOR, on an 8 node ring and then extend GAL to higher dimensional torus networks.

5.1.1 GAL on a ring

In order to sense approximate global information on an 8 node ring, consider 8 sets of injection queues at each source — each set corresponding to a destination. Each set comprises two queues — a minimal queue and a non-minimal one. When a packet is received from the terminal node, it is enqueued in the minimal queue for the packet's destination, if the length of that queue is less than a threshold, T, otherwise it is enqueued in the shorter of the two queues (Figure 5.1). With this scheme, NN is always routed minimally, i.e., the threshold of the minimal queues is never reached and the throughput is optimal at $\theta = 2$. When we route TOR using this scheme and increase the injected load, initially all the packets are routed minimally. However, when 0.33 load is accepted, the minimal direction gets saturated and the threshold is surpassed. The algorithm now becomes non-minimal and starts sending the rest of the traffic the long way around.

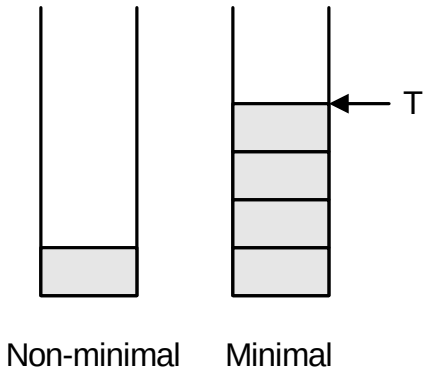


Figure 5.1: A packet is injected into the non-minimal injection queue when the minimal injection queue for its destination reaches a threshold.

As illustrated in Figure 5.2, this scheme routes TOR traffic minimally at low loads. Only when the minimal channels become saturated at a load of 0.33, does the minimal queue length exceed the threshold. At this point, some traffic is routed non-minimally. As the load is increased, more traffic is routed non-minimally. At saturation, the load is exactly

balanced with $5/8$ of the traffic routing minimally and $3/8$ non-minimally. Thus, by making the global routing decision (minimal or non-minimal) using global congestion information (sensed by the injection queues) this scheme is able to achieve optimal performance on both benign (NN) and difficult (TOR) traffic patterns, something no previously published routing algorithm has achieved.

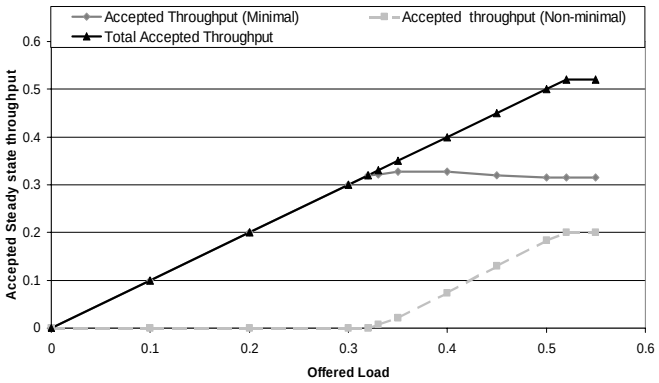


Figure 5.2: GAL on tornado traffic on an 8 node ring

5.1.2 GAL routing in higher dimensional torus networks

In a torus with n dimensions, we divide the sets of possible paths from a source s to a destination d into 2^n *quadrants*, one for each combination of directions (one direction per dimension). GAL provides a separate set of injection queues for each *quadrant*. Figure 5.3 shows the setup for each node for a 2-D network. There is an infinite source queue that models the network interface to the terminal node. There are $S = k^2$ sets, each set comprising 4 injection queues. When the injection unit receives a packet from the source queue, the injection set is determined by its destination. Within that set, one injection queue (and therefore the quadrant to route in) is selected as follows: the queue associated with the quadrant having the smallest distance from the source to the destination whose occupancy is less than a threshold, T, is chosen. If all the queues have surpassed their threshold, then the shortest queue is selected. Once the quadrant is selected, the packet is routed adaptively within that quadrant just as in GOAL.

As shown is Figure 5.3, our implementation of GAL, like GOAL, requires 3 virtual

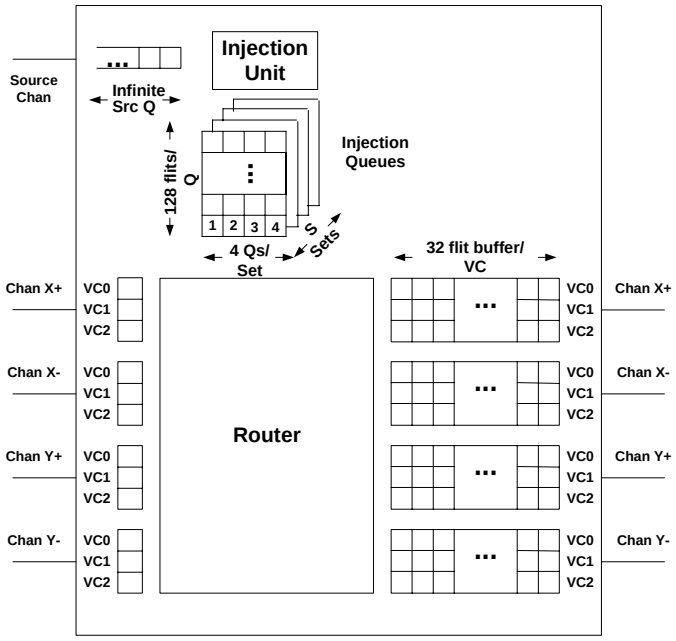


Figure 5.3: GAL node for a 2-D torus

channels (VCs) per unidirectional physical channel to achieve deadlock freedom in the network. The argument presented in Appendix B also applies to GAL, proving that GAL is free of deadlocks. Once a quadrant has been selected for a packet, the packet monotonically makes progress in that quadrant, providing deterministic freedom from livelock.

5.2 Performance evaluation of GAL

We next compare the performance of GAL with the 3 adaptive algorithms, CHAOS, MIN AD and GOAL, based on the six figures of merit we have used thus far.

All the assumptions regarding the node model for GAL stated in Section 5.1 hold for the experimental setup. Figure 5.3 shows the experimental setup for a node for GAL. Each node has an infinite source queue to model the network interface and there are $S = 64$ sets of injection queues for an 8×8 torus. Each set has 4 injection queues (with 128 flits each) corresponding to each of the 4 quadrants. The router is output queued and queues packets into one of the 3 (32 flit deep) VC buffers per physical output channel. Each packet is assumed to be one flit long to separate the routing algorithm study from flow control

issues. The threshold value, T, for GAL is kept at 2 flits for each minimal injection queue. The total buffer resources are held constant across all algorithms, i.e., the product of the number of VCs and the VC channel buffer depth is kept constant.

5.2.1 Throughput on benign and hard traffic

We first compare the throughput of the four algorithms on the benign and adversarial traffic patterns described in Table 2.2. The two benign traffic patterns are shown in Figure 5.4 while the four adversarial patterns are shown in Figure 5.5 with an expanded vertical scale. The figures show that GAL is the only algorithm that gives best performance on both these categories of traffic patterns. It becomes a GOAL-like load balancing algorithm and matches the throughput of GOAL on adversarial traffic. At the same time it behaves minimally on benign patterns and matches the performance of MIN AD on them.

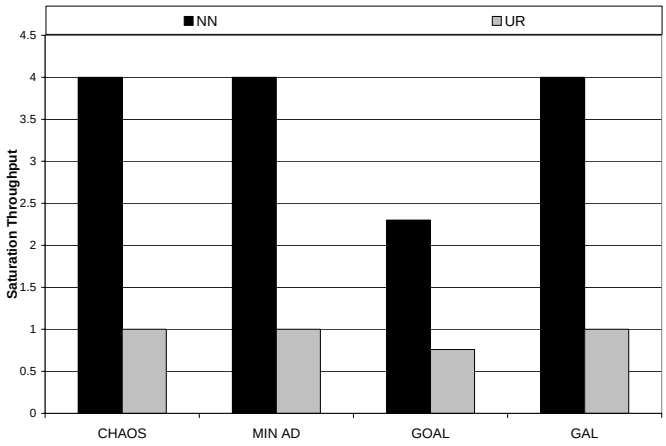


Figure 5.4: Comparison of saturation throughput on benign traffic on an 8-ary 2-cube

5.2.2 Throughput on random permutations

In order to evaluate the performance of the algorithms for the *average* traffic pattern, we measured the performance of each algorithm on $1,000$ random permutations as we did in the previous chapter.

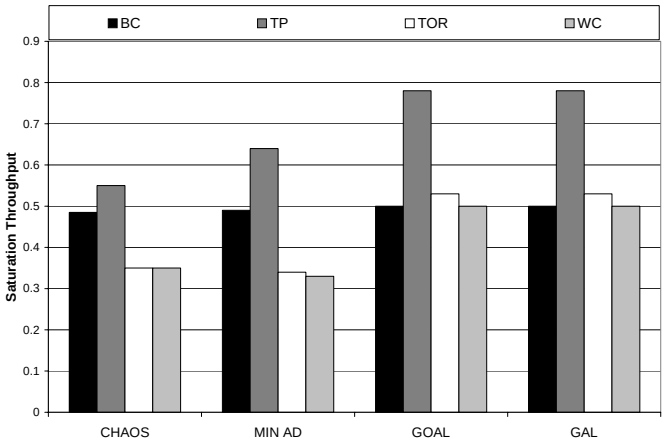
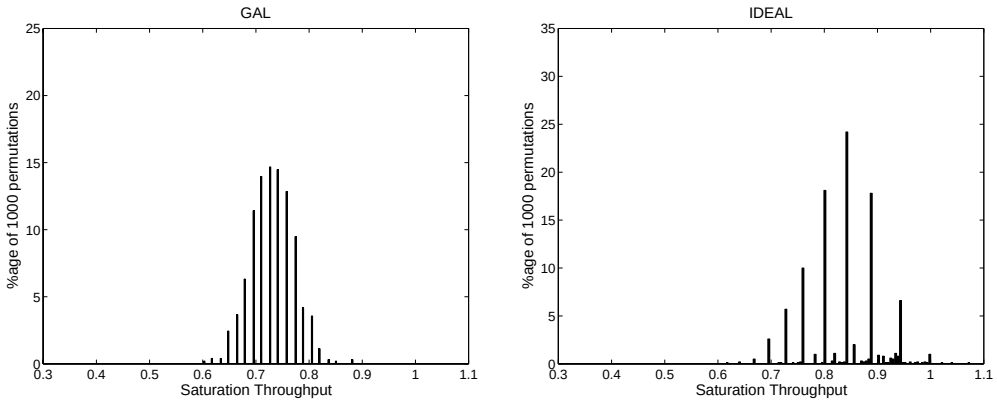


Figure 5.5: Comparison of saturation throughput on adversarial traffic on an 8-ary 2-cube



(a) (b)

Figure 5.6: Histograms for the saturation throughput for 10^3 random permutations. (a) GAL, (b) Ideal.

The histogram of saturation throughput over these permutations for GAL is shown in Figure 5.6(a). The histogram in Figure 5.6(b) shows the ideal throughput for each of the random permutations evaluated by solving a maximum concurrent flow problem over the traffic patterns constrained by the capacity of every link as described in [38]. The best, average and worst-case throughput in this experiment for each of the algorithms and the ideal throughput values for the sampled permutations are presented in Figure 5.7.

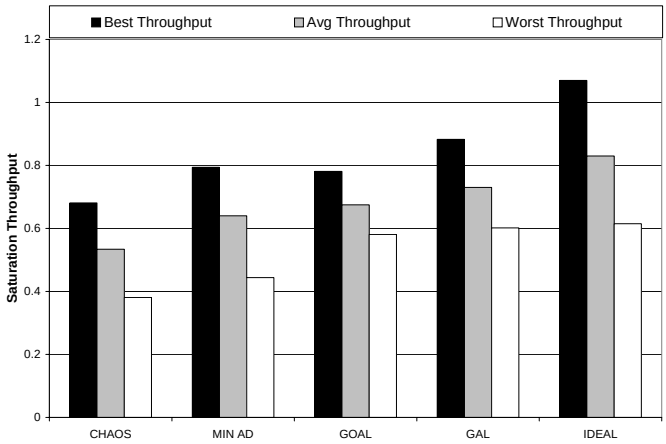


Figure 5.7: Best-case, Average and Worst-case Saturation Throughput for 10^3 random traffic permutations

The figure shows that over the $1,000$ permutations, GAL has top performance in the best, average and worst-case throughput — exceeding the worst-case throughput of GOAL and achieving 98% of the ideal worst-case throughput for this sampling. GAL gives 89% average throughput compared to the ideal average throughput. Part of this gap is because the ideal throughput is evaluated assuming ideal flow control while GAL uses a realistic flow control mechanism.

5.2.3 Latency at low loads and hot-spot traffic

GAL performs optimally on the low-load latency experiment described in the previous two chapters. As illustrated in Figure 5.8, at a low load of 0.2 for UR traffic, the non-minimal algorithms, GAL and CHAOS, behave just like MIN AD and deliver packets with optimal latency. In contrast, GOAL sacrifices some of the locality due to its oblivious misrouting.

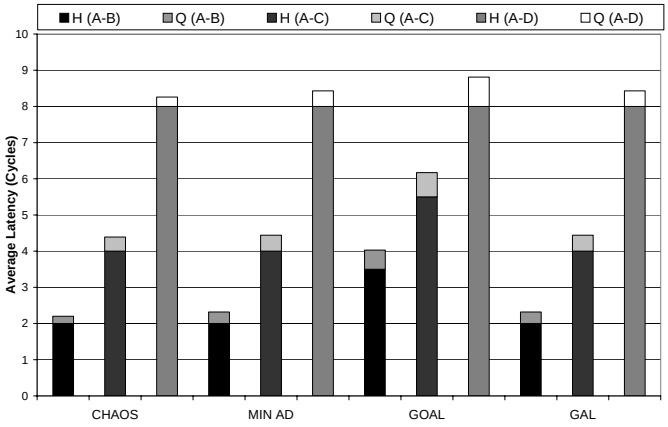


Figure 5.8: Average total — hop (H) and queuing (Q) — latency for 10^4 packets for 3 sets of representative traffic paths at 0.2 load

Figure 5.9 shows the performance of each routing algorithm on the hot-spot pattern and the background BC pattern. CHAOS, MIN AD, and GOAL have similar performance on hot-spot traffic. GAL once again performs the best among all the algorithms, outperforming MIN AD by 6.5%. This is because the minimality inherent in MIN AD forces it to route some fraction through the clustered hot nodes. The globally load balancing GAL senses this congestion and routes traffic that would otherwise have to go headlong through the hot nodes with MIN AD, the long way around.

5.2.4 Stability

We have discussed the importance of stability of a routing algorithm in Chapter 4. An algorithm is stable if the accepted throughput remains constant even as the offered load is increased beyond the saturation point. Achieving stability can be challenging for non-minimal, adaptive routing algorithms because it is difficult to differentiate between congestion caused by load imbalance and congestion caused by saturation of a load balanced network. In the former case, rerouting traffic, possibly non-minimally, can alleviate this imbalance and increase throughput. However, in the load balanced, but saturated case, rerouting traffic may introduce imbalance and decrease throughput. GAL solves this instability problem by measuring changes in throughput and adjusting its queue thresholds.

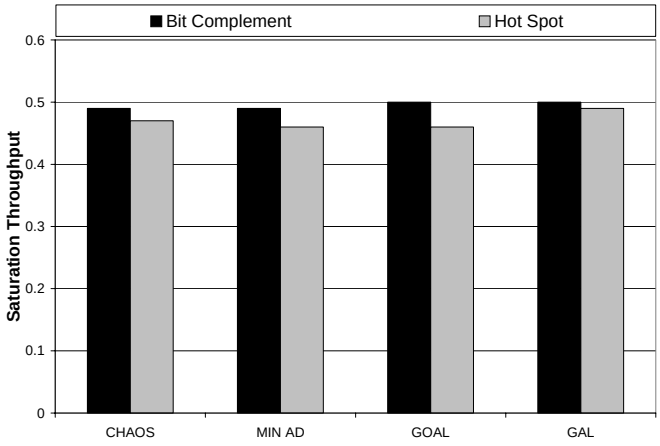


Figure 5.9: Saturation Throughput for the Hot-Spot traffic pattern and the background Bit Complement pattern

Changing these thresholds allows control over the fraction of packets sent non-minimally, which, in turn, is used to maintain stability. Before describing the mechanisms and rules for these adjustments, we examine a simple example to demonstrate the necessity of controlling the fraction of packets sent non-minimally.

Consider the case of routing UR traffic with GAL, but using a fixed queue threshold. For an offered load beyond saturation, traffic will initially be routed minimally as the minimal queues are empty. Since the network cannot sustain all the traffic, backpressure will cause packets to back up into the minimal source queues. Although minimal routing perfectly balances load for UR traffic, the occupancy of these queues will eventually exceed the fixed threshold and some traffic will be routed non-minimally. At this point, throughput will begin to drop as channel bandwidth is wasted by non-minimal packets (Figure 5.10). If, instead, packets had not been routed non-minimally, throughput would have remained stable.

To control the number of packets routed non-minimally, GAL dynamically varies each threshold T between a minimum value $T_{\min}$ and a maximum value $T_{\max}$ set by the queue depth. This is done independently for each queue set. Then, the threshold value limits the fraction of packets routed non-minimally — increasing T decreases the likelihood that a packet is placed in a non-minimal queue and, when $T = T_{\max}$, all packets are routed

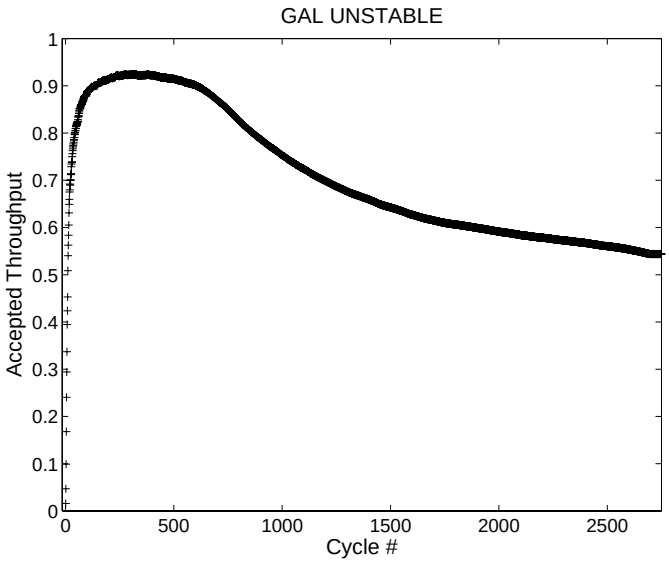


Figure 5.10: GAL with fixed threshold is unstable for UR traffic at 1.1 offered load

minimally. To ensure stability, T is incremented whenever a drop in throughput is observed. Revisiting our stability example, for any $T < T_{\max}$, some fraction of packets will be routed non-minimally, reducing throughput and, thus, triggering an increase in T. This continues until $T = T_{\max}$ and stability is achieved. Because the traffic pattern may later change so that non-minimal routing is again productive, T is decremented whenever throughput is constant or increasing.

We approximate the throughput for each source-destination pair by instrumenting a running total of the drained packets over a window of n_1 cycles, $\overline{D}$, from all the injection queues in the corresponding set at each source node. Then, the change in this drainage, ΔD, is computed over n_2 cycles:

$$\overline{D}_t = \sum_{i=0}^{n_1-1} D_{t-i} \qquad\qquad \Delta D_t = \overline{D}_t - \overline{D}_{t-n_2},$$

where D_t is the number of departures from the queues in that set at time t. The estimated

throughput is then used to update the threshold each n_2 cycles:

$$T_{i+1} = \begin{cases} \min(T_i + 1, T_{\max}) & \text{if } \Delta D_i < 0, \\ \max(T_i - 1, T_{\min}) & \text{if } \Delta D_i \geq 0. \end{cases}$$

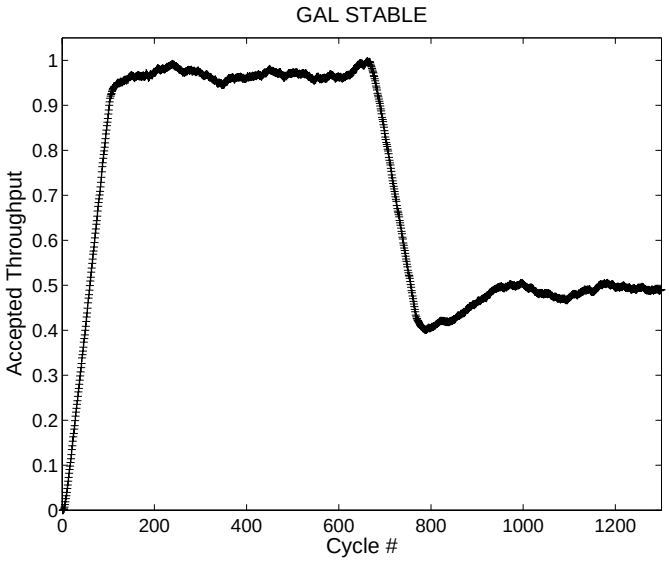


Figure 5.11: GAL is stable with threshold variation post saturation. UR traffic at 1.1 load switches to TOR at 0.55 load on Cycle 600.

Figure 5.11 shows an example of threshold adjustment. The test setup is as follows: we impose UR traffic at 1.1 injection load and then switch to TOR at 0.55 injection load on cycle 600. The parameter values used are $T_{\min} = 2$, $T_{\max} = 128$, $n_1 = 50$ and $n_2 = 20$. Both loads are post saturation, but the accepted throughput is stable. This is because when ΔD becomes negative, the threshold is incremented. This keeps happening as more packets are injected. Finally, when $T = T_{\max}$, the algorithm routes the UR traffic strictly minimally and the accepted throughput almost reaches the optimal value of 1.0. After cycle 600, the traffic changes to TOR at 0.55 load. Still $\Delta D < 0$ and hence, $T = T_{\max}$. Now GAL acts like a minimal algorithm routing TOR traffic and so the accepted throughput drops much below the optimal value of 0.53. After the lowest point, the throughput starts to level out again ($\Delta D = 0$) and hence, T is decremented. As a result, ΔD rises as traffic is sent non-minimally. This continues until $T = T_{\min}$ and the accepted throughput is stable for TOR

traffic. Thus, GAL with the threshold adjustment scheme is stable for both benign and hard traffic patterns.

