DISTRIBUTED ROUTER FABRICS

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

Brian Towles
September 2004

© Copyright by Brian Towles 2005
All Rights Reserved

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.

____

Balaji Prabhakar

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.

____

Stephen P. Boyd

Approved for the University Committee on Graduate Studies.

Abstract

As Internet traffic continues to double every year, the demands placed on the IP routers that deliver this traffic also increase. Traditional IP router architectures cannot scale to meet these demands, forcing architects to explore alternative designs. This thesis explores one alternative, the distributed router fabric. Distributed router fabrics are rooted in the interconnection networks used in supercomputers and their idea is simple: design the largest efficient router possible and then replicate and interconnect these routers to scale the fabric's switching capacity. This approach adopts the engineering advantages of interconnection networks, but also presents several challenges that are the focus of this thesis.

An important property of any IP router is the switching bandwidth it can provide to incoming traffic. As we show, this guaranteed line rate can be found for an arbitrary distributed fabric by solving a series of maximum-cost flow problems, significantly improving the characterization provided by previous approaches. Building on this result, we also show that maximizing the line-rate guarantee of a particular distributed fabric can be achieved by designing its routing algorithm. The optimal routing algorithm design problem can be cast as a convex program and, as a result, globally optimal routing algorithms can be efficiently determined for any fabric.

Once a packet's route through the fabric is determined, a flow-control method delivers that packet from source to destination. We show that existing fixed-size flits, or flow-control units, necessitate large control overheads and introduce variable-size flits to solve this problem. By carefully constraining amount of variation in our flit size, the hardware implementation is kept simple while overhead is greatly reduced. Long packets can also reduce the efficiency of flow control and we show that splitting these long packets into many shorter packets and then reassembling them provides higher fabric throughput.

Acknowledgments

To my colleges at Stanford, a large part of this thesis and my success as a graduate student is due directly to the positive impact you have had on me. First, I want to thank my readers, Professor Balaji Prabhakar and Professor Stephen Boyd, for both their time and influence. Professor Prabhakar along with Professor Nick McKeown and their groups provided a great resource at Stanford for me to learn the details of IP router design. Professor Boyd introduced me to the ideas of convex optimization through his classes that play a major role in this thesis. I would like to thank the members of the Concurrent VLSI Architecture group at Stanford with whom I had the privilege of working. It was a pleasure to collaborate with both the networking group (Amin, Arjun, Jin, John) and the Imagine team (Brucek, John, Scott, Ujval), in particular. I must also thank my officemates, Ujval and Sarah. Finally, I could write an entire chapter of thanks to my advisor, Bill Dally, but will choose to simply say that his support and guidance throughout my graduate career were invaluable. I am both incredibly proud and immensely lucky to be able to refer to myself as one of his students.

To my friends and family and their support during my graduate career, I am grateful. This is especially true for my parents who consistently offered both patience and wisdom from my first day of kindergarten to my last day of graduate school. Also, thanks to my dad for carefully proofreading a draft of this thesis.

Contents

Abstract **v**

Acknowledgments **vi**

1 Introduction **1**

1.1 The form and function of IP routers 1

1.2 Distributed router fabrics 4

1.3 Contributions 5

1.4 Outline 6

2 Worst case of oblivious routing algorithms **8**

2.1 Preliminaries 9

2.2 Linearity of oblivious routing algorithms 10

2.3 The homogeneous case 11

2.3.1 Narrowing the worst-case search 12

2.3.2 Bipartite graph representation 15

2.3.3 Maximum-weight matching 15

2.4 Implementation and experiments 17

2.4.1 Computing bipartite edge weights 19

2.4.2 Worst-case example 23

2.4.3 Throughput distributions 27

2.5 Extensions 29

2.5.1 Heterogeneous cases 30

2.5.2 Symmetry optimizations 32

2.6 Related work . . . 35
2.7 Summary . . . 35

3 Design of worst-case optimal routing algorithms 37
3.1 Worst-case routing design as a flow problem . . . 38
3.1.1 Routing algorithm representation . . . 38
3.1.2 Worst-case problem . . . 41
3.2 Optimization approaches . . . 42
3.2.1 Linear programming representation . . . 43
3.2.2 Projected subgradient method . . . 46
3.2.3 Symmetry optimization . . . 50
3.3 Experiments . . . 52
3.3.1 Design tradeoffs . . . 53
3.3.2 Comparison to Valiant's algorithm in faulty and irregular topologies 60
3.4 Implementation . . . 64
3.4.1 Routing algorithm approximation . . . 64
3.4.2 Hardware organization . . . 69
3.5 Adaptive routing . . . 74
3.6 Related work . . . 76
3.7 Summary . . . 76

4 Flow control 78
4.1 Background . . . 79
4.1.1 IP packets . . . 79
4.1.2 Flow control in interconnection networks . . . 80
4.2 Flit sizing . . . 82
4.2.1 Fixed-size flits . . . 82
4.2.2 Variable-size flits . . . 87
4.2.3 Implementation . . . 89
4.3 Packet sizing . . . 91
4.4 Reordering . . . 95
4.5 Summary . . . 102

5 Conclusion **104**

5.1 Traffic admission and quality of service 105

5.2 Taking advantage of traffic structure 108

A Notation and definitions **111**

B A subgradient method for the WCORDP **113**

Bibliography **118**

List of Tables

2.1 Ideal throughput as a fraction of network capacity for DOR and ROMM over several traffic patterns on an 9-ary 2-cube. 24

3.1 Summary of routing algorithms 55

3.2 TCAM encoding of a path of weight 28ϵ destined to node d. 72

4.1 Summary of control overheads for a typical 256-node network with source routing ($P_0 = 34$ bits and $F_0 = 24$ bits). 86

List of Figures

1.1 The basic organization of an input-queued IP router. 3
1.2 A crossbar and load-balanced router fabric. 3
1.3 A simple distribution router fabric using a ring topology. 5

2.1 Example of independent contributions to a channel's load. 11
2.2 Construction of the bipartite graph for finding channel load due to a particular permutation. 16
2.3 A 4-ary 2-cube (torus) network. 18
2.4 Example dimension order and ROMM routes in a 2-dimensional network. . 20
2.5 Code for the DOR algorithm. 21
2.6 Code for the ROMM algorithm. 22
2.7 Possible executions of ROMM when routing from $(0, 0)$ to $(1, 2)$ in a 8-ary 2-cube network. 23
2.8 Adversarial pattern for ROMM in a 9-ary 2-cube 26
2.9 Worst-case throughput of DOR and ROMM for k-ary 2-cubes (left) and for 5-ary n-cubes (right). 27
2.10 Distribution of throughput over the set of permutation traffic patterns for DOR on a 9-ary 2-cube. 28
2.11 Distribution of throughput over permutation traffic patterns for ROMM on a 9-ary 2-cube. 29
2.12 Maximum-cost flow problem used to solve heterogeneous cases of the worst-case traffic problem. 31

3.1 Code to compute a projection onto the probability simplex. 50

3.2 Tradeoff between average packet distance and worst-case throughput for a 8-ary 2-cube network. 54
3.3 Routing algorithms produced by interpolation. 58
3.4 Tradeoff between average packet distance and average-case throughput for a 8-ary 2-cube network. 60
3.5 The worst-case throughput of both an optimal oblivious algorithm and Valiant's randomized algorithm in a 6-ring with a variable number of faulty channels. 61
3.6 The worst-case throughput of both an optimal oblivious algorithm and Valiant's randomized algorithm on faulty torus and random networks. . . . 62
3.7 A random network with 16 nodes and 48 channels created using linear preferential attachment. 63
3.8 Cumulative distribution of path weights for a routing algorithm that minimizes path length while maintaining optimal worst-case throughput in a 8-ary 2-cube network. 65
3.9 Performance of approximated routing algorithms for a 8-ary 2-cube network. 68
3.10 Hardware organizations for route lookup. 71
3.11 Minimum area hardware organization for route lookup as a function of the average number of paths per source-destination pair and the path granularity. 73
3.12 A network with a gap in the worst-case performance of an adaptive routing algorithm and an optimal oblivious routing algorithm. 75