5.3 Summary of GAL

To summarize this chapter thus far, we have introduced Globally Adaptive Load-balanced Routing (GAL), a new algorithm for routing in k-ary n-cube networks that adapts globally by sensing global congestion using queues at the source node. By sensing global congestion, GAL adapts to route minimally when it can, and non-minimally only when the minimal channels become congested. Previous adaptive algorithms are unable to adapt in this manner. GAL also adapts locally, sensing channel congestion using channel queues.

At low loads and on benign traffic patterns, GAL routes all traffic minimally and thus, matches the low latency and high throughput of minimal routing algorithms on such *friendly* traffic. On adversarial traffic patterns, GAL routes minimally at low loads and then switches to non-minimal routing as congestion is detected by the injection queues. At saturation, GAL matches the throughput of optimally load-balanced oblivious routing. This combines the best features of minimal algorithms (low latency at low load) and obliviously load balanced algorithms (high saturation throughput).

While GAL performs better than any other known routing algorithm on a wide variety of throughput and latency metrics, there are four serious issues with GAL. First, it has very high latency once it starts routing traffic non-minimally. Second, it is slow to adapt to changes in traffic. Third, it requires a complex method to achieve stability. Finally, it is complex to implement. These issues are all related to GAL's use of injection queue length to infer global congestion. In the rest of this chapter, we introduce channel queue routing (CQR - pronounced "seeker") [42] which matches the ability of GAL to achieve high throughput on both local and difficult patterns but overcomes the limitations of GAL. CQR overcomes these limitations by using *channel* queues rather than *injection* queues to estimate global congestion. CQR gives much lower latency than GAL at loads where non-minimal routing is required. It adapts rapidly to changes in traffic, is unconditionally stable, and is simple to implement.

5.4 Motivation for Channel Queue Routing (CQR)

For *adversarial* traffic patterns, the focus of an adaptive routing algorithm must change as load is increased. At low loads, packets should be routed minimally in order to minimize delays. As loads increase, minimizing delay requires shifting some packets to non-minimal routes in order to balance channel load. In this section, we derive the optimal adaptive routing algorithm for the tornado traffic pattern on the 8-node ring network and compare it to the performance of GAL. While GAL matches this optimal algorithm at both very low loads and loads near saturation, it transitions too slowly from minimal to non-minimal at intermediate loads, resulting in delays approximately 100 cycles more than optimal. This behavior is representative of GAL on all adversarial patterns because it does not misroute until the delays of the network are already large.

In order to understand how tornado traffic should be routed for optimal delay, we present an approximate theoretical model from queueing theory.

5.4.1 A Model from queueing theory

Queueing networks have been studied extensively in literature. A popular way to study a network of queues is to use the Jackson open queueing network model [18]. This model assumes a *product-form*[1] network in which each queue is an independent $M/M/1$ queue. However, it differs from the standard model by assuming that service times, instead of being constant, are exponentially distributed with unit mean. Based on an idea first considered by Kleinrock [23], Mitzenmacher [29] observed that assuming each queue to be an independent $M/D/1$ queue is a good approximation. While the approximate model no longer strictly remains a product-form network, the results prove accurate in practice.

For our queueing model, we assume each queue to be an independent $M/D/1$ queue with arrival rate α and service rate 1. The queueing delay of a packet in such a queue is given by [2],

$$Q(\alpha) = \frac{1}{2(1-\alpha)}$$

[1] A network is product-form if in equilibrium distribution each queue appears as though it were an independent process with Poisson arrivals.

If a packet traverses a linear network of m such queues, it incurs a queueing delay due to m queues and a hop delay of m. Its total delay is approximated by

$$L^m(\alpha) = m(Q(\alpha) + 1).$$

We shall use this approximate model in the rest of this discussion.

5.4.2 Routing tornado traffic for optimal delay

When routing TOR optimally, there are two classes of packets — those that are routed minimally and those sent along the non-minimal path. Let the injection load for each node be α and the rates at which each node sends packets minimally and non-minimally be x_1 and x_2, respectively. Then, below saturation,

$$\alpha = x_1 + x_2 \tag{5.1}$$

Packets sent minimally (non-minimally) must traverse 3 (5) $M/D/1$ queues each with an arrival rate of 3α (5α).[2] Hence, the delay encountered by a packet sent minimally is given by

$$D_1(x_1) = L^3(3x_1) = \frac{3}{2(1-3x_1)} + 3 \tag{5.2}$$

and the delay of a non-minimally routed packet is given by

$$D_2(x_2) = L^5(5x_2) = \frac{5}{2(1-5x_2)} + 5. \tag{5.3}$$

Then the average delay is simply a weighted sum of these delays,

$$D(\alpha) = \frac{1}{\alpha}[x_1 D_1(x_1) + x_2 D_2(x_2)]. \tag{5.4}$$

We now analyze the fraction of traffic sent minimally and non-minimally to minimize delay as described in [2].

[2] At steady state and due to the symmetry of the network and the traffic pattern, we assume all nodes route traffic identically.

Theorem 5. *For tornado traffic on an* 8 *node ring, the optimal fraction,* x_1*, of total traffic,* α*, sent minimally is given by*

$$x_1 = \begin{cases} \alpha & \text{for } \alpha \leq 0.13 \\ (\alpha + 0.13)/2 & \text{for } 0.13 < \alpha \leq 0.53 \\ 0.33 & \text{for } \alpha > 0.53 \end{cases}$$

Proof. For minimum delay, each node should send all traffic along the minimal path until the incremental delay of routing minimally is greater than that of routing non-minimally. The routing will be strictly minimal as long as

$$\frac{\partial D_1(\alpha)}{\partial x_1} \leq \frac{\partial D_2(0)}{\partial x_2} \tag{5.5}$$

Solving Equation 5.5, we get $\alpha \leq 0.13$. Hence, the switch point from strictly minimal to non-minimal occurs at $\alpha_s = 0.13$.

Once each node starts to send traffic non-minimally ($\alpha > \alpha_s$), the optimal fraction of load sent minimally (x_1) and non-minimally (x_2) will be such that the incremental delay along either directions is the same:

$$\frac{\partial D_1(x_1)}{\partial x_1} = \frac{\partial D_2(x_2)}{\partial x_2}. \tag{5.6}$$

Solving Equations 5.6, 5.1 yields $x_1 = (\alpha + \alpha_s)/2$ for $\alpha > \alpha_s$. Finally, after the network saturates at $\alpha = 0.53$, $x_1 = 0.33$ and $x_2 = 0.2$. The plots for the accepted throughput along minimal and non-minimal paths are shown in Figure 5.12. □

Substituting the values of x_1 and x_2 in Equations 5.2, 5.3, and 5.4, we get the average minimal, non-minimal and overall delay of the packets according to our model. As shown in Figure 5.13, the average delays along both paths are similar and give a low overall average latency.

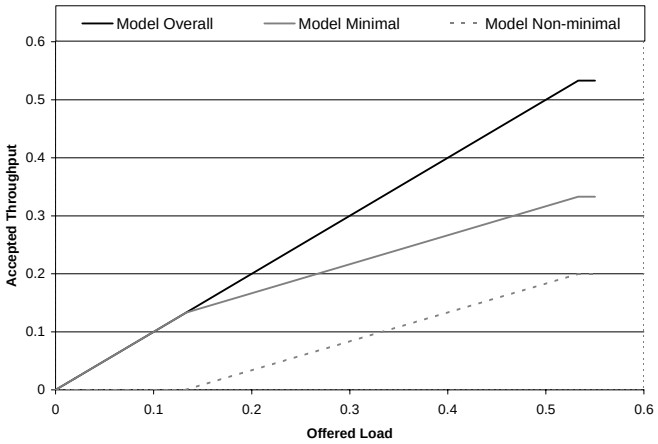


Figure 5.12: The optimal fraction of traffic in the minimal and non-minimal paths for tornado traffic on an 8 node ring

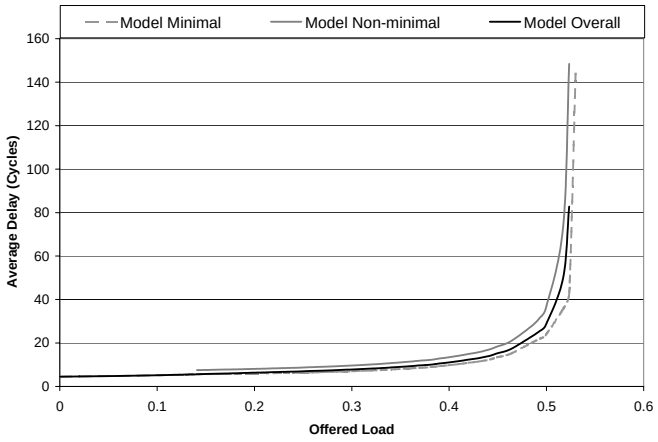


Figure 5.13: Optimal minimal, non-minimal and overall latency of the theoretical model for tornado traffic on an 8 node ring

5.4.3 GAL's latency on tornado traffic

GAL routes a packet minimally if the occupancy of the minimal injection queue associated with the packet's destination is below a threshold. Otherwise, the threshold is exceeded and the packet is injected into the non-minimal injection queue and routed non-minimally.

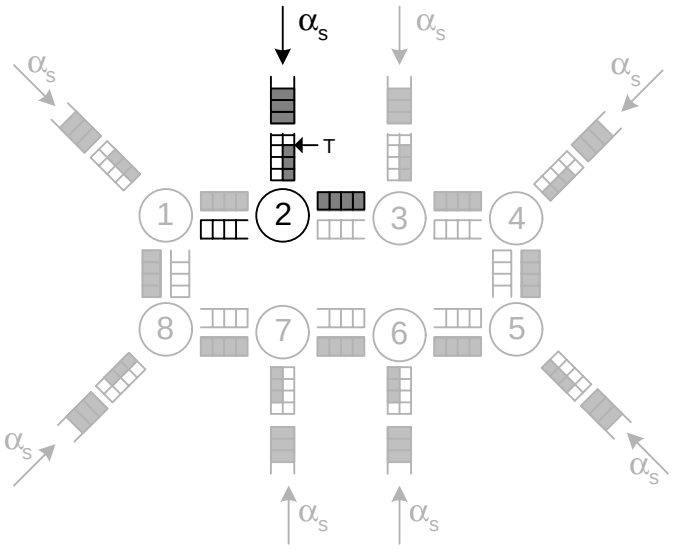


Figure 5.14: GAL on tornado traffic on an 8 node ring at the point when it switches from minimal to non-minimal. Only 1 set of injection queues corresponding to the destination is shown.

With this approach, GAL routes packets minimally until the capacity along the minimal path is saturated. When this happens, the queues along the minimal path start to fill up and the network backpressure implicitly transfers this information to the injection queues in the source node. When the occupancy of the minimal injection queue surpasses a threshold, T, packets are injected into the non-minimal injection queue and the routing switches from strictly minimal to non-minimal. This switching does not happen until all the queues along the minimal path are completely filled as shown in Figure 5.14. Hence, the load at which this switch occurs is simply the saturation load for strictly minimal routing of the tornado pattern and is given by $\alpha_s = 0.33$. Figure 5.2 shows how GAL starts to send traffic non-minimally only after the injection load, $\alpha \geq 0.33$. The accepted throughput at saturation is still the optimal value of 0.53.

However, the subtle point to note is that while GAL routing is throughput-optimal on the tornado pattern, delay in the switching from strictly minimal to non-minimal incurs a high price as far as the latency is concerned.

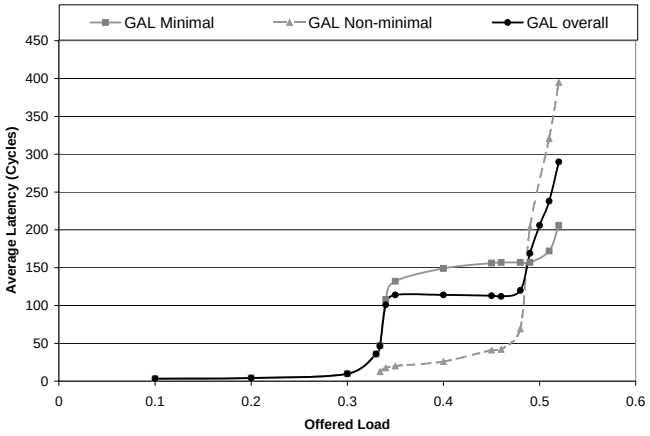


Figure 5.15: Latency-load plot for GAL on tornado traffic on an 8 node ring

Figure 5.15 shows the average minimal, non-minimal and overall packet delay for GAL. Since GAL routes minimally all the way to the saturation point for the strictly minimal routes, the latency incurred by the minimal packets after a load of 0.33 is very high. This causes the average overall packet delay, which is a weighted average of the minimal and non-minimal delays, to be of the order of 100s of cycles for an 8 node network much before the network saturation point of 0.53.

5.4.4 Routing tornado traffic with CQR

We observe that the reason GAL performs so poorly on tornado traffic is that it waits too long to switch its policy from strictly minimal to non-minimal. It does so because its congestion sensing mechanism uses the occupancy of the injection queues which is not very responsive to network congestion. CQR addresses this problem by sensing network load imbalance using its channel queues while at the same time relying on the network's implicit backpressure to collect approximate global information from further parts of the network. Consider the highlighted node 2 in Figure 5.16. For the tornado traffic pattern, the minimal and non-minimal queues for this node are the clockwise and counterclockwise outgoing

queues, respectively. The instantaneous occupancy of both these queues are labeled as q_m and q_{nm}. If routed minimally (non-minimally), a packet will traverse 3 (5) queues, each with occupancy q_m (q_{nm}). In order to keep the delay along both directions balanced, CQR routes minimally as long as $3q_m \leq 5q_{nm}$, else it routes non-minimally. Such a congestion sensing scheme is more responsive than the one used in GAL and unlike GAL, the switch occurs much before all the minimal queues are filled as shown in Figure 5.16. We call this routing method Channel Queue Routing (CQR).

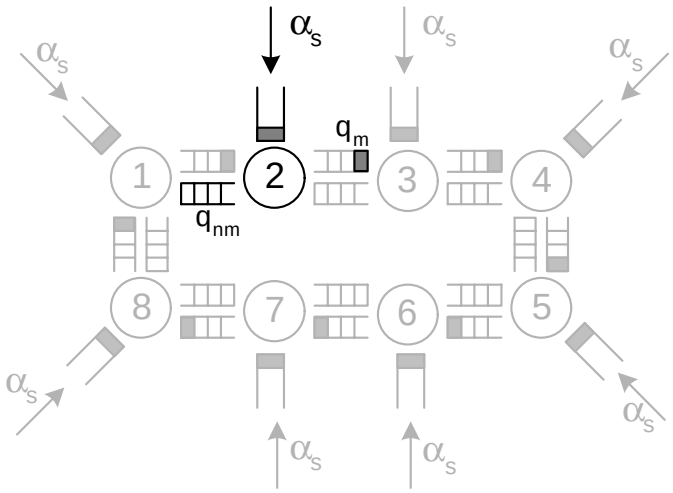


Figure 5.16: CQR on tornado traffic on an 8 node ring at the point when it switches from minimal to non-minimal

Figure 5.17 shows the fraction of traffic sent the minimal and non-minimal way as the injected load increases. The switch point, $\alpha_s = 0.12$, is very close to the one that we derived in our model. Since the switch point for CQR on TOR is near-optimal, it gives a much smaller average overall delay as shown in Figure 5.18. The latency-load curve is very similar to the one we derived in Figure 5.13. It should be noted that the plots do not exactly match the model that we described as the model itself is based on an approximation and the queues are not strictly a network of independent $M/D/1$ queues.

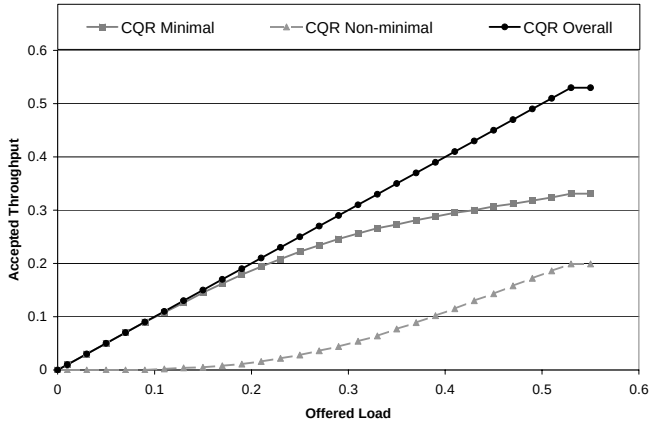


Figure 5.17: CQR throughput on tornado traffic on an 8 node ring

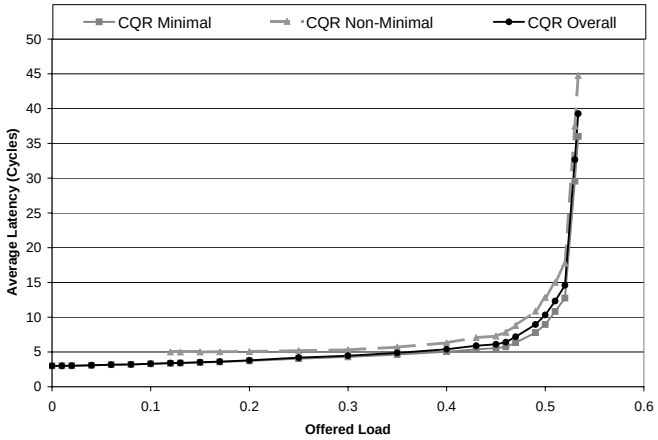


Figure 5.18: Latency-load plot for CQR on tornado traffic on an 8 node ring

5.5 Channel Queue Routing

Like in GAL, CQR adaptively selects a *quadrant* to route in for each packet. Suppose the source node is $s = \{s_1, s_2, \ldots, s_n\}$ and the destination node is $d = \{d_1, d_2, \ldots, d_n\}$, where x_i is the coordinate of node x in dimension i. We compute a minimal direction vector $r = \{r_1, r_2, \ldots, r_n\}$, where for each dimension i, we choose r_i to be $+1$ if the short direction is clockwise (increasing node index) and -1 if the short direction is counterclockwise (decreasing node index). Choosing a quadrant to route in simply means choosing a *quadrant vector*, QR, where for each unmatched dimension i we could choose $QR_i = r_i$ ($QR_i = -r_i$) if we want to route minimally (non-minimally) in that dimension. In order to get approximate quadrant congestion information just like in the one dimension case, each node records the instantaneous occupancies of its outgoing channel queues along both directions ($+$ and $-$) in each dimension. Then the congestion, Q_j, for quadrant j is approximated by the length of the shortest outgoing queue for that quadrant. If the hop count for Q_j is H_j, then H_jQ_j is approximately the delay the packet will incur in Q_j. To balance delay over all quadrants, we must make the H_jQ_j's equal for all j. In order to achieve this, CQR selects the quadrant j corresponding to the minimum H_jQ_j.

For instance, Figure 5.19 shows a portion of an 8-ary 2-cube network. Source $(0, 0)$ wants to route a packet to destination node $(2, 3)$. There are 4 choices of quadrants, $Q_1(++)$, $Q_2(+-)$, $Q_3(-+)$, and $Q_4(--)$ with hop counts 5, 8, 9, and 11, respectively, for this source-destination pair. The source node records the occupancy, q_a, q_b, q_c, and q_d of each outgoing channel. The congestion of the minimal quadrant Q_1 is approximated by $\min(q_a, q_b)$. Analogously, the congestion of quadrants Q_2, Q_3, and Q_4 are approximated by $\min(q_a, q_d)$, $\min(q_b, q_c)$, and $\min(q_c, q_d)$, respectively. CQR selects the quadrant j corresponding to the minimum hop count-congestion product, i.e., the quadrant corresponding to the minimum of $5Q_1$, $8Q_2$, $9Q_3$, and $11Q_4$.

Once the quadrant is selected, the packet is routed adaptively within that quadrant using 3 VCs in the same way as is done in GOAL and GAL.