4.1 A distribution of Internet (IP) packet lengths captured at the University of Memphis on October 4th, 2002. 80
4.2 A typical packet format for flit-buffer flow control. 81
4.3 An example of overhead vs. packet length. 85
4.4 Control overhead as a function of the flit size for $P_0 = 34$ bits and $F_0 = 24$ bits. 86
4.5 Input microarchitecture for a variable-size flit fabric router. 90
4.6 An example of flit and credit rate mismatch that can occur when credit-based flow control is used with variable size flits. 91

4.7 Saturation throughput of an 8-ary 2-cube (torus) network for various IP packet sizes under uniform traffic. 92
4.8 A simple model of the limits on saturation throughput due to packet length. 95
4.9 Two cases of the average fabric and total latency vs. offered load in an 8-ary 2-cube fabric. 98
4.10 Reorder buffer timing for a packet sent on cycle t being delayed by a packet sent k cycles earlier. 100
4.11 Distribution of reordering delay and reorder buffer size for an 8-ary 2-cube network with 1 flit per packet and uniform traffic. 101

5.1 An example point of presence topology. 109

B.1 Optimality gap vs. iteration for the subgradient method in a 4-ary 2-cube. . 116
B.2 Primal and dual solutions vs. iteration for the subgradient method in a 4-ary 2-cube. 116

Chapter 1

Introduction

1.1 The form and function of IP routers

Internet protocol (IP) routers are one of the key building blocks of today's Internet — they are responsible for delivering the IP packets upon which the world-wide web, electronic mail, and many other applications are based. This thesis, in particular, focuses on *core IP routers*. These are the routers with the highest switching capacities (1 Tb/s or more) and are typically located in the central offices of Internet service providers. While there are complex economic and sociological arguments to be made about the future growth of the Internet, our work is based on the fact that recent trends indicate the bandwidth demand placed on the Internet is approximately doubling every year [49]. This, in turn, drives the need for a corresponding growth in router capacity.

There are many possible ways to define the capacity of a router, but today's IP routers are designed to provide *line-rate service*. The idea of line-rate service is that the router itself never becomes a bottleneck to the flow of packets through the Internet. Rather, the only limiter of the rate of packet delivery is the bandwidth of the channels connecting these routers. Viewed in another way, IP routers are designed to be robust under a wide variety of conditions. If a large fraction of traffic shifts from one destination to another, perhaps as the result of an important news event, then the IP router should continue to deliver those packets up to the full rate of the newly popular output channel.

Line-rate service, by itself, could be considered the vanilla flavor of a much richer set

of possible services. There are many additional quality of service (QoS) features that are provided by and have been proposed for IP routers. Traffic could be split into multiple classes, some with a higher priority than others or, some traffic, such as real-time video streams, may require latencies guarantees, to mention a few examples. However, the main focus of this thesis is on providing line-rate service. As we will see, this can still be a challenging goal. Also, this is not to say that additional QoS features are not important or challenging to implement and we revisit this topic in Section 5.1.

Given a particular set of services, a designer is then faced with task of implementing an IP router. The progression of packets through a typical router can be broken into three broad stages: packet processing, packet storage, and packet switching (Figure 1.1).[1] Both processing and packet storage take place on line cards and the lines cards can be thought of as the "brains" of the IP router. Common processing tasks include route lookup (output port determination), statistics gathering, and packet classification and filtering. Once processing is complete, the packet is stored in a line card's memory until it is forwarded for switching. The switching phase of the router is responsible for simply delivering packets from their input port to their output port — the "brawn" of the router. While the capacity of the line cards is proportional to the line rate, the capacity of the router's switch, often called the router's *fabric*, must grow with the line rate times the number of router ports.