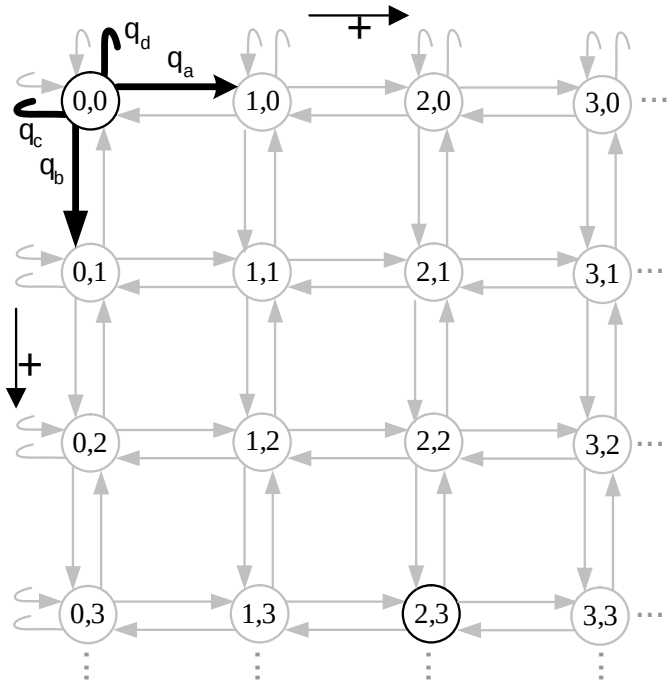


Figure 5.19: Quadrant selection from source $(0,0)$ to destination $(2,3)$ in CQR

5.6 CQR v.s. GAL: Steady-state performance

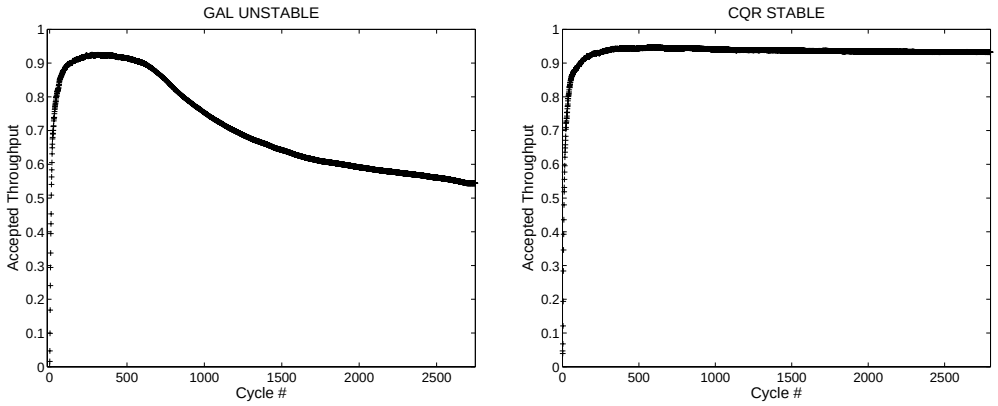
5.6.1 Summary of previous performance metrics

Table 5.1 shows that CQR is able to exactly match the performance (throughput, Θ, and latency, T) of GAL across all the figures of merit described in the first part of this chapter. Θ_{UR} represents throughput on a benign traffic, UR, while Θ_{TOR} is throughput on the adversarial TOR traffic. Θ_{avg} is the average throughput for a sampling of 1000 random permutations, while Θ_{HS} is the throughput on hot-spot traffic. Finally, T_{low} is the average latency on UR traffic at 0.2 load.

CQR, unlike GAL (with fixed threshold), is stable post-saturation as it makes its routing decision using the *channel* queues instead of injection queues. Since congestion due to post-saturation injection load remains in the source queues, CQR does not mix up the congestion due to load imbalance and that due to injection load in excess of saturation. Thus, the accepted throughput remains flat even after the offered load is in excess of saturation as

Table 5.1: Table summarizing the performance of CQR and GAL. The throughput is normalized to network capacity and latency is presented in cycles.

	GAL	CQR
$\Theta_{\mathbf{UR}}$	1.0	1.0
$\Theta_{\mathbf{TOR}}$	0.53	0.53
$\Theta_{\mathbf{avg}}$	0.73	0.73
$\Theta_{\mathbf{HS}}$	0.49	0.49
$\mathbf{T_{low}}$	4.45	4.45



(a) (b)

Figure 5.20: Performance of GAL and CQR on UR traffic at 1.1 offered load. (a) Throughput decreases over time for GAL (with fixed threshold) as it is unstable post saturation (b) CQR is stable post saturation.

shown in Figure 5.20.

5.6.2 Latency at Intermediate Loads

The biggest improvement of CQR over GAL is that at intermediate loads, it delivers packets with much lower latency than GAL for traffic patterns which require the algorithms to route non-minimally for optimal throughput. This effect is more pronounced for injection loads closer to saturation when the non-minimal paths have to be used to give higher system throughput. Section 5.4 shows an example of CQR's improvement over GAL giving up to $20\times$ lower average delay than GAL on TOR in the injection load range of $0.30 - 0.53$.

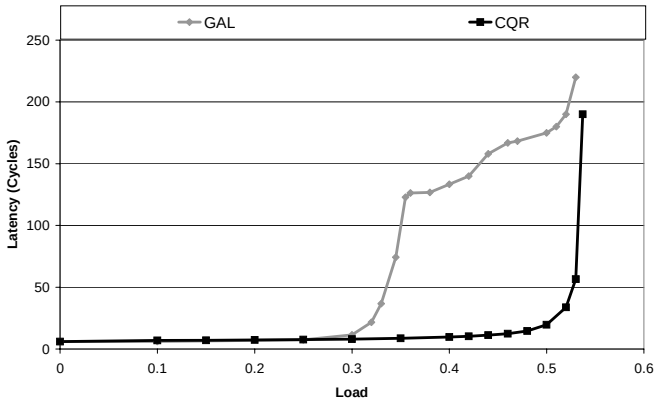


Figure 5.21: Latency profile of CQR and GAL for the 2D-tornado traffic

Figure 5.21 shows how the advantage of CQR over GAL extends to a two dimension network — an 8-ary 2-cube. The traffic pattern is the 2-dimension version of the tornado traffic on a ring, i.e., source (i, j) sends to $(i + k/2 - 1, j + k/2 - 1)$. GAL's latency-load plot rises up in *steps* which correspond to the points where a particular quadrant saturates and GAL starts routing along the next non-minimal quadrant. There are only 2 such steps for 3 non-minimal quadrants as quadrants 2 and 3 are identical in terms of the distance from the source to the destination. In contrast, CQR has a very sharp latency profile giving up to $15\times$ lower average delay than GAL in the injection load range of $0.30 - 0.53$.

5.7 CQR v.s. GAL: Dynamic traffic response

Traditionally, most adaptive routing algorithms in interconnection networks have been evaluated with static traffic patterns and only their steady-state performance has been studied. However, adaptive routing algorithms are characterized by both a steady-state and a transient response. In this section, we demonstrate how CQR's routing mechanism gives much better transient response than GAL.

5.7.1 Step Response

The first experiment we perform is to subject the network to a traffic pattern that requires the algorithm to adapt from minimal to non-minimal routing — tornado traffic. The time taken for the algorithm to adapt to congestion is both a function of the routing decision mechanism and the per-channel flow control. In order to focus on the routing decision, we keep the channel queues for each algorithm the same. We then impose the tornado traffic pattern at a load of 0.45 at cycle 0 and measure the step response of GAL and CQR.

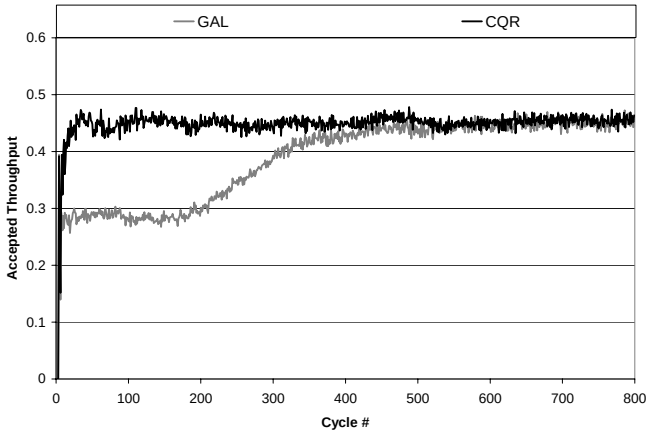


Figure 5.22: Transient response of CQR and GOAL to the tornado traffic pattern at 0.45 load started at cycle 0

Figure 5.22 shows the step response of both algorithms averaged over 100 runs. The response of GAL is much slower than CQR taking as much as $4\times$ the number of cycles to reach peak throughput. This is because, in order to adapt to routing non-minimally,

GAL must wait till the minimal injection queues fill up and surpass their threshold. However, since CQR senses load imbalance using channel queues, its step response is smooth reaching peak throughput in just 30 cycles.

5.7.2 Barrier Model

In the next transient experiment, we simulate the communication between the processors in a multi-processor environment. In this model we assume that the nodes of a multi-processor have a fixed amount of computation to perform between synchronization primitives called *barriers*. Each processor waits until every processor has completed its communication. This model assumes that all the packets are in the node's source queue before starting, instead of the usual Poisson-like injection process as used in the traffic patterns described before. The number of cycles that each processor waits is then a function of how fast the routing algorithm adapts to network congestion.

In order to stress the adaptivity of the two algorithms, we choose the destinations of each packet according to the 2D-tornado traffic pattern, i.e., each node (i, j) sends to $(i + \frac{k}{2} - 1, j + \frac{k}{2} - 1)$. The number of packets that a node has to communicate is called the *batch size*. The latency measured is the number of cycles taken for all the processors to completely route their batch of packets. We report this latency normalized to the batch size.

Figure 5.23 shows the response of CQR and GAL as the batch size is swept from 2 to 10^4 packets (shown on logarithmic scale). For small batch sizes, the latency is the same for both the algorithms as there is little congestion in the network. For extremely large batch sizes, the latency is again the same, and is given by the reciprocal of the throughput of the algorithm on the 2D-tornado traffic (0.53 for both CQR and GAL). The interesting region in the plot is when the batch size ranges from 5 to $2,000$ packets. This region underscores the importance of the more responsive congestion sensing mechanism in CQR which gives as much as $1.5\times$ performance improvement over GAL.

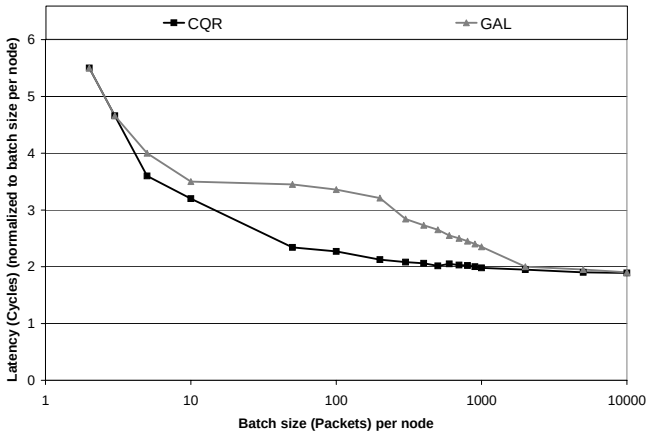


Figure 5.23: Dynamic response of CQR and GAL v.s. the batch size per node showing CQR's faster adaptivity

5.8 Summary of CQR

To summarize, CQR senses approximate global congestion and makes an adaptive global routing decision using channel queue congestion to route minimally on local traffic, while routing non-minimally to load balance difficult traffic patterns at high loads. CQR, like GAL, matches the throughput of minimal algorithms on local patterns, and load-balanced oblivious algorithms on difficult patterns — something that other algorithms, that do not make a global adaptive decision, cannot achieve.

Channel Queue Routing overcomes a number of issues associated with GAL. Most importantly, the latency for CQR at loads that require non-minimal routing is much lower than for GAL. This is because it does not need to run the minimal traffic well into saturation, with the resulting high latencies, before switching to non-minimal routing. CQR starts sending packets along non-minimal routes as soon as the delays due to minimal and non-minimal routing are matched — resulting in minimum latency. CQR also has a much faster transient response than GAL. This is because the channel queue rapidly reflects global congestion while the injection queue (used for sensing in GAL) requires that all of the queues along the minimum paths fill up before sensing congestion. CQR is also unconditionally stable while GAL requires a threshold adaptation mechanism to give stable performance. Finally, CQR is very simple to implement compared to GAL.

5.9 Summary of load-balanced routing on tori

In this and the previous two chapters, we have looked at various oblivious and adaptive load-balanced routing algorithms on torus networks and compared their performance on several metrics. Table 5.2 summarizes the performance of the different algorithms on these metrics on an 8-ary 2-cube. All numbers for throughput, Θ, are presented in fraction of network capacity, while numbers for latency, T, are presented in cycles. The numbers for Θ_{ben} are throughput values for the benign UR traffic. Θ_{wc} represent the worst throughput for each algorithm. Θ_{avg} is the average throughput for a random sampling of traffic permutations. Θ_{hs} is the throughput on hot-spot traffic with BC traffic running in the background. Θ_{hs} numbers are not presented for ROMM, RLB, and RLBth as they do not yield 0.5 throughput on the background BC traffic, unlike the other algorithms. The latency column T_{low} represents the average packet delay for UR traffic at a low injection load of 0.2, while T_{int} represents packet delay for TOR traffic at an intermediate injection load of 0.4. Finally, each algorithm is either stable or unstable post-saturation. The worst performance in any column is given a grade "F" and the other algorithms are graded relative to it.

Table 5.2: Report Card for all routing algorithms

Algo	Θ_{ben}	Θ_{wc}	Θ_{avg}	Θ_{hs}	$\mathrm{T}_{\mathrm{low}}$	$\mathrm{T}_{\mathrm{int}}$	Stab
VAL	0.5 (F)	0.5 (A)	0.5 (C)	0.25 (F)	10.5 (F)	20.5 (B)	Yes (P)
DOR	1.0 (A)	0.25 (D)	0.31 (F)	0.31 (D)	4.5 (A)	∞ (F)	Yes (P)
ROMM	1.0 (A)	0.21 (F)	0.45 (D)	-	4.5 (A)	∞ (F)	Yes (P)
RLB	0.76 (C)	0.31 (C)	0.51 (C)	-	6.5 (C)	5.5 (A)	Yes (P)
RLBth	0.82 (B)	0.30 (C)	0.51 (C)	-	5.6 (B)	5.5 (A)	Yes (P)
CHAOS	1.0 (A)	0.35 (B)	0.53 (C)	0.47 (A)	4.5 (A)	∞ (F)	No (F)
MIN AD	1.0 (A)	0.33 (C)	0.63 (B)	0.46 (A)	4.5 (A)	∞ (F)	Yes (P)
GOAL	0.76 (C)	0.5 (A)	0.68 (A)	0.47 (A)	5.45 (B)	5.5 (A)	Yes (P)
GAL	1.0 (A)	0.5 (A)	0.73 (A)	0.49 (A)	4.45 (A)	120 (D)	Yes (P*)
CQR	**1.0 (A)**	**0.5 (A)**	**0.73 (A)**	**0.49 (A)**	**4.45 (A)**	**5.5 (A)**	**Yes (P)**

The first five rows of the table show oblivious algorithms. Minimal oblivious algorithms (DOR and ROMM) perform optimally on benign traffic, but yield very poor worst-case and

average-case throughput. Being minimal, they yield optimal latency at low loads for benign traffic. VAL completely randomizes routing to give optimal worst-case performance. However, it destroys all locality in benign traffic giving poor throughput and latency on it. RLB and RLBth strike a balance between these two extremes. They load balance hard traffic while preserving some locality in friendly traffic. In doing so, they do not get an "F" grade in any of the performance metrics. All oblivious algorithms perform poorly on hot-spot traffic and are stable post saturation. While RLB is optimal in the worst-case in 1-D, it gives sub-optimal throughput guarantees in higher dimensions.

The next three rows in the table show adaptive algorithms. We incorporate adaptivity in RLB (GOAL) to alleviate its sub-optimal worst-case throughput on higher dimensions. In doing so, GOAL, unlike the other adaptive algorithms (MIN AD and CHAOS), gets "A"s in both worst and average throughput. However, since it misroutes obliviously, it cannot achieve the optimal performance (latency and throughput) of the minimal adaptive algorithms on benign traffic. Adaptive algorithms are able to easily route around hot-spots in the network and so get "A"s in hot spot performance. Finally, we show that the CHAOS algorithm in unstable because it leads to starvation for some source-destination pairs on non-uniform traffic.

The last two rows show Globally Adaptive Load-balanced algorithms, GAL and CQR. Both CQR and GAL are non-minimal, adaptive algorithms that combine the best features of minimal algorithms on benign traffic and of load-balancing algorithms on adversarial traffic patterns, getting "A"s in almost all performance metrics. Since GAL senses congestion using injection queues, it gives very high latency at intermediate loads on traffic patterns that require non-minimal routing for optimal throughput. Moreover, it needs a complex mechanism to make it stable. CQR senses approximate global congestion using channel queues and alleviates both these problems of GAL, emerging as the only algorithm with best performance across the board.

In the next chapter, we look at extending the benefits of CQR and GAL to arbitrary regular network topologies. We will generalize the concepts of minimal and non-minimal quadrants to sets of routes in the arbitrary topology and develop methods based on channel queues to sense congestion in these path sets.

Chapter 6

Universal Globally Adaptive Load-Balancing

Building on the concepts of load-balanced routing on torus networks in the previous three chapters, we now propose a universal load-balanced algorithm for arbitrary symmetric topologies. We represent the topology by a directed graph $G(V, E)$, where V are the vertices (nodes) and E are the edges (channels) in the network. Since the graph is symmetric, the degree of each node is the same.

Minimal adaptive routing (MIN AD) routes packets from a source, s, to a destination, d, along minimal paths, adaptively choosing at each hop, between the (possibly) many minimal paths, based on queue lengths. However, we have demonstrated that strictly minimal routing suffers from sub-optimal worst-case throughput, at least in torus networks. In this chapter, we prove that this statement extends to several other symmetric topologies as well. We also proved in Chapter 3 that Valiant's routing algorithm (VAL) gives optimal worst-case throughput. However, it suffers from sub-optimal performance on benign traffic. This claim holds for any symmetric topology (Theorem 2), and is not restricted to torus networks alone.

In order to guarantee optimal worst-case and best-case performance on an arbitrary symmetric topology, we propose an algorithm that routes minimally on benign traffic, and starts to load balance like VAL when the traffic pattern is adversarial.

6.1 Motivation

To understand why both MIN AD and VAL cannot achieve optimal worst-case and best-case performance, we consider an $N = 4$ node toy network shown in Figure 6.1. The capacity of this network is given by $2B/N = 4b$ bits/s. We consider routing a benign traffic pattern and a permutation traffic pattern on this network.

For the benign pattern, we assume a friendly uniform random (UR) traffic, where every node sends traffic to every possible destination, uniformly and at random. Routing UR optimally requires sending each packet minimally — along the direct channel connecting its source to its destination. Thus, MIN AD performs optimally on UR, yielding a throughput, $\Theta = 4b$ bits/s, or 100% of capacity.

For the permutation traffic, node 0 (2) wants to send all of its traffic to node 1 (3) and vice versa. Minimally routing this permutation results in an uneven distribution of load over the channels in the network. The black arrows in Figure 6.1 illustrate that only 4 links are utilized by MIN AD leaving all the other channels idle. This yields a throughput of $\Theta = b$ bits/s, or $1/4$ of capacity.

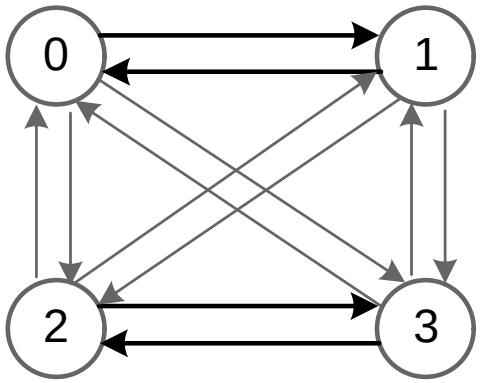


Figure 6.1: A 4 node toy network. We consider the permutation: $0 \leftrightarrow 1$; $2 \leftrightarrow 3$. The black arrows show the paths along which MIN AD routes traffic.

VAL routes any permutation traffic optimally on this network. For our example, VAL routes a packet from s to d through a randomly chosen intermediate node, i (Figure 6.2). Since i can be s or d, VAL routes along a direct (1 hop) path with probability $1/4 + 1/4 = 1/2$. It also routes along two 2 hop paths with probability $1/4$ each. Thus, at an injection

load of α, all channels have a load of 0.5α, yielding a throughput, $\Theta = 2b$ bits/s (50% of capacity). The drawback of VAL is that it yields the same throughput on every traffic pattern. Thus, for the best-case UR traffic, VAL's throughput is only 50% of the optimal throughput as achieved by MIN AD.

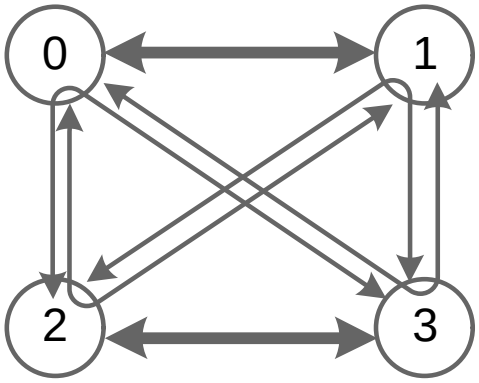


Figure 6.2: Optimally routing a permutation traffic pattern with VAL. The thick (1 hop) arrows signify a load of $\alpha/2$, while the thin (2 hop) arrows contribute a link load of $\alpha/4$.

In order to perform optimally on both these cases, we exploit the fact that for minimal routing on UR, all channels are equally loaded, while in the permutation traffic case, the queues along the direct channels fill up, leaving the other queues completely empty. This imbalance in the queue occupancies is an indication that the traffic pattern is adversarial for minimal routing, and packets should be routed non-minimally to balance load. In order to load-balance any adversarial traffic, we route packets like VAL, routing from s to d via a randomly picked intermediate node, q. The paths from $s \rightarrow q$ and $q \rightarrow d$ are minimal paths. We call this method Universal Globally Adaptive Load-balanced routing (UGAL).

At the source node of a packet, UGAL records the queue length, q_m, for the outgoing channel along the shortest path from $s \rightarrow d$. It also picks a random intermediate destination, q, and records the queue length, q_{nm}, along the shortest path from $s \rightarrow q$[1]. Denote the hop count of path $s \rightarrow d$ as H_m and that of $s \rightarrow q \rightarrow d$ as H_{nm}. In order to balance load along both minimal and non-minimal paths, UGAL routes a packet minimally if $q_m H_m \leq q_{nm} H_{nm}$, else it routes along the non-minimal path $s \rightarrow q \rightarrow d$.

Figure 6.3 shows that on UR traffic, UGAL routes minimally virtually all the time.

[1]For the case when $q = s$ or $q = d$, $q_{nm} = q_m$.

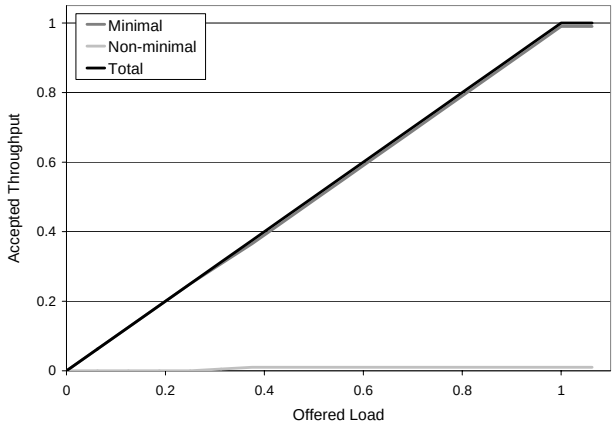


Figure 6.3: UGAL throughput on a 4 node network for UR traffic

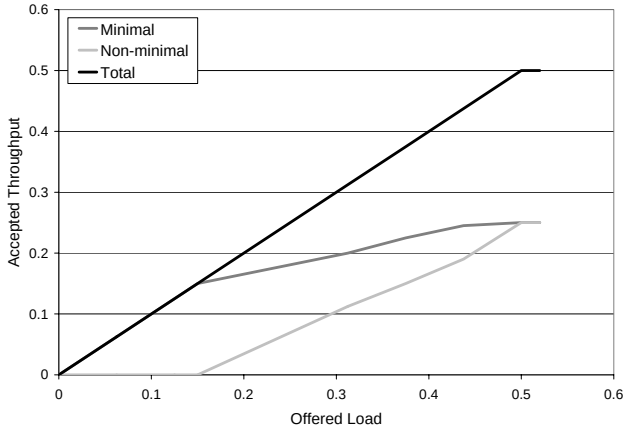


Figure 6.4: UGAL throughput on a 4 node network for permutation traffic

On permutation traffic, however, UGAL routes minimally at very low loads, but resorts to routing non-minimally at moderate to high loads (Figure 6.4). Finally, near saturation, it routes half the packets minimally and half the packets along the 2 hop non-minimal paths. As a result of this adaptive decision to misroute, UGAL yields optimal throughput on both UR and permutation traffic. Figure 6.5 shows the latency-load plots for both these traffic patterns. UGAL saturates at 50% of capacity on permutation traffic and at 100% of capacity on UR.

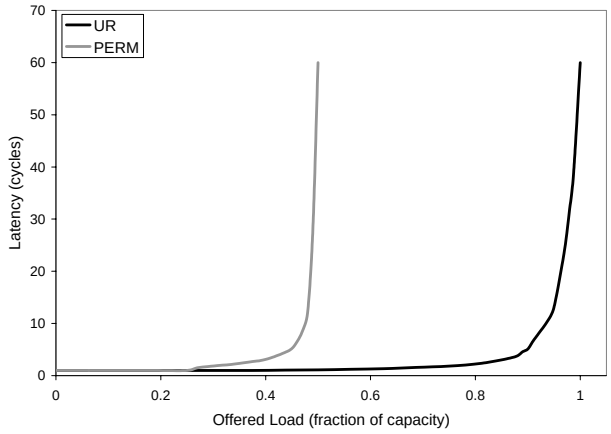


Figure 6.5: Latency-load plots for UGAL on a 4 node network for UR and permutation traffic

6.2 UGAL on arbitrary symmetric topologies

UGAL can be easily extended to an arbitrary symmetric topology. Once a packet destined for a destination, d, is received from the source queue at its source node, s, UGAL decides whether to route it minimally or like VAL, based on the congestion of the channel queues. Denote q_m as the shortest output queue length among all shortest path channels, i.e., outgoing channels along shortest paths from $s \rightarrow d$. UGAL then picks a random intermediate destination node, q. Denote q_{nm} as the shortest output queue length among all shortest path channels from $s \rightarrow q$. As in the toy example, $q_{nm} = q_m$ if $q = s$ or $q = d$. Moreover, let the hop count for the path $s \rightarrow d$ be H_m, and that for $s \rightarrow q \rightarrow d$ be H_{nm}. UGAL routes P

minimally if $q_m H_m \leq q_{nm} H_{nm}$, else it routes P from s to d via q, routing minimally and adaptively in each of the two phases.

Intuitively, it is easy to understand why UGAL gives optimal worst-case throughput. In the worst-case, all nodes start to route packets like VAL, yielding a throughput of 0.5 of capacity. Thus UGAL degenerates into VAL to ensure its throughput is never below 0.5. We now present a proof for the optimal worst-case performance of UGAL.

Theorem 6. *UGAL gives optimal worst-case throughput,* $\Theta_{wc}(\mathrm{UGAL}) = 0.5$ *of capacity.*

Proof. We prove this result by contradiction. Suppose the throughput for UGAL on some traffic pattern is $\Theta^* < 0.5$ of capacity. It follows that at an injection load of $\alpha = \Theta^*$, some channel, C, in the network is saturated, i.e., traffic crosses C at a mean rate of b bits/s (the capacity of C). Consider all the source-destination (s,d) pairs sending traffic across C. There are two cases to consider:

Case 1: Each (s,d) pair routes its traffic like Valiant (VAL).

We know from Theorem 2 that if *all* nodes route traffic like VAL, channel C (or any other channel) gets saturated iff the injected load is $\alpha \geq 0.5$ of capacity. In this case, only the sources corresponding to the (s,d) pairs mentioned above route traffic like VAL across C. No other node sends traffic across C. It follows that C cannot be saturated at an injection load less than 0.5 of capacity. This is a contradiction.

Case 2: There is is at least one source-destination pair, (s_1,d_1), that routes traffic strictly along shortest paths.

Figure 6.6 shows a schematic of the situation where (s_1,d_1) routes traffic strictly minimally through the saturated channel C. Since C is saturated, and the buffers along the path from $s_1 \rightarrow d_1$ are finite, the output queue at s_1 will eventually back up and s_1 will start routing its traffic to d_1 along non-minimal paths, which again is a contradiction.

Since both the cases lead to a contradiction, we get the desired result. □

In the next section, we apply UGAL to three different 64 node symmetric topologies — a fully connected graph, a 2-D torus, and a 4-D cube connected cycle. We evaluate UGAL on each of these topologies using a cycle accurate simulator written in C. The total buffer resources are held constant across all topologies, i.e., the product of the number of VCs and the VC channel buffer depth is kept constant.

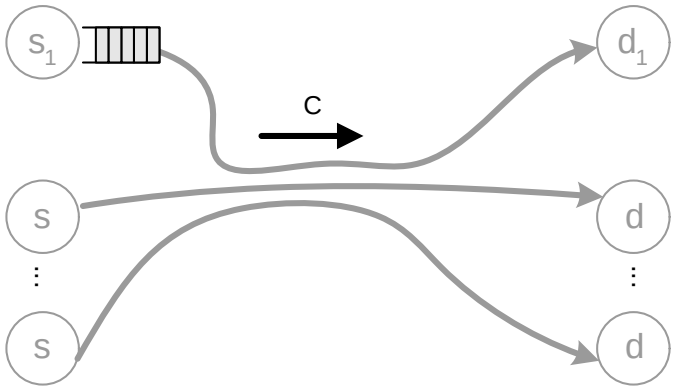


Figure 6.6: The congested channel, C, at saturation

6.3 Fully connected graph

A fully connected graph of N nodes, K_N, has a channel connecting every source-destination pair. The bisection bandwidth of K_N is $B = N^2b/2$ bits/s. Therefore, the capacity of a $N = 64$ node fully connected graph (K_{64}) is given by $2B/N = Nb = 64b$ bits/s.

6.3.1 Deadlock avoidance

Implementing UGAL on K_N requires 2 Virtual Channels (VCs) per physical channel (PC) to avoid deadlock. We use a deadlock avoidance scheme similar to the *structured buffer pool* scheme of [36, 48]. A packet that has traversed h hops may be buffered in any VC j, such that $j \leq h$. Since a packet can take at most 2 hops on K_N, 2 VCs ensure that there is no cyclic dependency in the channel dependency graph (CDG), thereby avoiding deadlock.

6.3.2 Throughput on benign and hard traffic

The uniform random (UR) traffic is the best-case traffic pattern for this topology. An adversarial traffic pattern for K_N, is any permutation (PERM) traffic, i.e., every source node sends traffic to exactly one distinct destination node (except to itself). It is easy to see that due to the fully connected nature of the graph, its performance on any permutation traffic is identical.

We have already demonstrated in Section 6.1 that MIN AD performs optimally on UR.

On K_{64}, MIN AD yields a throughput of $\Theta = 64b$ bits/s, or 100% of capacity. However, the performance of MIN AD degrades severely on PERM traffic (Appendix A.2). For a general K_N, MIN AD yields a throughput of $\Theta_{wc} = 1/N$ of capacity (0.78% of optimal for $N = 64$), for PERM traffic.

VAL routes PERM traffic optimally on K_{64}, yielding a throughput, $\Theta = 32b$ bits/s (50% of capacity). However, it yields the same throughput on every traffic pattern. Thus, for the best-case UR traffic, VAL's throughput is only 50% of the optimal throughput as achieved by MIN AD.

Figure 6.7 shows that UGAL achieves optimal throughput on both these traffic patterns. Since UR naturally balances load over all links if routed minimally, the congestion of all the output queues for any node is the same. Thus, no packet is routed like VAL for this traffic pattern yielding optimal throughput of $\Theta = 100\%$ of capacity. For PERM traffic, the queues along the direct paths start to back up while those along non-minimal paths remain empty. Sensing, this unbalanced congestion, UGAL starts to route packets like VAL, once again yielding an optimal throughput of $\Theta = 50\%$ of capacity.

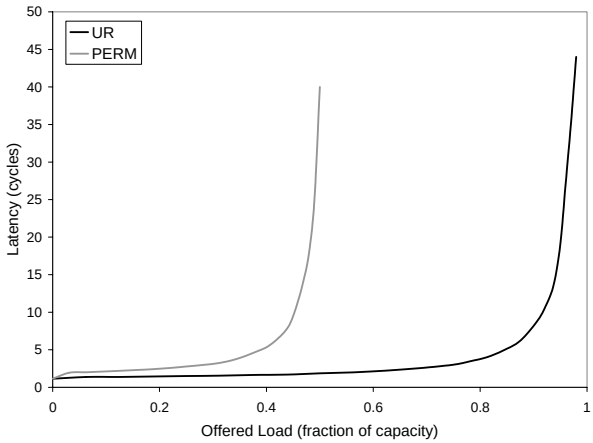


Figure 6.7: Latency-load plots for UGAL on UR and PERM traffic on K_{64}. UGAL saturates at 1 for UR and at 0.5 for PERM.

Figure 6.8 summarizes the performance of MIN AD, VAL, and UGAL on the UR and PERM traffic patterns. UGAL is the only algorithm that performs optimally on both the best-case and the worst-case traffic.

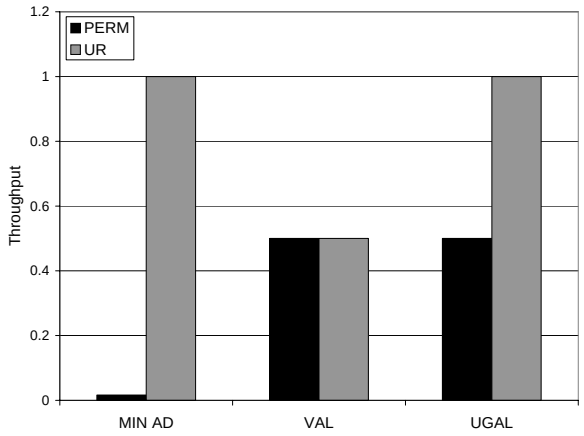


Figure 6.8: Throughput of MIN AD, VAL, and UGAL on UR and PERM traffic patterns on K_{64}

6.4 k-ary n-cube (torus)

For our second topology, we choose an 8-ary 2-cube, or an 8×8 torus network. We have extensively studied routing algorithms specifically designed for such networks in Chapters 3, 4, and 5. In this section, we apply the generic UGAL algorithm to a 64 node torus, and compare its performance with the $*$-channels algorithm [16] (MIN AD) and VAL. As we did in the previous three chapters, we compare the throughput of the different algorithms on benign traffic, adversarial traffic, and on a sampling of 1000 random traffic permutations.

6.4.1 Deadlock avoidance

Implementing UGAL on the torus requires three VCs per PC to avoid deadlock. We use the same scheme used in the $*$-channels algorithm, which we also adapted for deadlock avoidance in CQR, GAL, and GOAL.

6.4.2 Throughput on benign and hard traffic

We compare the throughput of the three algorithms on the benign and adversarial traffic patterns described in Table 2.2. The two benign traffic patterns are shown in Figure 6.9,

while the four adversarial patterns are shown in Figure 6.10 with an expanded vertical scale. The figures show that UGAL is the only algorithm that gives best performance on both the benign and the adversarial traffic. On benign patterns, UGAL routes minimally and matches the performance of MIN AD. On adversarial traffic, UGAL load balances the traffic and matches or exceeds the throughput of VAL on them. MIN AD's performance suffers on adversarial traffic as it cannot balance load over all the channels of the network. The worst-case throughput reported for MIN AD is a bound on its exact worst-case derived in Appendix A.2. In the worst-case, MIN AD degrades to a throughput of 66.67% of optimal, while UGAL's worst-case throughput (like VAL) is optimal. At the same time UGAL's throughput (like MIN AD) is optimal in the best-case, while VAL yields a poor best-case throughput of 12.5% of optimal.

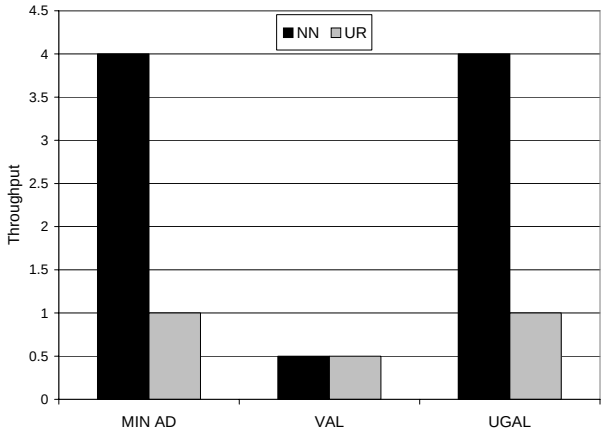


Figure 6.9: Comparison of saturation throughput of three algorithms on an 8×8 torus for two benign traffic patterns

6.4.3 Throughput on random permutations

In order to evaluate the performance of the algorithms for the *average* traffic permutation, we measured the performance of each algorithm on $1,000$ random permutations, as in the previous two chapters.

The histograms of saturation throughput for MIN AD and UGAL are shown in Figure 6.11. No histogram is shown for VAL because its throughput is always 0.5 for all

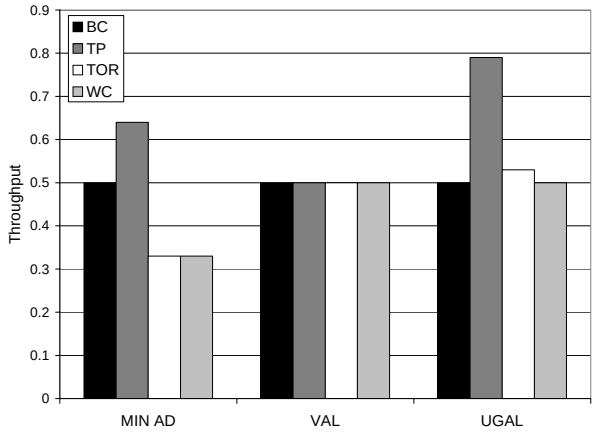
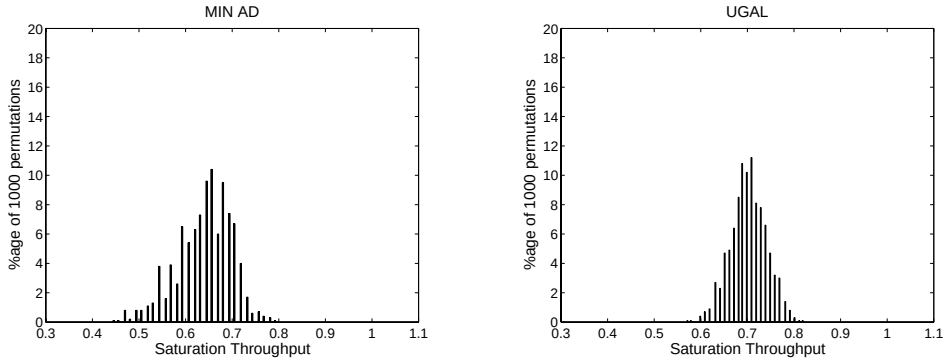


Figure 6.10: Comparison of saturation throughput of three algorithms on an 8×8 torus for four adversarial traffic patterns



(a) (b)

Figure 6.11: Histograms for the saturation throughput for 10^3 random permutations on an 8×8 torus. (a) MIN AD, (b) UGAL.

permutations. The highest, average and worst throughput in this experiment for each of the algorithms are presented in Figure 6.12.

The figures show that over the $1,000$ permutations, UGAL's worst-case throughput for this sampling exceeds that of VAL. The minimal algorithm, MIN AD, has a sub-optimal worst-case throughput. Since UGAL routes the benign permutations minimally and load balances the hard permutations, its average throughput is the highest of the three algorithms (40% more than VAL and 12% more than MIN AD).

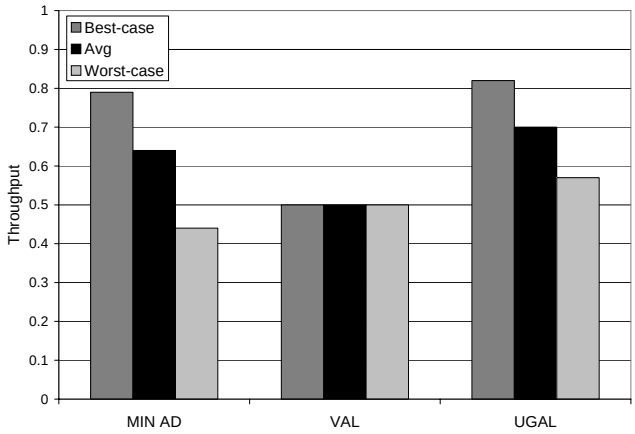


Figure 6.12: Throughput of MIN AD, VAL, and UGAL in the best, average and worst-case of 10^3 traffic permutations on an 8×8 torus

6.5 Cube Connected Cycles

Finally, we test UGAL on the Cube-Connected Cycles (CCC) topology. The CCC topology was proposed by Preparata & Vuillemin in [35] as a topology for a general-purpose parallel system. An n-dimensional CCC, CCC(n), is simply a hypercube of n dimensions, with each node replaced by a ring of n nodes. Thus, for a CCC(n), there are a total of $N = n2^n$ nodes, each with a degree of 3. Figure 6.13 illustrates a schematic of a CCC(3). For clarity, we have combined two unidirectional links into a single bidirectional link.

Minimal routing on the CCC is not as simple as in the previous two topologies. Meliksetian and Chen [28] proposed an algorithm that computes the shortest path from a source

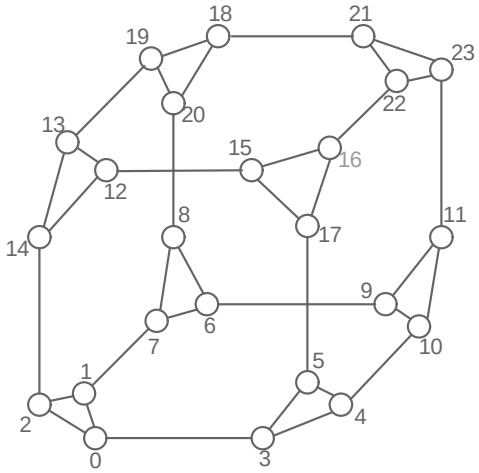


Figure 6.13: A Cube connected cycle, CCC(3)

to a destination in a CCC. Since the shortest path between a source to a destination need not be unique, our implementation of MIN AD on CCC, routes a packet along one of the shortest paths, adaptively deciding between the different paths at each hop. Consider the example of routing from node 14 to node 11 in Figure 6.14. There are 4 possible shortest paths, each 6 hops long. At each hop, a packet is routed along a channel that takes it 1 hop closer to the destination. If there are two or more such outgoing channels, MIN AD routes the packet along the channel with the shorter queue length. In our example, the packet in node 14 could be routed along channels $14 \rightarrow 2$, $14 \rightarrow 12$ or $14 \rightarrow 13$. Suppose the length of the output queue for channel $14 \rightarrow 2$ was the shortest. The packet is therefore routed to node 2. At node 2, the choices of channels are $2 \rightarrow 0$ or $2 \rightarrow 1$. At this hop, the length of the queue for channel $2 \rightarrow 0$ is shorter. Hence, the packet is routed to node 0. Thereafter, the packet follows a direct path along nodes 3, 4, and 10, before reaching its destination node 11.

6.5.1 Deadlock avoidance

In our experiments, we have simulated a 64 node CCC, CCC(4). As we did for the K_N topology, we use the structured buffer pool technique to avoid deadlock in the CCC(4). Since the maximum hop count for MIN AD, VAL, and UGAL on CCC(4) is 8, 16, and 16,

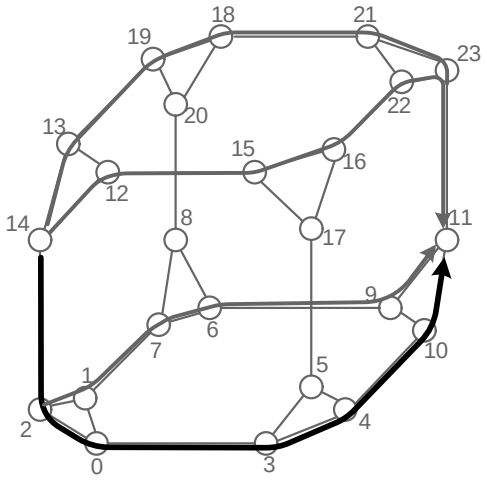


Figure 6.14: Example of MIN AD routing from source 14 to destination 11 on CCC(3)

respectively, we use 8 VCs per PC for MIN AD and 16 VCs for both VAL and UGAL.