The rapid scaling of line rates combined with the demand for higher port counts on routers has fueled a shift away from traditional, centralized fabric organizations. For example, the crossbar fabric (Figure 1.2[a]) has been popular in core routers, but its centralized control becomes a liability as the router's total switching bandwidth approaches 1 Tb/s [43]. One alternative fabric that solves the centralized control problem of the crossbar is the load-balanced architecture shown in Figure 1.2(b). Many researchers have explored an approach in this vein, including Keslassy's load-balanced router [39], which is based on the ideas of Chang et al. [17], and Iyer and McKeown's parallel packet switch [36]. These approaches are closely related to Clos's network [21] and the ideas of channel slicing and inverse multiplexing. A similar organization appears in Cisco's CRS-1 [20], which provides a total capacity of up to 46 Tb/s.[2]

[1]Routers may duplicate or reorder these phases based on their particular architecture, but the basic functionalities are common to most IP routers.

[2]IP router manufacturers often present the total capacity of a router as the input plus output bandwidth.

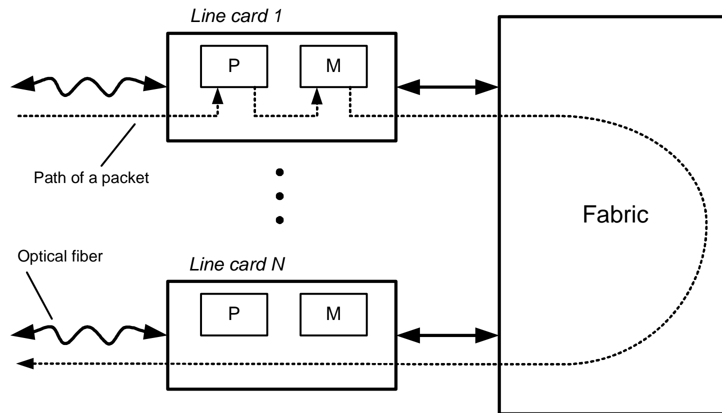


Figure 1.1: The basic organization of an input-queued IP router. The path of a typical packet is shown as a dotted line. Packets arrive from long-haul optical fibers are processed (P) and then stored in memories (M) before being forwarded to the fabric. The fabric is then responsible for deliver the packets to their destination ports.

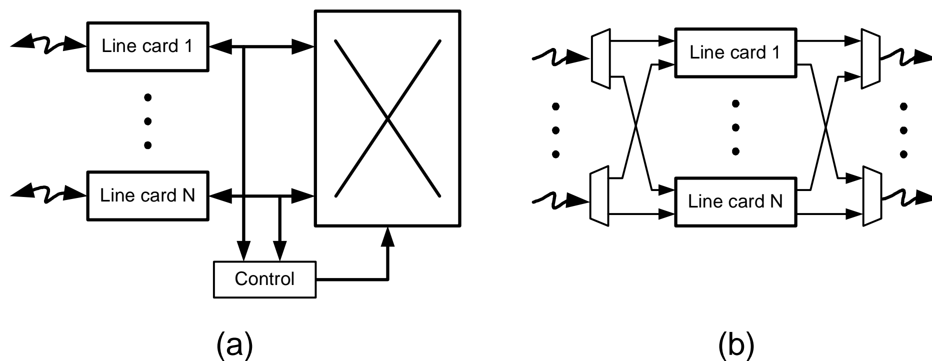


Figure 1.2: A (a) crossbar and (b) load-balanced router fabric. As shown, the fabric of the load-balanced organization is split in two stages — one before the line cards and one after.

These alternatives to the crossbar are certainly more scalable and alleviate the problems associated with centralized control. However, their design philosophy is still centralized — the architectures are all designed around a particular topology and the movement of packets is synchronized and intimately tied to that topology. The drawback of this philosophy is that it sacrifices flexibility: flexibility in the topology of the fabric and flexibility in the movement and timing of packets through the fabric. This ultimately leads to a more expensive router. Instead, we explore a solution built around flexibility — a distributed router fabric.

1.2 Distributed router fabrics

Distributed router fabrics are rooted in the interconnection networks traditionally found in supercomputers and, notably, this approach has been adopted in the Avici TSR [23]. The idea is simple: design the largest *efficient* router possible and then replicate and interconnect these "building block" routers to scale the fabric's switching capacity. The fabric itself is a network within the IP router. Figure 1.3 shows a small example of this idea. An important point is that these building block or *fabric routers* can be simpler than even the simplest IP router. The complex line cards and their long-haul fiber interfaces, large buffers, and processing capabilities are not replicated in the fabric routers. As shown, the line card functionality remains at the periphery of the fabric. Also, each fabric router operates asynchronously and with local control. Using hop-based flow control between the fabric routers keeps their buffering requirements low, further simplifying their design. See Dally and Towles [26] for more details on the design of typical interconnection networks.

With the flexibility of this approach comes more design decisions. First, how should the fabric routers be interconnected? (What is the fabric's topology?) The correct answer is tied to the cost of the available packaging technologies. For example, long signals are inherently more expensive to implement than short signals — the bandwidth electrical signaling decreases with distance and even low-cost multimode optical signaling costs at least 20 times as much per unit bandwidth as electrical signaling. The least expensive solution must trade off signal distance with the number of signals. Gupta et al. [32] explore this

Using this definition the CSR-1 has 92 Tb/s of capacity.

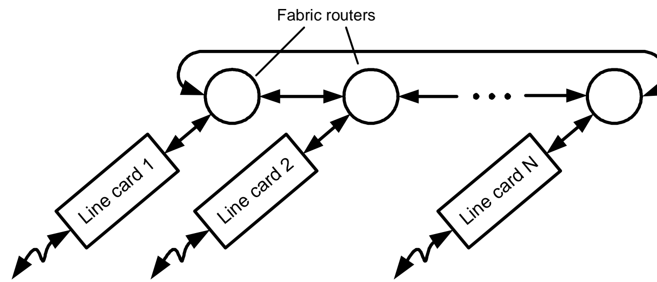


Figure 1.3: A simple distribution router fabric using a ring topology.

tradeoff in general interconnection networks. A key advantage of distributed fabric is that this exploration is possible because their operation is not tied to a particular topology.

Another design variable closely tied to topology is the size of the fabric routers. In the context of interconnection networks, routers are generally designed to fit on a single chip. This greatly simplifies their design as no control decision is subject to the bandwidth limitations and latency penalty associated with crossing a chip boundary. However, this is not the only possible design point for fabric routers as we explore in Section 5.2.

Once we have the fabric's topology, and implicitly the size of its fabric routers, we are faced with the key questions that are the focus of this thesis. First, for a particular fabric, what are line-rate guarantees we can make? As we will see, the answer is intimately tied to how packets are routed through the fabric itself. Moreover, we can specifically design routing algorithms that maximize the guaranteed line rate. Second, how do we design a flow control technique (scheduling packets in time) to maximize the efficiency of a distributed fabric?

1.3 Contributions

This thesis includes several key contributions to the design of distributed router fabrics.

- *The guaranteed line rate of a given distributed fabric can be efficiently found.* By taking advantage of the linearity of oblivious routing algorithms, the problem of finding the traffic pattern that determines the fabric's line-rate guarantee can be solved as a series of maximum-cost flow problems. This enables an efficient, polynomial-time

approach for finding this guarantee for any distributed router fabric topology when oblivious routing is used.