6.5.2 Throughput on benign and hard traffic

We now compare the throughput of MIN AD, VAL, and UGAL on the traffic patterns shown in Table 6.1. The first two traffic patterns, NN and UR, are benign and the next three are adversarial traffic permutations. The last traffic permutation, WC, is the worst-case traffic permutation for each algorithm among all possible permutations. The WC permutation for UGAL is the BC permutation, while any permutation is the worst-case for VAL. For MIN AD, the worst-case pattern (derived in Appendix A.2) sets an upper bound on the throughput that MIN AD can yield in the worst-case.

As in the case of the previous two topologies, UGAL and MIN AD perform optimally on the benign patterns, yielding a throughput of 3 and 1 on NN and UR, respectively. VAL's throughput on NN (16.67% of optimal) and on UR (50% of optimal) is significantly lower than MIN AD and UGAL.

For adversarial traffic, VAL yields a throughput of 50% of capacity for all patterns. However, MIN AD cannot guarantee this worst-case throughput. As shown in Appendix A.2, MIN AD's throughput in the worst-case can be as low as 0.2 of capacity. UGAL balances

Table 6.1: Traffic patterns for evaluation of routing algorithms on CCC(4). A node's address is represented in 6 bits for CCC(4).

Name	Description
NN	Nearest Neighbor: each node sends to one of its three neighbors with probability $1/3$ each.
UR	Uniform Random: each node sends to a randomly selected node.
BC	Bit Complement: Node (b_5,b_4,b_3,b_2,b_1,b_0) sends to (b'_5,b'_4,b'_3,b'_2,b'_1,b'_0).
BR	Bit Reversal: Node (b_5,b_4,b_3,b_2,b_1,b_0) sends to (b_0,b_1,b_2,b_3,b_4,b_5).
TP	Transpose: Node (b_5,b_4,b_3,b_2,b_1,b_0) sends to (b_2,b_1,b_0,b_5,b_4,b_3).
WC	Worst-case: The permutation that gives the lowest throughput by achieving the maximum load on a single link (See Appendix A.2).

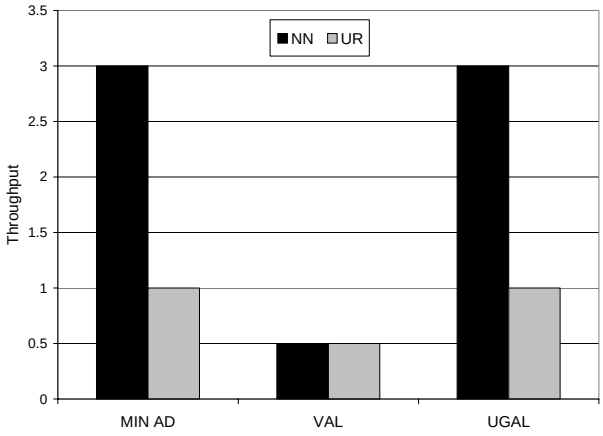


Figure 6.15: Comparison of saturation throughput of three algorithms on CCC(4) for two benign traffic patterns

each adversarial pattern most efficiently, yielding the highest throughput in each of the traffic permutations. At the same time, UGAL matches the optimal worst-case throughput of VAL.

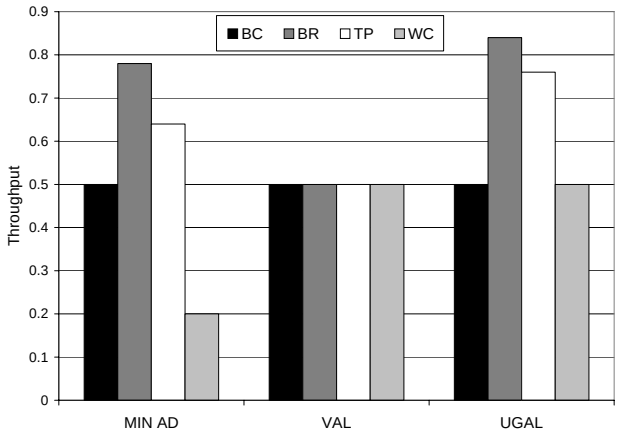
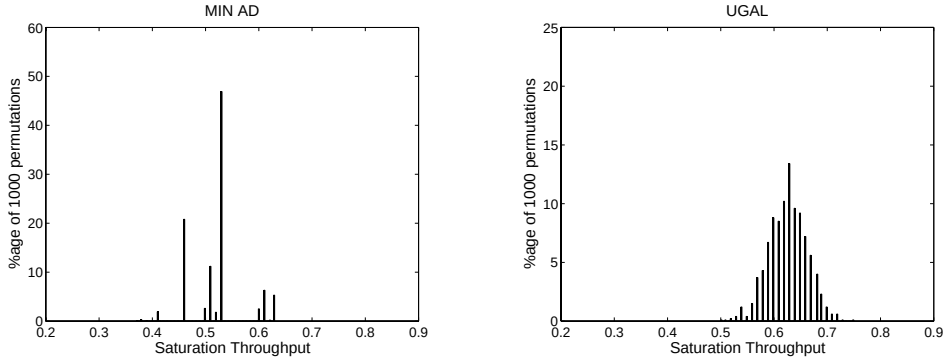


Figure 6.16: Comparison of saturation throughput of three algorithms on CCC(4) for four adversarial traffic patterns

6.5.3 Throughput on random permutations

On the *average* permutation, UGAL again outperforms both MIN AD and VAL. The histograms for throughput on 1000 random traffic permutations are presented for MIN AD and UGAL in Figure 6.17. Since VAL's throughput is 0.5 for all permutations, its histogram is not presented. The best-case, average and worst-case throughput for each of the algorithms on this sampling is shown in Figure 6.18.

The figures illustrate how UGAL load-balances adversarial traffic patterns (resulting in highest worst-case throughput), while at the same time not sacrificing the locality inherent in benign traffic permutations, yielding 21% and 26% higher average throughput than MIN AD and VAL, respectively.



(a) (b)

Figure 6.17: Histograms for the saturation throughput for 10^3 random permutations on CCC(4). (a) MIN AD, (b) UGAL.

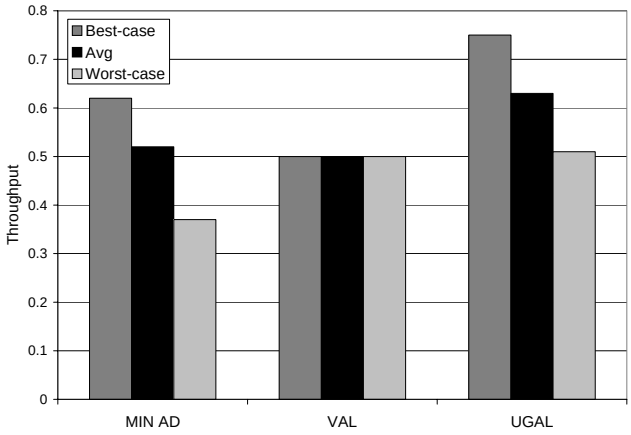


Figure 6.18: Throughput of MIN AD, VAL, and UGAL in the best, average and worst-case of 10^3 traffic permutations on CCC(4)

6.6 Summary

In this Chapter, we have presented Universal Globally Adaptive Load-balanced routing (UGAL). UGAL is an adaptive routing algorithm that guarantees optimal worst-case performance, without sacrificing any locality on benign (best-case) traffic. UGAL achieves this property by adaptively deciding when to route packets minimally or non-minimally based on the congestion of the channel queues. UGAL is *universal* because it can be applied to an arbitrary symmetric topology. We applied UGAL to three symmetric topologies — fully connected graphs, torus networks, and cube connected cycles — and demonstrated its effectiveness on each of them through extensive simulations.

Chapter 7

Delay and buffer bounds

Up to this point, we have extensively studied the performance of interconnection networks on metrics such as throughput, average latency, and stability. In this chapter, we examine other important metrics such as statistical guarantees on packet delay, buffer occupancy, and packet reordering. We apply recent results in queueing theory to propose a methodology for bounding the buffer depth and packet delay in high radix, load-balanced interconnection networks [39]. We present a methodology for calculating such bounds for a practical high radix network and through extensive simulations show its effectiveness for both bursty and non-bursty injection traffic.

7.1 Background

High radix Interconnection networks are widely used in supercomputer networks (Merrimac Streaming Supercomputer [10]) and for I/O interconnect (Infiniband Switch fabric [34]). Most research for such interconnection networks focuses on analyzing the throughput and average packet latency of the system. However, little work has been done towards bounding the occupancy of the buffers in the network or the delay incurred by a packet.

The buffer occupancy and the delay distributions play a key role in the design and performance of the network. Bounding the number of packets in a buffer in the network is valuable for network administration and buffer resource allocation. A statistical bound

on the packet delay is essential for guaranteeing Quality of Service for delivery of packets. Moreover, the delay distribution is directly related to packet reordering through the network.

Queueing theory [23] provides a huge body of useful results which apply to *product-form* networks. Unfortunately, these results rely on unrealistic assumptions (the most unrealistic being independent exponentially distributed service times at each node as opposed to deterministic service in a real system), and therefore, people are reluctant to make use of them. The analysis of a network of deterministic service queues is a known hard problem. The difficulty in analysis primarily arises from the fact that the traffic processes do not retain their statistical properties as they traverse such a network of queues.

Given a myriad of sophisticated techniques developed for analyzing a single deterministic service queue, there has been some recent work that attempts to decompose the network based on large deviations techniques [51, 52]. Most of these results are applicable in convergence regimes, such as in the case when there are several flows passing through a queue, called the *many sources asymptotic* regime. Using the many-sources-aysmptotic, Wischik [51, 52] shows that the distribution of a traffic flow is preserved by passage through a queue with deterministic service, in the limit where the number of independent input flows to that queue increases and the service rate and buffer size increase in proportion. More recently, Eun & Shroff [15, 14] use similar convergence results to significantly simplify the analysis of such a network. In particular, they show that, if internal nodes in a network are capable of serving many flows, we can remove these nodes from consideration and the queueing behavior of other network nodes remains largely the same.

In this chapter, we use the aforementioned convergence results to bound the queue depth and packet delay in high radix interconnection networks. Our simulations show that such convergence results start to kick in when the radix (degree) of the nodes is as small as $16 - 32$. We also use our bounds to study the reordering of packets through the network and to propose a routing mechanism with almost no flow control overhead.

7.2 Many sources queueing regime

As discussed in Section 7.1, a recent result by Eun & Shroff [15, 14] shows how network analysis can be simplified when the queues in the network carry several flows (called the many sources regime). The authors prove that when an upstream queue serves a large number of traffic flows, the queue length of a downstream queue converges to the queue length of a simplified single queueing system obtained by just removing the upstream queue.

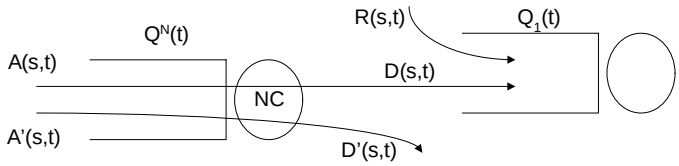


Figure 7.1: An upstream queue feeding some flows into a downstream queue

Consider the set up of two FIFO queues in Figure 7.1. Let there be N flows going into the upstream queue. The subset of these flows that go on into the downstream queue have a combined arrival process given by $A(s,t)$ which is the total number of packets arriving in the time interval (s,t). The remaining set of flows have arrival process $A'(s,t)$. Let the service capacity of the upstream queue be NC, i.e. the service per flow is C packets per time step. The departing flows from the first queue going into the downstream queue have a departure process given by $D(s,t)$. The downstream queue can also receive more cross traffic given by $R(s,t)$. Let the queue depth of the downstream queue at time t be $Q_1(t)$ while that for the upstream queue be $Q^N(t)$. In order to find the Prob$(Q_1 > x)$, we can simplify the above scenario into just one queue.

Figure 7.2 shows a simplified scenario of Figure 7.1. In this figure, the effect of the upstream queue on $A(s,t)$ is ignored. Let the queue depth of the downstream queue for this scenario be $Q_2(t)$. The authors of [14] prove that as $N \to \infty$, Prob$(Q_1 > x)$ converges to Prob$(Q_2 > x)$ and that the speed of this convergence is exponentially fast. Hence, with a modest number of multiplexed sources, the convergence results start to hold.

At a first glance, it may seem that the flows traversing the upstream queue should become "rate shaped", making the departure process over a time interval t, $D(0,t)$, smoother than the corresponding arrival process $A(0,t)$. However, on closer analysis, this need not

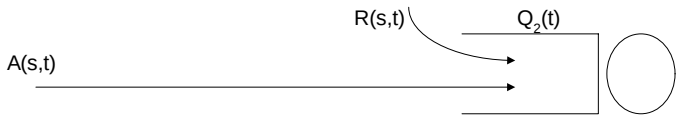


Figure 7.2: Simplified scenario of the set up of two queues

be the case. Consider an individual flow, i, traversing the upstream queue and going to the downstream queue. Its arrival process is $A_i(0,t)$ and departure is $D_i(0,t)$. Now the arrival and departure processes are related as: $D_i(0,t) = A_i(0,t) + Q_i^N(0) - Q_i^N(t)$, where $Q_i^N(0)$ and $Q_i^N(t)$ are the backlog of flow i in the upstream queue at times 0 and t, respectively. Due to fluctuations in the queue depth and the interaction among the N flows in the upstream queue, $Q_i^N(0)$ can be larger than $Q_i^N(t)$, making the departure process for that flow larger than the arrival process.

7.3 Application to high-radix fat trees

In the rest of this chapter, we shall apply the many sources regime results to analyzing buffer depth in a popular high radix topology — Clos [7] or fat tree networks [26]. The high radix switch queues are an appropriate application for the many sources asymptotic results. As the radix (and the number of sources) increases, the statistical properties of the flows get preserved as they traverse the network. Our simulations show that the convergence results hold for a radix as small as $16 - 32$.

Throughout the discussion, we assume that the traffic pattern is admissible, i.e., no destination is oversubscribed. In order to prevent inadmissible traffic, an end-to-end reservation scheme for admission control can be used which does not admit a flow if the destination is oversubscribed.

In our experimental set up, we have simulated a specific kind of fat tree — a k-ary n-tree network [33]. A k-ary n-tree network has n levels of internal switches and a total of k^n leaf nodes that can communicate with each other using these switches. There are nk^{n-1} internal switches which have $2k$ incoming ports and $2k$ outgoing ports. The internal switches have buffers inside where packets are stored and serviced to their appropriate output port. Figure 7.3 shows a diagram for a 2-ary 4-tree.

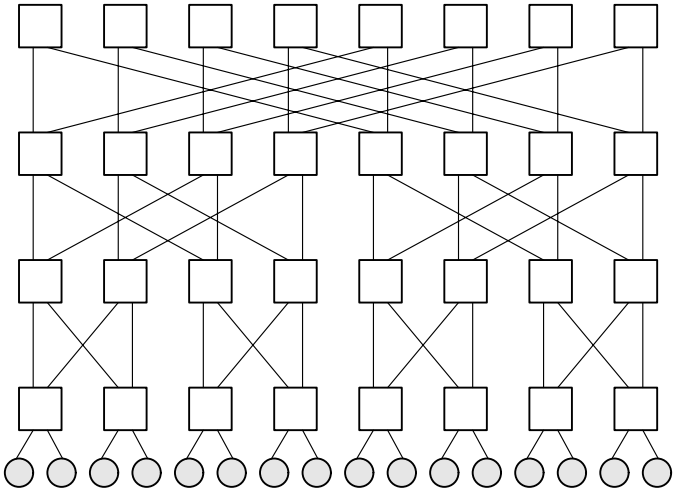


Figure 7.3: A 2-ary 4-tree

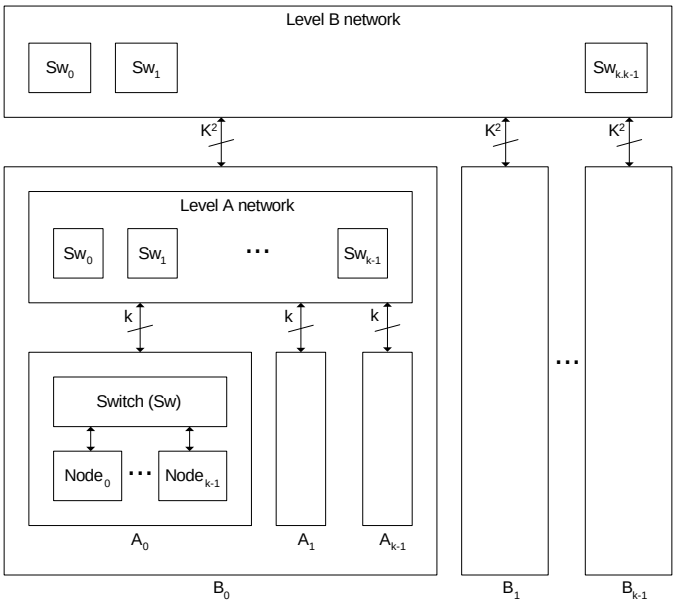


Figure 7.4: The k-ary 3-tree used in our smulations

In our simulations, we concentrate on high radix trees with $k \geq 16$. Figure 7.4 shows a hierarchical schematic for a k-ary 3-tree. There are two levels of hierarchy denoted by levels A and B and three levels of switches since $n = 3$. Depending on k, each entity of level A may be thought of as a board (or a backplane comprising several boards), while each entity of level B may be thought of as a backplane (or cabinet). The wiring between levels is abstracted out for simplicity.

Load balancing on such a fat tree is easily accomplished using Random Root Routing (RRR) in which each packet is routed to a randomly chosen root switch and then down to the desired destination, traversing a total of $2n-1$ internal switches (hops) for every packet. A more sophisticated approach — called Random Common Ancestor Routing (RCAR) — is to route up to a (randomly chosen) common ancestor of the source and destination leaf node and then down to the destination node. In our analysis, we focus on RRR as it enables us to treat all traffic patterns as two phases of uniformly random traffic, thus making the analysis more tractable. The analysis for RRR is also a conservative analysis for RCAR as the latter has strictly fewer packets using the resources in the upper levels of the network.

7.4 Results

We first study the impact of the radix on the buffer occupancy distribution in the queues at each hop of the fat tree network. Our approach is to first increase the radix k in a k-ary 3-tree network while keeping the per channel bandwidth constant. We then plot the Complementary Cumulative Distribution Functions (CCDFs) of the queue occupancy at each of the 5 hops of the network. We perform this experiment for non-bursty Bernoulli and bursty injection traffic. For both these injection processes, the many sources convergence results start to manifest themselves at reasonably small values of k. Using these values of k, we can analytically calculate the exact CCDF from queueing theory for each of the 5 queues, thus giving us buffer depth bounds. We then use the per-hop buffer depth bound to approximate the end-to-end delay of a packet through the network by convolving the per-hop distributions obtained. This approximation gives very accurate results especially at high injection loads.

7.4.1 Increasing the radix k

In the very first experiment, we inject packets at each source according to a non-bursty Bernoulli iid process. Each source injects a packet with a probability p at every time step. We increase the radix k of each switch and measure the queue depth at each of the 5 hops of a k-ary 3-tree network. Figure 7.5 shows that the CCDFs of the queues are quite divergent for a low radix ($k = 2$) but tend to converge[1] to almost identical plots as k is increased to 16. This is because, for a high enough radix, the statistical properties of the flows are preserved as they traverse the network. Hence, for the non-bursty injection process, a radix of 16 is a reasonable working parameter to use the convergence results.

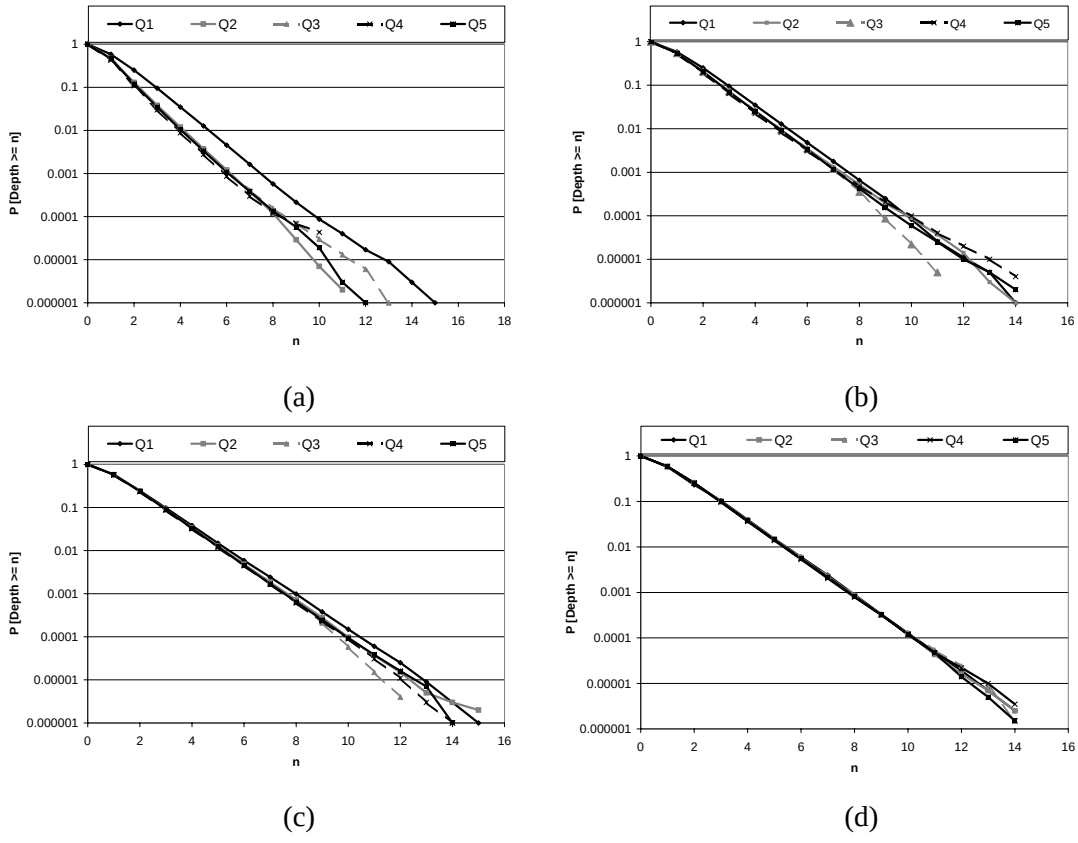


Figure 7.5: Queue depth at each hop of a k-ary 3-tree for (a) k=2 (b) k=4 (c) k=8 and (d) k=16

[1]We assume "convergence" if for a probability of 10^{-3}, the queue occupancy for each hop is within 3% of the analytically derived value (explained in the next subsection) with 99% confidence.

7.4.2 Analytically obtaining the CCDF

We now describe our analytical approach for obtaining the CCDFs of the queue depths at each hop. For the 16-ary 3-tree case with non-bursty injection, it suffices to obtain the CCDF for the first hop as the other hops behave almost identically.

Let A be the random variable that represents the total traffic at each time step to the queue at the first hop.

$$A = \sum_{i=1}^{k} X_i \tag{7.1}$$

where $X_1, X_2, \ldots X_k$ are IID Bernoulli random variables corresponding to the k sources such that if p_i is the probability that the source i will send a packet along this queue, then

$$X_i = \begin{cases} 1 & \text{with probability } \mathrm{p_i} \\ 0 & \text{with probability } \mathrm{q_i} = 1 - \mathrm{p_i} \end{cases} \tag{7.2}$$

From Equations 7.1,7.2 we get

$$E[A] = \sum_{i=0}^{k} p_i$$

Let the service capacity of the queue be $C = 1$ packet per time step. If the queue is stable i.e., $E[A] < C$, we can find the Probability Generating Function (PGF)[2] of the queue size $Q(z)$ using the formula derived in [24]:

$$Q(z) = P(Q = 0)\frac{(z-1)A(z)}{(z - A(z))} \tag{7.3}$$

In our case

$$A(z) = \prod_{i=0}^{k}(q_i + p_i z) \text{ and } P(Q = 0) = 1 - E[A] \tag{7.4}$$

The derivation for $P(Q = 0)$ is shown in Appendix D. Moreover, $p_i = p$ for all i since

[2]The PGF, G, of a random variable, X, is given by $G(z) = E(z^X) = \sum_{i=0}^{\infty} f(i)z^i$, where f is the probability mass function for X.

all sources inject traffic with the same probability. Using Equation 7.3, we can find out the queue size distribution as follows:

$$P(Q = n) = \frac{1}{n!} \left[\frac{d^n}{dz^n} Q(z) \right]_{z=0} \tag{7.5}$$

Equation 7.5 can then be used to derive the CCDF for the queue depth at each hop of the 16-ary 3-tree network.

Figure 7.6 shows the analytically evaluated queue depth with the error margins derived as in [14] against the measured values obtained in the previous subsection. The queues at each hop are at a utilization of 60% i.e., $E[A]/C = 0.6$. The model matches almost exactly with the simulations. While the observed values were obtained through simulations run for 2 days, the theoretical CCDF could be derived in a matter of minutes using the Maple software [6] for solving Equation 7.5. The advantage is that we could quickly derive the buffer depth required for which the overflow probability would be of the order of 10^{-15}, something that was not tractable by simulations alone.

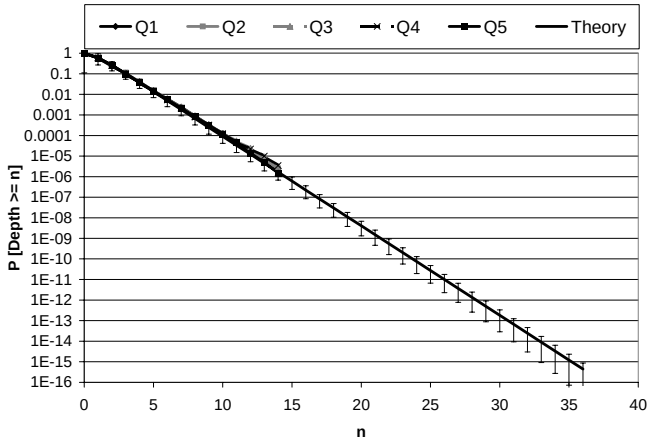


Figure 7.6: Analytically derived queue depth against the measured queue depth at each hop for a 16-ary 3-tree at 0.6 load

7.4.3 Approximating the delay CCDF

Having found the buffer depth distributions at each hop, our next aim is to approximate the end-to-end delay of packets traversing the network. The latency, T, incurred by a packet is the sum of two components, $T = H + D$, where H is the hop count and D is the queueing delay. For the 16-ary 3-tree case, $H = 5$. The random variable D is the queueing delay incurred due to buffering at each of the hops in the network. The per-hop delay in each queue is directly related to the occupancy of the queue and its service capacity. In order to find the distribution of D, we make the simplifying assumption that the per hop delays at each hop are independent of each other. This assumption has been shown to give accurate results in practice, especially as the load is increased [53]. This is because, as the buffer load increases, the output of a queue (which is the input to the downstream queue) gets less correlated with the input process.

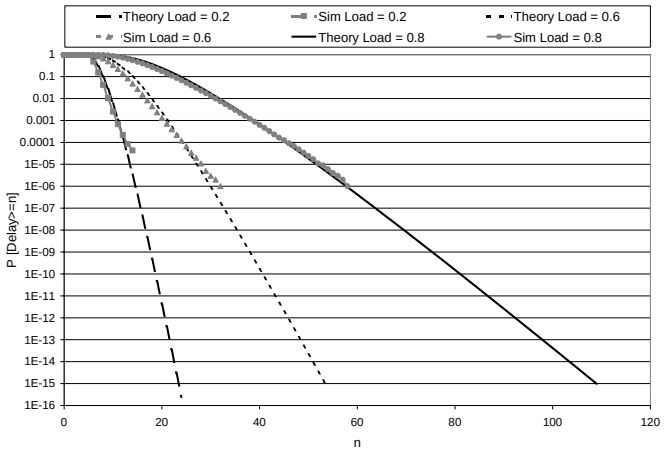


Figure 7.7: End-to-end packet delay (theoretical and measured) for a 16-ary 3-tree at different injected loads

Using this independence assumption, we can find the distribution of D by simply convolving the per-hop delay distributions calculated before. Convolving the distributions is equivalent to simply taking the product of their PGFs. For the case when the service is 1 packet per time step, D is given by

$$D(z) = \prod_{i=1}^{H} Q_i(z) \tag{7.6}$$

Finally, adding a constant H to D gives us the end-to-end delay distribution.

Figure 7.7 compares the analytical delay CCDF with the measured values for injection loads of 0.2, 0.6 and 0.8. As expected, the analytical and observed plots are a close match for the highest load of 0.8. They also are reasonably good approximations for the lower loads.

7.4.4 Bursty Flows

The radix at which convergence holds depends on the size of the network and the nature of the injection sources. In this subsection, we use bursty sources for injection into the network instead of the non-bursty Bernoulli sources that we have used thus far. The injection process is now based on a simple Markov ON/OFF process which produces packet bursts that are geometrically distributed with average burst length of 5 packets. We repeat the same set of experiments as in the Bernoulli injection case.

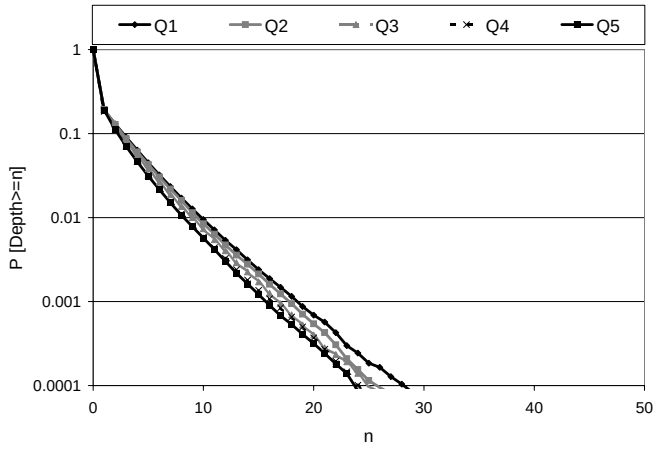


Figure 7.8: Queue depth at each hop of a 16-ary 3-tree for bursty traffic and 0.6 load

As we increase the radix k, keeping the load fixed at 0.6 and the per channel bandwidth constant, we once again observe a convergence in the CCDFs of queue lengths at different hops. For the bursty case, the CCDFs for $k = 16$ do not converge as well as in the Bernoulli

case (Figure 7.8). A higher value of $k = 32$ (Figure 7.9) shows convergence sufficient for the above analysis to give good results.

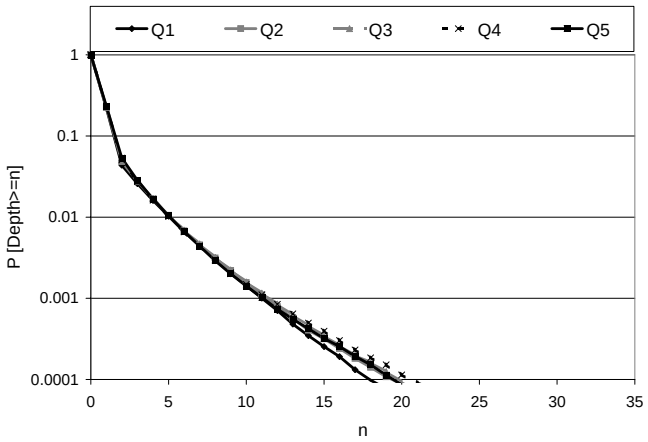


Figure 7.9: Queue depth at each hop of a 32-ary 3-tree for bursty traffic and 0.6 load

The analytical derivation of the queue depth for such a correlated, bursty injection process is mathematically involved and is beyond the scope of this thesis. The interested reader is referred to [22] for more details. In order to evaluate the end-to-end delay CCDF for the bursty case, we use the *measured* distribution for the queue delay at the first queue and convolve it five times to get the delay distribution. As seen in Figure 7.10, the CCDF obtained by the convolution is very close to the measured delay CCDF for $k = 32$.

7.5 Discussion

7.5.1 Using buffer bounds to disable flow control

An interesting fallout from deriving buffer depth bounds is that we can use them to make the flow control of the packets trivial. In practice, the packets traversing the network need to be allocated resources before they can actually use them. This process of allocation of network resources is called flow control. The most commonly used flow control mechanism for allocating buffers to packets is the credit-based flow control [12] (Chapter 13). In order not to drop packets, a packet in the upstream node must not leave for the downstream node

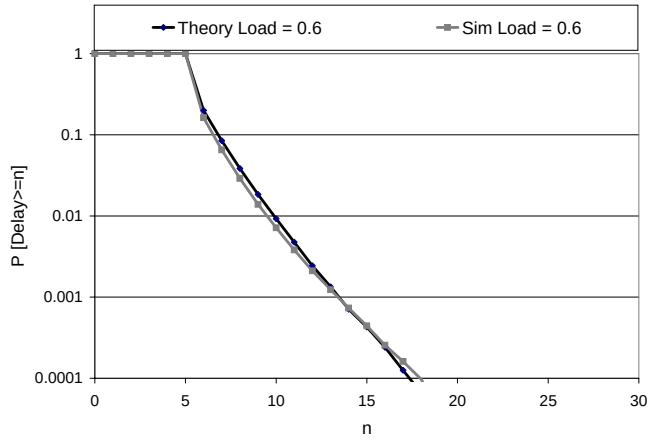


Figure 7.10: End-to-end packet delay for a 32-ary 3-tree for bursty traffic at an injected load of 0.6

until it gets a *credit* signifying there is enough buffer space available there. If we were to send a packet to the downstream node ignoring this flow control information, the packet would be dropped if there was no space in the downstream node. However, we can set our buffer depths to a reasonable size and make sure that for admissible traffic, the probability of a drop due to buffer overflow is substantially lower than the probability of a drop due to a hard error. In order to do this, we must either police the injection process or provide a modest internal speedup to the network.

Take, for instance, the non-bursty injection process on a 16-ary 3-tree. We have successfully computed the buffer depth requirement for a particular injection load of 0.6 that has a very low probability of exceeding (of the order 10^{-15}). Repeating our calculations for different loads, we can study how the buffer depth requirements grow as the load is increased, while keeping the probability of overflow constant at 10^{-15}.

Figure 7.11 shows that as the injected load reaches the saturation value of 1, the buffers start to grow without bound. However, at slightly less than this saturation value, say 0.9, a buffer size of 160 packets is required to ensure that the drop probability without flow control is less than 10^{-15}. Hence, either by policing the injection to make sure that the injection rate stays below 0.9 or by providing a small internal speedup of $1/0.9 = 1.11$, we can eliminate the flow control overhead from the routing process. A similar analysis can be carried out for other injection processes.

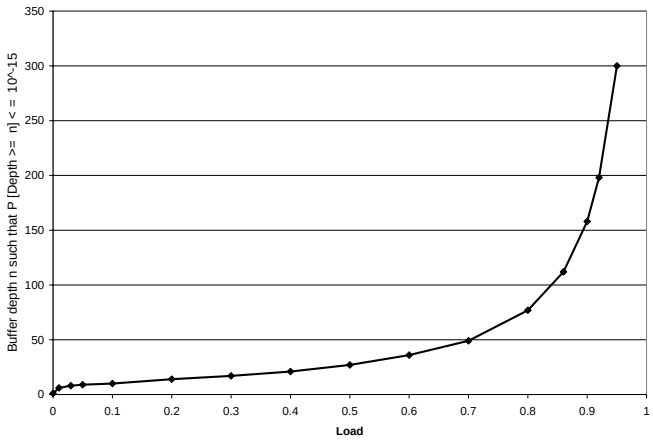


Figure 7.11: Buffer depth requirement at various injection loads

7.5.2 Bounding the reordering of packets

While load balancing algorithms such as RRR and RCAR increase the throughput of the network, they also reorder the packets traveling from a particular source, s, to a particular destination, d. This happens because packets are sent over different paths with potentially different delays from s to d. For instance, Figure 7.12 shows packets sent over 3 different paths. The black packet is the next packet expected by d but it is delayed in the congested path P_1. To deliver packets in sequence to d, the other packets (which were injected later than the black packet but reached earlier) must be stored and reordered in a Reorder Buffer (ROB) using the well known sliding window protocol [46].

At each destination, there must be a ROB corresponding to each source. Choosing the size of each ROB is critical to the throughput of the network. Typically, when the ROB becomes full, packets are dropped and an end-to-end retry protocol retransmits the dropped packets. Note that the ROB cannot block packets in the network as this may lead to a deadlock — the packet that the ROB awaits in order to drain gets blocked by the ROB. It is therefore essential to evaluate a ROB size such that the probability of it getting full is significantly low.

Unlike the case of the switch buffers discussed thus far, the ROB occupancy varies with traffic patterns. Consider a permutation traffic pattern (PERM) and a non-permutation traffic such as Uniform Random (UR). In PERM, s sends packets to a fixed destination,

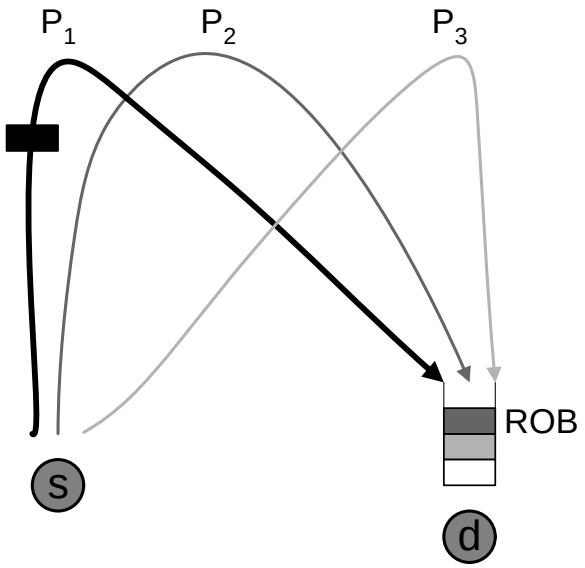


Figure 7.12: Reordering of packets sent over 3 different paths

while in UR, s sends to a randomly chosen destination. Since reordering occurs for source-destination pairs, permutation patterns use only one ROB at each destination sending more traffic to each ROB than a non-permutation traffic such as UR. Consequently, the ROB occupancy for any permutation is bigger than that for a non-permutation traffic for the same injection load. For this reason, we focus on the PERM traffic pattern for our ROB size calculations.

Let us denote the CCDF for the delay that we evaluated in Section 7.4.3 by D_i, i.e., $D_i = P[Delay \geq i]$. Let the load on the network be a fraction l of its capacity. Let Q_r be the occupancy of each reorder buffer. Then, for our canonical example of non-bursty Bernoulli arrivals, we can theoretically bound Q_r as follows:

Theorem 7. *For non-bursty Bernoulli arrivals, if* $\rho_d = P[Delay = d] = D_d - D_{d+1}$, *then*

$$P[Q_r \geq k] \leq \sum_{d=0}^{\infty} \rho_d \, e^{(e^{\eta_d}-1)S_d - \eta_d k} \tag{7.7}$$

$$\textit{where } \eta_d = \ln(k/S_d) \textit{ and } S_d = l \sum_{i=0}^{d-1} (1 - D_i)$$

Proof. In order to find $P[Q_r \geq k]$, we must find the probability of the event that while the ROB is waiting for the next in-sequence packet (call it C), k or more packets injected after C arrive at the ROB. Without loss of generality, let C be injected by the source at time 0. Once C arrives at the ROB after some delay, d, the buffer starts to drain. We need to count the number of packets that are generated and reach the ROB through paths different from that of C in the interval $(0, d]$. The probability that this number exceeds k for all d is equivalent to $P[Q_r \geq k]$.

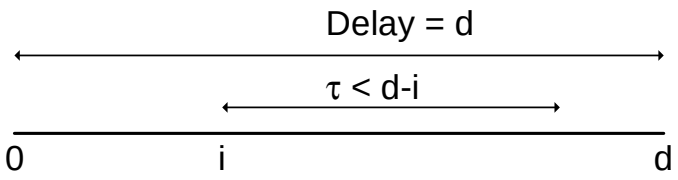


Figure 7.13: Time-line for deriving ROB occupancy

Figure 7.13 shows the time-line for packet C. For each discrete time step, i, in the interval $(0, d]$, define an indicator variable, T_i, such that $T_i = 1$ if a packet is generated at time i and also reaches the ROB with a delay less than $d - i$ [3]. Since the load is l and the delay CCDF is D_i, we have

$$T_i = \begin{cases} 1 & \text{with probability } \mathrm{l(1 - D_{d-i})} \\ 0 & else \end{cases}$$

Then, if $C's$ delay is given by a random variable D_C,

$$P(Q_r \geq k) = \sum_{d=0}^{\infty} P(D_C = d) P(Q_r \geq k | D_C = d) \tag{7.8}$$

Now, $P(D_C = d) = \rho_d = D_d - D_{d+1}$ and for a fixed $D_C = d$, $Q_r = \sum_{i=1}^{d} T_i$. Thus, to find a bound on $P(Q_r \geq k)$, we need to bound the probability that the sum of the Bernoulli variables T_i exceeds k. We use a Chernoff bounding technique similar to the one used in [30].

[3] For simplicity, we ignore the probability that the packet will follow the same path as C as the number of paths to the destination is large.

For a fixed $D_C = d$, and any $\eta_d > 0$,

$$P(Q_r \geq k) = P(e^{\eta_d \sum_{i=1}^{d} T_i} \geq e^{\eta_d k}) \leq \frac{\prod_{i=1}^{d} E[e^{\eta_d T_i}]}{e^{\eta_d k}}$$

The RHS of the above inequality can be simplified to $\frac{\prod_{i=1}^{d}(1+(e^{\eta_d}-1)l(1-D_{j-i}))}{e^{\eta_d k}}$. Finally, using the fact that $1 + x < e^x$, we can simplify the inequality to

$$P(Q_r \geq k | D_C = d) \leq e^{(e^{\eta_d}-1)S_d - \eta_d k} \tag{7.9}$$

where $S_d = l \sum_{i=0}^{d-1}(1 - D_i)$.

To get the tightest bound, we must minimize the exponent in the RHS of Equation 7.9. Solving, we get $\eta_d = \ln(k/S_d)$. Substituting in Equation 7.8, we get the desired result. □

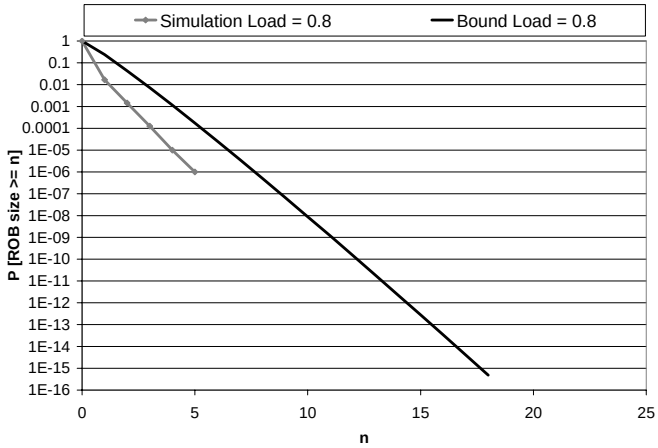


Figure 7.14: Bounding the Reorder Buffer Occupancy

Using the delay distribution derived previously, we can evaluate Equation 7.7 to get a bound on the ROB occupancy. Figure 7.14 compares the theoretical bound with the simulated ROB occupancy for an injection load of 0.8. The probability of overflow for a ROB of size 18 packets is less than 10^{-15}. Repeating our calculations for different loads enables us to study how the ROB size requirement grows with load for the same overflow probability.

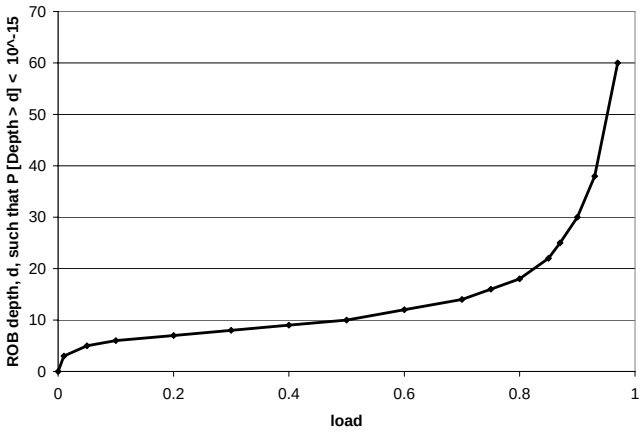


Figure 7.15: ROB size requirements at various injection loads

Figure 7.15 shows the required ROB size with increasing injection load. As in the case of the switch buffers, the ROB size also grows without bound as the load reaches the saturation value. At a load of 0.9 or less, we require at most 30 packet sized ROBs for packets to be reliably reordered and delivered to their destination. A similar calculation can be carried out for bursty injection.

7.6 Summary

In this chapter, we have used recent convergence results in queueing theory to propose a methodology for bounding the buffer depth and packet delay in high radix interconnection networks. We have presented extensive simulations for both non-bursty and bursty injection traffic and shown that the convergence results start to hold for radix values as small as 16. Using the delay distributions, we study packet reordering in the network and estimate bounds for the reorder buffer size in the network. Finally, we use the bounds to propose a routing mechanism with negligible flow-control overhead by either policing the network at the source or introducing a small internal speedup in the network.

Chapter 8

Conclusion and future work

The design of an interconnection network has three aspects — the topology, the routing algorithm used, and the flow control mechanism employed. The topology is chosen to exploit the characteristics of the available packaging technology to meet the bandwidth, latency, and scalability [17] requirements of the application, at a minimum cost. Once the topology of the network is fixed, so are the bounds on its performance. For instance, the topology determines the maximum throughput (in bits/s) and zero-load latency (in hops) of the network. This thesis has demonstrated the significance of the routing algorithm used in the network towards achieving these performance bounds. Central to this thesis is the idea of load-balancing the network channels. A naive algorithm that does not distribute load evenly over all channels, stands to suffer from sub-optimal worst-case performance. However, unnecessary load-balancing is overkill. Spreading traffic over all channels when there is no uneven distribution of traffic, leads to sub-optimal best-case and average-case performance. This thesis explores routing algorithms that strive to achieve high worst-case efficiency without sacrificing performance in the average or best-case.

Chapter 3 examined oblivious routing algorithms and proposed two algorithms that give high worst-case throughput while sacrificing modest performance in the best-case. However, exploiting the oblivious nature of the routing algorithm, an adversary could load a link in the network, resulting in sub-optimal worst-case performance. Chapter 4 introduced adaptivity to counter such an adversary. However, since the decision to misroute

was still oblivious, the best-case performance remained sub-optimal. Chapter 5 then introduced globally adaptive load-balanced (GAL) routing which, unlike any previous routing algorithm, performed optimally in the worst-case and the best-case. Building on these concepts, Chapter 6 extended GAL to an arbitrary symmetric topology. Finally, Chapter 7 examined and analyzed other important performance metrics such as delay and buffer occupancy guarantees in load-balanced, high-radix interconnection networks.

8.1 Future directions

The study of load-balanced routing has opened several problems for future research in interconnection networks. While the problem of finding the exact worst-case traffic pattern has been solved for oblivious routing [49], evaluating the worst-case throughput for an adaptive algorithm remains an open question. In Chapter 6, we proposed an adaptive algorithm with optimal throughput guarantees. Moreover, in Appendix A, we present an efficient algorithm to *bound* the worst-case throughput for a class of adaptive algorithms — minimal adaptive routing. However, to our knowledge, there is no known analytical technique that generates the worst-case pattern for any given adaptive routing algorithm.

Adaptive routing algorithms, whether global (like UGAL) or local (like MIN AD), because they adapt to changes in network traffic, are characterized by both a steady-state and a transient response. In contrast, oblivious algorithms (like VAL and RLB) are entirely characterized by their steady-state response. The transient response of an adaptive algorithm depends not only on the routing algorithm, but also on the details of per-channel flow control. In particular, short per-node queues that give stiff backpressure provide more rapid response to traffic transients. Fully characterizing the transient response of adaptive routing is another interesting open question.

In Chapter 7, we use the bounds we derive to propose a routing mechanism with negligible flow-control overhead by introducing a small internal speedup in the network. The advantages of disabling flow control are manifold. Significant bandwidth is saved which is otherwise used up by flow control credits. Moreover, practical flow control methods like credit based flow control can require substantial buffer space to maintain full throughput on a single virtual channel. We are currently doing a study to quantify the overhead

cost of flow control in interconnection networks. Moreover, the methodology described in this chapter is applicable only to oblivious routing algorithms. Developing a technique for bounding delay and buffer depth for adaptive routing is another interesting topic for future study.

Finally, UGAL's use of channel queues to sense global congestion has applications beyond selecting the quadrant in which to route each packet. This measure of global congestion could, for example, be used to drive a data or thread migration policy. Also, in cases when a packet can be delivered to any one of a set of destinations (e.g., the packet can be sent to any server that provides a particular service), the measure of global congestion can be used to decide which server to select.

Appendix A

Finding the worst-case throughput

A.1 Oblivious routing

The problem of finding the worst-case throughput for any oblivious routing algorithm, R, on a graph $G(V, E)$ was solved by Towles & Dally in [49]. In an oblivious routing algorithm, a source node, s, routes a fixed fraction of its traffic to a destination node, d, across any given edge (channel), e. Building on this property, Towles & Dally show that the problem of finding the worst-case traffic pattern can be cast as a maximum-weight matching of a bipartite graph constructed for each channel in the graph. We now describe this method in some detail.

A.1.1 Worst-case traffic for oblivious routing

Algorithm 1 shows the steps for finding the worst-case traffic pattern for R on $G(V, E)$. Since each source-destination pair sends a fixed fraction of its traffic across an edge, e, the traffic across e is simply a linear sum of the contributions from different source-destination pairs. In order to maximize the load on e, it is sufficient to find a (possibly partial) permutation of such pairs, that maximizes the traffic sent across e. For this purpose, we construct a *channel load* graph (step $1(a)$), G_e, for e from G. G_e is a bipartite graph with two sets of vertices, S and D ($|S| = |D| = |V|$). An edge from $i \in S$ to $j \in D$ in G_e is assigned

a weight, $\delta(i,j)$, equal to the fraction of traffic i sends to j across edge e in G for algorithm R. A maximum weight matching, M_e, is then evaluated (step $1(b)$) for G_e, which corresponds to the worst-case traffic pattern for e. We repeat the process over all channels[1] and return the M_e with the largest weight as our final traffic pattern. Denote the weight of this matching as w. Now, if every source injects traffic at a rate α bits/s, the load on the worst-case edge is $w\alpha$ bits/s. Assuming a channel bandwidth of b bits/s, it follows that the worst-case throughput is b/w bits/s.

Algorithm 1 Finding the worst-case traffic for oblivious routing algorithm, R, on $G(V,E)$

1. For all edges, $e \in E$,
   (a) $G_e \leftarrow \mathrm{Construct\ channel\ load\ graph(G, e)}$.
   (b) $M_e \leftarrow \mathrm{Find\ maximum\ weight\ matching(G_e)}$.
2. *Return the matching with the biggest size.* Return $M_{\max}, \mathrm{s.t.}|\mathrm{M_{max}}| = \max_{\mathrm{e}\in\mathrm{E}} |\mathrm{M_e}|$.

A variant of Algorithm 1 can be used to analytically compute the throughput of R on a specific traffic pattern, Λ. For this purpose, we slightly modify our construction of graph G_e. Given Λ and R, the channel load graph, G_e, for edge e is constructed as follows: an edge from $i \in S$ to $j \in D$ in G_e is assigned a weight, $\delta(i,j)$, equal to the portion of the total traffic i wishes to send to j in Λ, that is routed across edge e in G by algorithm R. The load on e due to Λ is just the sum, W_e, of the weight of all the edges in G_e. Finally the maximum load, $W_{\max}$, over all W_e's, is used to evaluate the throughput, $\Theta = b/W_{\max}$ bits/s.

A.1.2 RLB is worst-case optimal on a ring (Theorem 4)

Proof. In order to prove that the worst-case throughput for RLB is 0.5 on a ring, we first run Algorithm 1 on an 8-node ring (Figure A.1) and then on a generic k-node ring. For any k-node ring, the capacity is given by $2B/N = 8b/k$ bits/s. Since the graph is symmetric, we consider the channel load graph for any one edge, $(3 \rightarrow 4)$, as illustrated in Figure A.2. To

[1] For symmetric graphs, we need to consider any one channel in the network.

find the maximum weight matching, we use the algorithm presented in [19]. The matching returned is shown in Figure A.2 in black arrows. The weight of this matching is $w = (7 + 5 + 3 + 1)/8 = 2$, resulting in a worst-case throughput of $\Theta_{wc} = b/2$, or 0.5 of capacity.

Figure A.1: We consider edge $(3 \rightarrow 4)$ of an 8-node ring

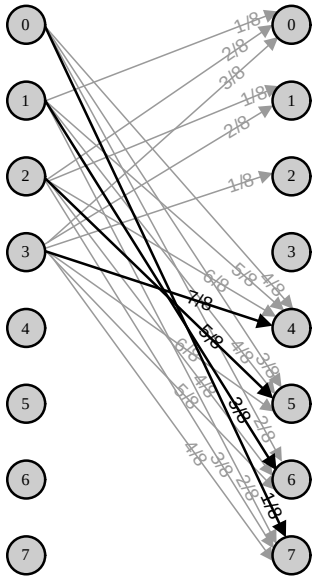


Figure A.2: The channel load graph for edge $(3 \rightarrow 4)$ for RLB on an 8-node ring. The maximum weight matching is shown in black arrows.

In general, for a k-node ring[2], the weight of the maximum weight matching is $w = ((k-1) + (k-3) + \cdots + 1)/k = k/4$. Therefore, $\Theta_{wc} = 4b/k$ or 0.5 of capacity. □

A.1.3 Worst-case permutations for different oblivious algorithms

Applying algorithm 1 on an 8-ary 2-cube, we enumerate the worst case permutations for each of the five oblivious algorithms we studied in Chapter 3:

[2]For simplicity, we assume k is even.

- Dimension Order: The transpose traffic permutation — (i, j) sends to (j, i) — is the worst-case pattern for this scheme. This skewed loading pattern overloads the last right-going link of the 1^{st} row, resulting in a throughput of 0.25 of capacity.
- Valiant: Any traffic permutation is the worst-case pattern.
- ROMM : The worst-case permutation yields a saturation load of 0.208 of capacity. Figure A.3 shows the destination of each source node (i, j) in the worst case permutation.

	0	1	2	3	4	5	6	7
0	(0,3)	(0,0)	(7,5)	(7,0)	(7,6)	(4,1)	(0,1)	(0,2)
1	(6,3)	(0,5)	(0,6)	(0,7)	(1,0)	(5,1)	(6,1)	(6,2)
2	(7,3)	(6,0)	(5,4)	(5,2)	(4,6)	(7,7)	(7,1)	(7,2)
3	(7,4)	(1,5)	(1,6)	(1,7)	(2,0)	(6,7)	(6,6)	(6,5)
4	(0,4)	(4,7)	(4,2)	(5,0)	(2,4)	(5,7)	(5,6)	(5,5)
5	(1,4)	(2,5)	(2,6)	(2,7)	(3,0)	(5,3)	(4,5)	(4,3)
6	(1,3)	(4,4)	(3,2)	(3,3)	(3,4)	(6,4)	(1,1)	(1,2)
7	(2,3)	(3,5)	(3,6)	(3,7)	(4,0)	(3,1)	(2,1)	(2,2)

Figure A.3: Worst case traffic permutation for 2 phase ROMM. Element $[i, j]$ of the matrix gives the destination node for the source node (i, j).

- RLB: Figure A.4 shows the worst case permutation which yields a throughput of 0.313 of capacity.
- RLBth: The worst case permutation for RLBth is very similar to that for RLB and is not presented.

A.2 Bounding the worst-case throughput of MIN AD routing

In the previous section, we described an efficient method to determine the worst-case traffic pattern for any oblivious routing algorithm. The reason this method is restricted to oblivious routing is that an oblivious algorithm sends a predetermined fraction of traffic from a

0	(0.1)	(0,0)	(4,1)	(3,1)	(1,1)	(7,1)	(0,2)	(0,3)
1	(0,4)	(5,0)	(6,6)	(2,6)	(5,1)	(6,1)	(7,2)	(7,3)
2	(7,4)	(6,0)	(3,7)	(4,4)	(4,2)	(5,2)	(6,2)	(6,3)
3	(7,5)	(7,6)	(7,7)	(5,5)	(3,5)	(5,4)	(5,3)	(6,4)
4	(0,7)	(7,0)	(5,6)	(4,5)	(4,6)	(2,7)	(6,5)	(2,5)
5	(0,6)	(6,7)	(4,7)	(1,6)	(4,0)	(3,4)	(2,4)	(1,5)
6	(1,4)	(5,7)	(1,7)	(2,0)	(4,3)	(3,3)	(2,2)	(2,3)
7	(0,5)	(1,0)	(3,6)	(3,0)	(3,2)	(2,1)	(1,2)	(1,3)
	0	1	2	3	4	5	6	7

Figure A.4: Worst case traffic permutation for RLB. Element $[i, j]$ of the matrix gives the destination node for the source node (i, j)

source node, s, to a destination node, d, across a given channel, C. An adaptive algorithm can dynamically decide how much of the traffic goes along C, depending on the state of the network. For this reason, finding the worst-case traffic for adaptive algorithms continues to remain an open question.

A.2.1 An algorithm for bounding the worst-case throughput of minimal routing

In this section, we present a method to bound the worst-case throughput for any minimal routing algorithm. Since the routing algorithm only needs to be minimal and not oblivious, the bound also holds for minimal adaptive routing algorithms. Before we proceed, we define the terms *Necessary Shortest Path (NSP) edge* and *Necessary Shortest Path Edge (NSPE) graph*.

A directed edge, e, of a graph, $G(V, E)$, is an NSP edge for the source-destination pair (s, d), if all shortest paths from s to d include e. It is easy to see that e will be an NSP edge if the shortest distance from s to d in $G(V, E)$ is strictly less than the shortest distance from s to d in $G(V, E \setminus e)$.

An NSPE graph, $\hat{G_e}$, is a bipartite graph for the edge e constructed from the graph $G(V, E)$. It has two sets of vertices — the source set, S, and the destination set, D. Both S and D have cardinality $|V|$. For each pair (i, j), such that $i \in S$ and $j \in D$, there is an

edge from i to j in $\hat{G}_e$, if e is an NSP edge from i to j in $G(V, E)$. Intuitively, the NSPE graph attempts to construct an adversarial traffic pattern that exploits the fact that routes are restricted to minimal paths alone. If the minimal paths necessarily pass through an edge, e, even an adaptive (but minimal) algorithm cannot route around that edge.

Algorithm 2 Finding an adversarial traffic for minimal routing on $G(V, E)$

1. For all edges, $e \in E$,
   (a) $\hat{G}_e \leftarrow \text{Construct NSPE Graph}(\mathrm{G}, \mathrm{e})$.
   (b) $M_e \leftarrow \text{Find maximum matching}(\hat{\mathrm{G}}_\mathrm{e})$.
2. *Return the matching with the biggest size.* Return $M_{\max}, \text{s.t.} |\mathrm{M}_{\max}| = \max_{\mathrm{e}\in\mathrm{E}} |\mathrm{M}_\mathrm{e}|$.

Algorithm 2 describes the steps involved in finding a bound on throughput for any minimal routing on graph $G(V, E)$. For every edge (channel), step 1(a) constructs an NSPE graph, $\hat{G}_e$, corresponding to e. Once $\hat{G}_e$ is constructed, step 1(b) constructs a maximum size matching, M_e, for $\hat{G}_e$. A maximum matching for the NSPE graph corresponds to a (possibly partial) traffic permutation where every source-destination pair sends all traffic across edge e. The process is repeated for every edge and finally the matching with the maximum size is returned in step 2. Denote the weight of this matching as $|M|$. Now, if every source node injects traffic at a rate α bits/s and the channel bandwidth is b bits/s, the maximum injection rate (throughput) is $b/|M|$ bits/s. This is an upper bound on the worst-case throughput of the specific minimal (oblivious or adaptive) routing algorithm.

The run-time complexity of Algorithm 2 is $O(EV^2(V + E)))$. For each edge, e, step 1(a) constructs the corresponding NSPE graph, $\hat{G}_e$. This requires checking for each source-destination pair in G, if e is an NSP edge. There are V^2 source-destination (s, d) pairs in G and checking if an edge is an NSP edge requires running a breadth-first-search (BFS) algorithm twice from s, which takes $O(V + E)$ time. Hence, the total time required for step 1(a) is $O(V^2(V + E))$. Once $\hat{G}_e$ is constructed, step 1(b) constructs a maximum size matching, M_e, for $\hat{G}_e$, which takes $O(V^3)$ time. The process is repeated for every edge and finally step 2 returns the matching with the maximum size, giving a total run time complexity of $O(EV^2(V + E))$. If the graph is symmetric, it is sufficient to consider just

one edge in G, reducing the complexity to $O(V^2(V+E))$.

A.2.2 Experiments on different topologies

Having described the algorithm to bound the worst-case throughput, we now demonstrate how minimal routing can give sub-optimal worst-case performance on three different 64 node topologies as described in Chapter 6. We know from Theorem 1 that the optimal worst-case throughput can be at most 0.5 the network capacity. Using Algorithm 2, we shall discover traffic patterns on which minimal routing performs significantly worse than optimal. Since all the networks are symmetric, we need to construct the NSPE graph for just one edge in each network for the algorithm.

Fully connected graph

We first consider a fully connected graph of $N = 64$ nodes. The bisection bandwidth of this network is $B = N^2b/2$ bits/s. Therefore, the capacity is given by $2B/N = Nb = 64b$ bits/s. The NSPE graph for any edge (u, v) for this graph is a trivial bipartite graph with just 1 edge from u to v. Hence, the matching size is also 1. It follows that the worst-case throughput bound for any minimal routing is $b/|M| = b$ bits/s ($1/64$ of capacity), only 0.78% of the optimal worst-case throughput of 0.5. In general, for an N node fully-connected graph, any permutation traffic yields a throughput of $1/N$ of capacity. While the adversarial traffic for this topology is trivial and can be obtained by inspection, the traffic patterns obtained for the next two topologies are much more subtle.

k-ary n-cube (torus) network

The next topology we consider is an 8-ary 2-cube (8×8 torus) network. The capacity of this network is $kb/8 = b$ bits/s. Again, due to symmetry, we construct the NSPE graph for a single channel. Without loss of generality, let this edge be $(3 \rightarrow 4)$ as shown in Figure A.5.

The NSPE graph is shown in Figure A.6. The figure also shows the corresponding maximum matching (arrows shown in black) for this graph. The size of the matching is 3 (throughput, $\theta = 0.33$ (66.67% of optimal)). As illustrated in Figure A.7, the partial

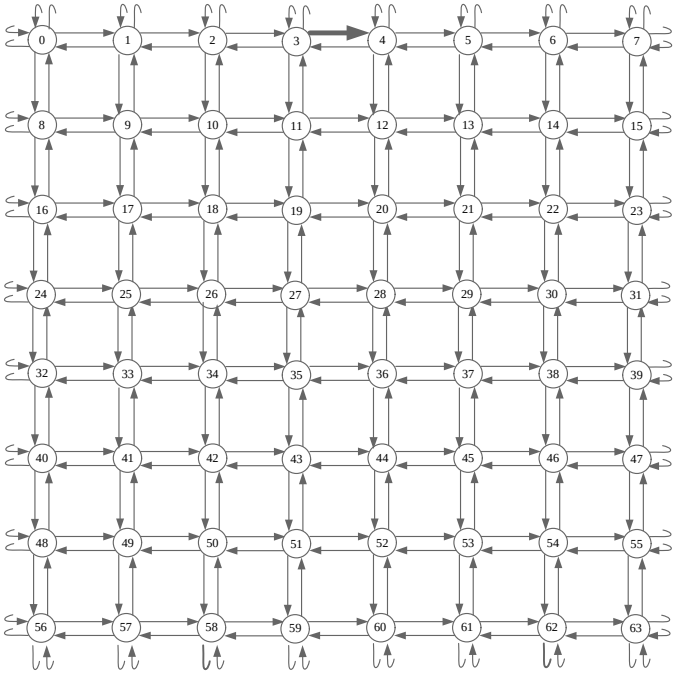


Figure A.5: Constructing an adversarial pattern for minimal routing on an 8×8 torus. We construct the NSPE graph for edge $3 \rightarrow 4$.

permutation obtained from the matching loads the link $(3 \rightarrow 4)$ with 3 flows, and is a subset of the tornado traffic permutation described in Table 2.2.

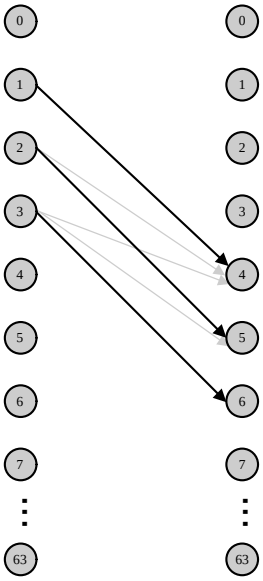


Figure A.6: The NSPE graph for edge $(3 \rightarrow 4)$ and its maximum matching (matched arrows are shown in black)

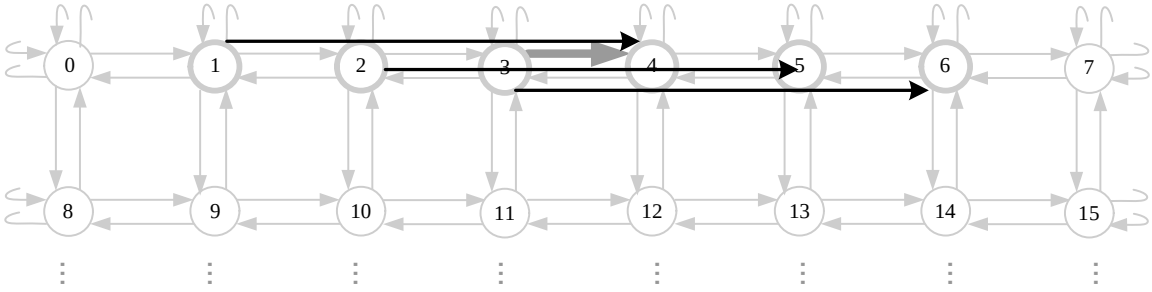


Figure A.7: The adversarial traffic pattern returned by the matching for edge $3 \rightarrow 4$

Cube Connected Cycles

We finally consider a 64 node cube connected cycle (CCC) topology. On this network, our algorithm returns a very subtle traffic pattern, on which any minimal routing algorithm can yield a throughput of 0.2 of capacity (40% of optimal).

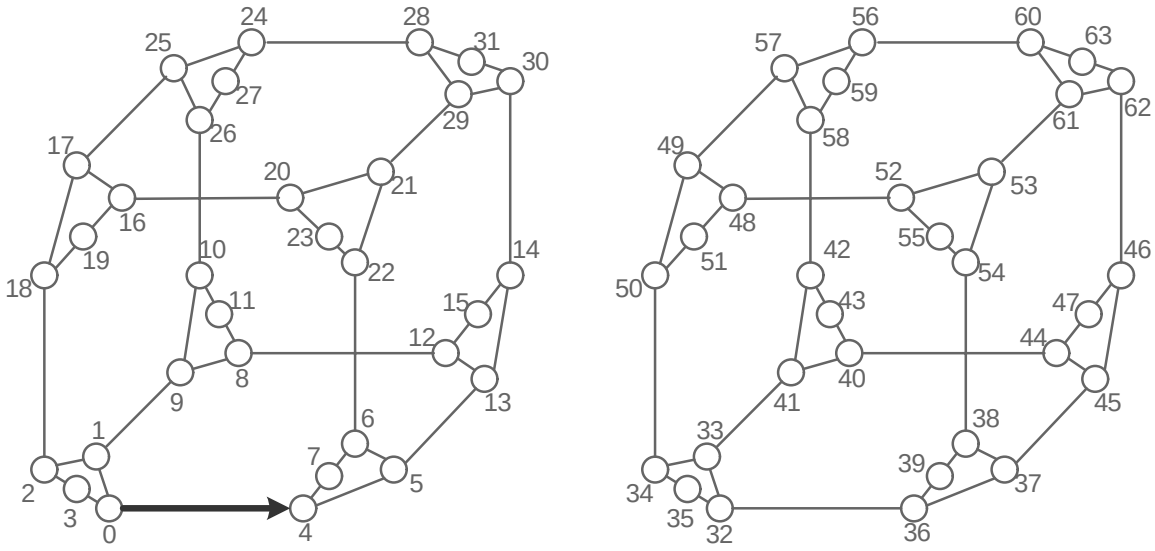


Figure A.8: Constructing an adversarial pattern for minimal routing on a 64 node cube connected cycle. We construct the NSPE graph for edge $0 \rightarrow 4$.

The network is shown in Figure A.8. To avoid unnecessary clutter, we omit the edges going from 1 cube to the other. We also combine the edges in two directions into a single bidirectional edge. The bisection bandwidth, $B = 2^n b$. Since for the 64 node CCC, $n = 4$, the capacity is given by $2B/N = 2(2^n b)/n2^n = 0.5b$ bits/s. We focus on the edge $(0 \rightarrow 4)$ in the network and construct its NSPE graph. The matching returned by the algorithm has a size of 10, implying that the worst-case throughput for any minimal algorithm is at most $0.1b$ bits/s (0.2 of capacity). The exact traffic pattern is shown in Figure A.9.

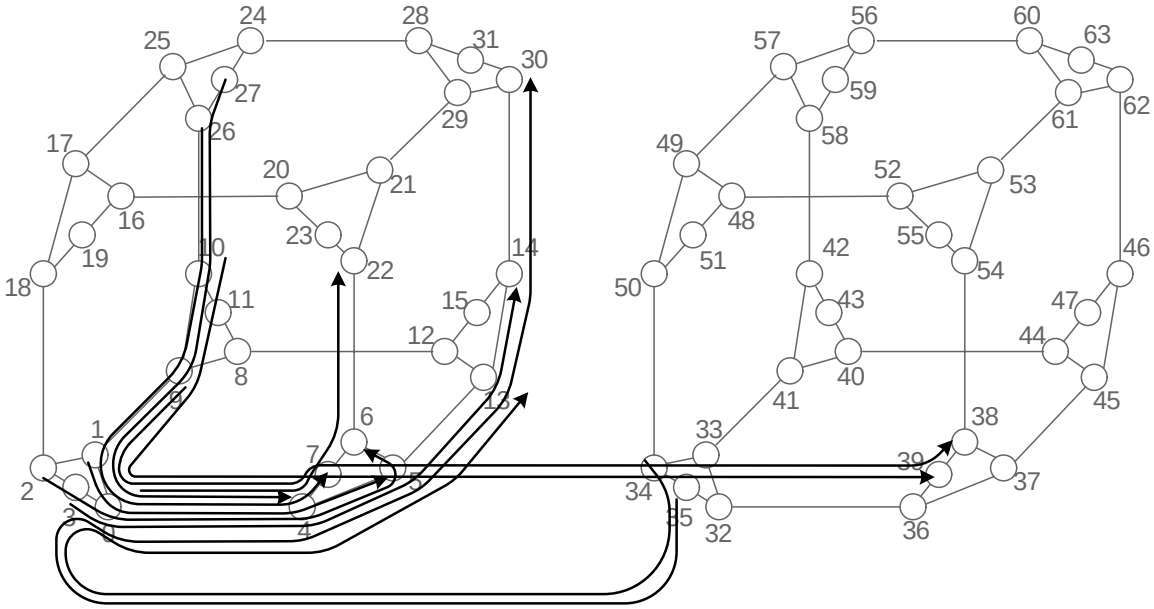


Figure A.9: An adversarial pattern for which the throughput of any minimal algorithm is 0.2 of capacity

Appendix B

GOAL is deadlock-free

We first prove a lemma that will be useful in proving that GOAL is deadlock-free. Lemma 1 proves that the Channel Dependency Graph (CDG*) for the *-channel network associated with GOAL is acyclic. Theorem 8 then proves that no *-channel can participate in a deadlock cycle, proving that GOAL is deadlock-free.

The CDG* for GOAL is a directed graph, $D(C, E)$. The vertices of D are the *-channels of the network and the edges of D are the pairs of *-channels (c_i^*, c_j^*) such that there is a direct dependency from c_i^* to c_j^*. Each *-channel can be uniquely identified by the 4-tuple (*node, id, dimension, direction*) indicating the node whose outgoing channel it is, the id of the *-channel (0 or 1), the dimension of the *-channel, and its orientation in that dimension ($+$ or $-$). Denote *node*$_i$ as the i^{th} coordinate of *node*. We now prove that CDG* for GOAL is acyclic.

Lemma 1. *The *-channel dependency graph (CDG*) for GOAL is acyclic.*

Proof. We first prove that a cycle cannot involve *-channels belonging to the same dimension. Consider dependencies existing between *-channels $(m, 0, d, +)$ and $(n, 0, d, +)$ belonging to a given dimension, d. If a packet, p, enters $(m, 0, d, +)$, it means that d is the most significant dimension it has to correct, and that p has not yet traversed the wrap-around edge in dimension d. Moreover, the chosen direction to correct d is $+$. The key observation here is that GOAL ensures that the direction along which a dimension is corrected is fixed, i.e., a packet cannot traverse in the $-$ direction along d, once it has traversed

in the $+$ direction along d, or vice versa. If p enters $(n, 0, d, +)$ afterwards, it follows that p has still not taken the wrap-around edge along d yet. Hence, if $n_i \leq m_i$, p would have had to move in the $-$ direction along d, which is not allowed by GOAL. Thus, $n_i > m_i$. Consequently, no cycles can exist in CDG* involving the $+$ (or $-$) $*_0$-channels in dimension d. Analogously, no cycles can exist in CDG* involving the $+$ (or $-$) $*_1$-channels in dimension d. Moreover, GOAL ensures that the *-channels along the $+$ ($-$) direction do not have any dependencies on those along the $-$ ($+$) direction in dimension d. Further, it also ensures that there are no cyclic dependencies between the $*_0$ and $*_1$ channels in d. Therefore, if there is a cycle in CDG*, it cannot involve *-channels corresponding to the same dimension.

We next prove that a cycle cannot involve *-channels belonging to different dimensions. Consider a dependency from a *-channel (m, id, u, dir) to a *-channel (n, id', v, dir'), where $u \neq v$. If p enters (m, id, u, dir), it follows that u is the most significant dimension that needs to be corrected. The next key observation is that GOAL does not traverse along a dimension, d, once d is corrected. It follows that when p enters (n, id', v, dir'), then $v < u$. Therefore, there cannot be any cycle involving *-channels corresponding to different dimensions, proving the result. □

Theorem 8. *GOAL is deadlock-free.*

Proof. Suppose that no *-channel ever participates in a deadlock cycle. Then, any packet, p, always has a *-channel in its available set of virtual channels, since CDG* is acyclic (Lemma 1). Hence, no deadlock can arise, given that VCs are assigned fairly and packets are of finite length.

Thus, it suffices to show that no *-channel can ever participate in a deadlock cycle. The proof by induction given in lemma 3.2 of [16] can be applied here verbatim for the GOAL algorithm and is not repeated in this appendix. □

Appendix C

Results for 16-ary 2-cube network

The following table presents data for a 16-ary 2-cube topology for the different algorithms described in this thesis.

Table C.1: Throughput numbers for a 16-ary 2-cube

Algo	Θ_{UR}	Θ_{NN}	Θ_{BC}	Θ_{TP}	Θ_{TOR}	Θ_{WC}	Θ_{avg}
VAL	0.5	0.5	0.5	0.5	0.5	0.5	0.5
DOR	1.0	4.0	0.5	0.25	0.33	0.25	0.31
ROMM	1.0	4.0	0.4	0.54	0.33	0.21	0.45
RLB	0.76	2.33	0.42	0.56	0.53	0.31	0.51
RLBth	0.82	4.0	0.41	0.56	0.53	0.3	0.51
CHAOS	1.0	4.0	0.49	0.56	0.30	0.3	0.53
MIN AD	1.0	4.0	0.49	0.63	0.29	0.29	0.63
GOAL	0.76	2.33	0.50	0.78	0.53	0.5	0.68
GAL	1.0	4.0	0.50	0.78	0.53	0.5	0.73
CQR	1.0	4.0	0.50	0.78	0.53	0.5	0.73

Appendix D

Probability of a queue being empty

In order to find $P(Q = 0)$ for the set up described in Chapter 7.4.2, let us construct the time series equation for the queue depth, Q. If Q_n is the depth of the buffer at the end of time slot n, then the occupancy in the next step will increase by the number of arrivals at slot $n + 1$ and decrease by 1 (0) if the queue is non-empty (empty) at time n. Mathematically, this means

$$Q_{n+1} = Q_n - \Delta_{Q_n} + A_{n+1} \tag{D.1}$$

where Δ_k is a shifted discrete step function

$$\Delta_k = \begin{cases} 1 & k = 1, 2, \ldots \\ 0 & k \leq 0 \end{cases} \tag{D.2}$$

Let us take expectations on both sides of Equation D.1. Since, at steady state, we can drop the subscripts we have

$$E(Q) = E(Q) - E(\Delta_Q) + E(A) \tag{D.3}$$

Now, from Equation D.2, it is obvious that $E(\Delta_Q) = P[Q > 0]$. Substituting in (D.3), we get

$$P[Q = 0] = 1 - E(A)$$

Bibliography

[1] K. V. Anjan and Timothy Mark Pinkston. Disha: a deadlock recovery scheme for fully adaptive routing. In *Proc. of the International Parallel Processing Symposium*, pages 537–543, Santa Barbara, CA, April 1995.

[2] D. Bertsekas and R. Gallager. *Data Networks: Second Edition*. Prentice-Hall, 1992.

[3] Kevin Bolding, Melanie L. Fulgham, and Lawrence Snyder. The case for chaotic adaptive routing. *IEEE Trans. on Computers*, 46(12):1281–1291, 1997.

[4] Cheng-Shang Chang, Duan-Shin Lee, and Yi-Shean Jou. Load balanced birkhoff-von neumann switches, part 1: one-stage buffering. *Computer Communications*, 25:611–622, 2002.

[5] Cheng-Shang Chang, Duan-Shin Lee, and Ching-Ming Lien. Load balanced birkhoff-von neumann switches, part 2: multi-stage buffering. *Computer Communications*, 25:623–634, 2002.

[6] B.W. Char, K.O. Geddes, B.Leong G.H. Gonnet, M.B. Monagan, and S.M. Watt. *Maple V Language Reference Manual*. Springer-Verlag New York, Inc., 1991.

[7] C. Clos. Bell system technical journal. *The Bell System technical Journal*, 32(2):406–424, March 1953.

[8] W. J. Dally, P. P. Carvey, and L. R. Dennison. The Avici terabit switch/router. In *Proc. of Hot Interconnects*, pages 41–50, August 1998.

[9] William J. Dally. Virtual-channel flow control. In *Proc. of the International Symposium on Computer Architecture*, pages 60–68, May 1990.

[10] William J. Dally, Patrick Hanrahan, Mattan Erez, Timothy J. Knight, Francois Labonte, Jung-Ho Ahn, Nuwan Jayasena, Ujval J. Kapasi, Abhishek Das, Jayanth Gummaraju, and Ian Buck. Merrimac: Supercomputing with streams. In *Proceedings of SuperComputing*, Phoenix, Arizona, November 2003.

[11] William J. Dally and Charles L. Seitz. The torus routing chip. *Distributed Computing*, 1(4):187–196, 1986.

[12] William J. Dally and Brian Towles. *Principles and practices of interconnection networks*. Morgan Kaufmann, San Francisco, CA, 2004.

[13] J. Duato, S. Yalamanchili, and L. Ni. *Interconnection Networks An Engineering Approach*. IEEE Press, 1997.

[14] Do Young Eun and Ness B. Shroff. Simplification of network analysis in large-bandwidth systems. In *Proc. of IEEE INFOCOM*, San Francisco, California, April 2003.

[15] Do Young Eun and Ness B. Shroff. Network decomposition in the many-sources regime. *Advances in Applied Probability*, 36(3):893–918, September 2004.

[16] L. Gravano, G. Pifarre, G. Pifarre, P. Berman, and J. Sanz. Adaptive deadlock- and livelock-free routing with all minimal paths in torus networks. *IEEE Trans. on Parallel and Distributed Systems*, 5(12):1233–1252, Dec. 1994.

[17] Amit K. Gupta, William J. Dally, Arjun Singh, and Brian Towles. Scalable opto-electronic network (SOEnet). In *Proc. of Hot Interconnects*, pages 71–76, Stanford, CA, August 2002.

[18] J. Jackson. Jobshop-like queueing systems. *Management Science*, 10(1):131–142, 1963.

[19] D. S. Johnson and C. C. McGeoch. Network flows and matching: first dimacs implementation challenge. *Series in DIMACS*, 12, 1993.

[20] Isaac Keslassy. *The Load-Balanced Router*. Ph.D. Dissertation, Stanford University, June 2004.

[21] Isaac Keslassy, Cheng-Shang Chang, Nick McKeown, and Duan-Shin Lee. Optimal load-balancing. In *Proc. of IEEE INFOCOM*, Miami, Florida, 2005.

[22] Han S. Kim and Ness B. Shroff. Loss probability calculations and asymptotic analysis for finite buffer multiplexers. *IEEE/ACM Trans. on Networking*, 9(6):755–768, 2001.

[23] L. Kleinrock. *Queueing Systems: Volume II*. John Wiley, 1975.

[24] Leonard Kleinrock. *Queueing Systems – Volume 1: Theory.*, pages 191–194, Eqn 5.85. Wiley, New York, 1975.

[25] S. Konstantinidou and L. Snyder. The Chaos Router: A practical application of randomization in network routing. *Proc. of the Symposium on Parallel Algorithms and Architectures*, pages 21–30, 1990.

[26] C. Leiserson. Fat-trees: Universal networks for hardware efficient supercomputing. *IEEE Trans. on Computers*, C-34(10):892–901, October 1985.

[27] D. Linder and J. Harden. An adaptive and fault tolerant wormhole routing strategy for k-ary n-cubes. *ACM Trans. on Computer Systems*, 40(1):2–12, Jan 1991.

[28] D. S. Meliksetian and C. Y. R. Chen. Optimal routing algorithm and the diameter of the cube-connected cycles. *IEEE Trans. on Parallel and Distributed Systems*, 4(10):1172–1178, 1993.

[29] M. Mitzenmacher. Bounds on the greedy routing algorithm for array networks. In *Proc. of the Symposium on Parallel Algorithms and Architectures*, pages 346–353, Cape May, New Jersey, June 1994.

[30] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge Univ. Press, 1995.

[31] T. Nesson and S. L. Johnsson. ROMM routing on mesh and torus networks. In *Proc. of the Symposium on Parallel Algorithms and Architectures*, pages 275–287, Santa Barbara, CA, 1995.

[32] Li-Shiuan Peh and William J. Dally. A delay model for router micro-architectures. *IEEE Micro*, 21(1):26–34, 2001.

[33] Fabrizio Petrini and Marco Vanneschi. k-ary n-trees: High performance networks for massively parallel architectures. In *Proc. of the International Parallel Processing Symposium*, pages 87–93, Geneva, Switzerland, April 1997.

[34] Greg Pfister. *High Performance Mass Storage and Parallel I/O*, chapter An Introduction to the InfiniBand Architecture, Chapter 42, pages 617–632. IEEE Press and Wiley Press, 2001.

[35] Franco P. Preparata and Jean Vuillemin. The cube-connected cycles: a versatile network for parallel computation. *Communications of the ACM*, 24(5):300–309, 1981.

[36] E. Raubold and J. Haenle. A method of deadlock-free resource allocation and flow control in packet networks. In *Proc. of the International Conference on Computer Communication*, pages 483–487, Toronto, Canada, August 1976.

[37] Steven L. Scott and Gregory M. Thorson. The Cray T3E network: Adaptive routing in a high performance 3D torus. In *Proc. of Hot Interconnects*, pages 147–156, August 1996.

[38] Farhad Shahrokhi and D. W. Matula. The maximum concurrent flow problem. *Journal of the ACM*, 37(2):318–334, April 1990.

[39] Arjun Singh and William J. Dally. Buffer and delay bounds in high radix interconnection networks. *Computer Architecture Letters*, 3, December 2004.

[40] Arjun Singh and William J. Dally. Universal globally adaptive load-balanced routing. In *Concurrent VLSI Architecture (CVA) Technical Report*, (ftp://cva.stanford.edu/pub/publications/ugal.pdf), January 2005.

[41] Arjun Singh, William J. Dally, Amit K. Gupta, and Brian Towles. GOAL: A load-balanced adaptive routing algorithm for torus networks. In *Proc. of the International Symposium on Computer Architecture*, pages 194–205, San Diego, CA, June 2003.

[42] Arjun Singh, William J. Dally, Amit K. Gupta, and Brian Towles. Adaptive channel queue routing on k-ary n-cubes. In *Proc. of the Symposium on Parallel Algorithms and Architectures*, pages 11–19, June 2004.

[43] Arjun Singh, William J. Dally, Brian Towles, and Amit K. Gupta. Locality-preserving randomized oblivious routing on torus networks. In *Proc. of the Symposium on Parallel Algorithms and Architectures*, pages 9–19, Winnipeg, Manitoba, Canada, Aug. 2002.

[44] Arjun Singh, William J. Dally, Brian Towles, and Amit K. Gupta. Globally adaptive load-balanced routing on tori. *Computer Architecture Letters*, 3, March 2004.

[45] H. Sullivan and T. R. Bashkow. A large scale, homogeneous, fully distributed parallel machine, I. In *Proc. of the International Symposium on Computer Architecture*, pages 105–117, 1977.

[46] A. S. Tanenbaum. *Computer networks*. Prentice Hall, Upper Saddle River, NJ, third edition, 1996.

[47] M. S. Thottethodi, A. R. Lebeck, and S. S. Mukherjee. BLAM: A high-performance routing algorithm for virtual cut-through networks. In *Proc. of the International Parallel and Distributed Processing Symposium*, Nice, France, April 2003.

[48] Sam Toueg and Jeffrey D. Ullman. Deadlock-free packet switching networks. In *Proc. of the ACM Symposium on the Theory of Computing*, pages 89–98, Atlanta, Georgia, United States, 1979.

[49] Brian Towles and William J. Dally. Worst-case traffic for oblivious routing functions. In *Proc. of the Symposium on Parallel Algorithms and Architectures*, pages 1–8, Winnipeg, Manitoba, Canada, August 2002.

[50] L. G. Valiant and G. J. Brebner. Universal schemes for parallel communication. In *Proc. of the ACM Symposium on the Theory of Computing*, pages 263–277, Milwaukee, MN, 1981.

[51] Damon J. Wischik. The output of a switch, or, effective bandwidths for networks. *Queueing Systems - Theory and Applications*, 32(4):383–396, 1999.

[52] Damon J. Wischik. Sample path large deviations for queues with many inputs. *Annals of Applied Probability*, 11(2):379–404, May 2001.

[53] D. Yates, J. Kurose, D. Towsley, and M. Hluchy. On per-session end-to-end delay distributions and the call admission problem for real-time applications with qos requirement. *Journal on High Speed Networks*, 3(4):429–458, 1994.

[54] Rui Zhang-Shen and Nick McKeown. Designing a predictable internet backbone network. In *Proc. of HotNets*, San Diego, CA, November 2004.