- *Line-rate optimal oblivious routing algorithms can be efficiently designed.* Building on our result for finding the line-rate guarantee for a particular fabric and its routing algorithm, the design of worst-case optimal routing algorithms can be cast as a convex optimization problem. Solutions to these optimization problems can be efficiently found, enabling the automatic design of routing algorithms that achieve the largest possible line-rate guarantee for a given distributed fabric.

- *Variable-size flits can be used to greatly reduce the control overheads of fixed-size flits.* The units of flow control (flits) used in interconnection networks have traditionally been fixed size to simplify implementation. However, this restriction necessitates large control overheads. Our approach of variable-size flits carefully loosens this restriction to maintain a simple hardware implementation while greatly reducing control overhead.

- *Long packets should be split to maximize fabric throughput.* Allowing packets to be spread across many routers within the distributed fabric leads to massive throughput losses due to resource coupling. By splitting these long packets into many smaller packets and reassembling and reordering the pieces before transmission out of the IP router, fabric throughput becomes nearly independent of the incoming IP packet size.

1.4 Outline

For the remainder of this thesis, we first focus on how routing affects line-rate guarantees in distributed router fabrics. Chapter 2 describes a technique to find the guaranteed line-rate of an arbitrary distributed fabric by solving a series of maximum-cost flow problems. This technique is applied to known routing algorithms and compared against the characterization given by existing approaches. Building on the technique of Chapter 2, Chapter 3 shows that the maximum-cost flow formulation can be extended to the design optimal routing algorithms for a particular fabric by solving convex optimization problems. This enables

an exploration of tradeoffs in the design of worst-case optimal routing algorithms and a evaluation of existing routing algorithms in various topologies. Our focus then shifts to flow control and in Chapter 4, we address flow-control issues specific to distributed fabrics, including flit and packet sizing and reordering. Conclusions and future work are presented in Chapter 5, and notation and definitions are included in Appendix A for reference. Details of a subgradient method for worst-case optimal routing design are included in Appendix B.

Chapter 2

Worst case of oblivious routing algorithms

As discussed in the introduction, when designing an IP router, we are concerned with the maximum line-rate guarantee that can be made for that router. Informally, the line rate is the highest bandwidth that can be supported by the IP router's inputs and outputs without the router itself ever becoming a bottleneck to the flow of packets. We formulate this as a worst-case problem — a particular fabric must have enough internal capacity to deliver packets under *any* sequence of packet arrivals that does not oversubscribe (exceed) the line rates of any input or output. The internal capacity of a fabric is determined by the bandwidth of the fabric's channels and also the ability of a fabric's routing algorithm to balance load over these channels where the routing algorithm determines the path each IP packet takes through the fabric.

In this chapter, we focus on the set of oblivious routing algorithms, that is, any routing algorithm that ignores network state when selecting a path through the fabric. Oblivious algorithms have an important linearity property (Section 2.2) that makes their analysis tractable. Building on this, we show that the problem of finding the worst-case channel load, and thus the guaranteed line rate, can be cast as a maximum-weight matching of a bipartite graph (Section 2.3). The worst-case pattern is then used to determine the worst-case throughput of a particular system. Extensions of this method to switch fabrics with unequal port bandwidths along with optimizations based on fabric symmetry are presented

in Section 2.5.

Finding the exact worst-case throughput offers a significant improvement in accuracy over existing techniques used to estimate the worst case. Previous studies of routing algorithms generally chose "bad" traffic patterns that the authors felt represented worst-case or near worst-case behavior [11, 48]. However, for the example presented in Section 2.4, these traditional techniques overestimate the worst-case throughput of the ROMM routing algorithm [48] by approximately 47%. Moreover, despite the fact the that ROMM uses randomization to more evenly spread load, its worst-case throughput is shown to be approximately half that of simple dimension-order routing for 2-dimensional torus networks.

2.1 Preliminaries

Before we tackle the problem of finding the maximum line rate that a fabric can guarantee, several assumptions and a description of the fabric model are useful. First, we abstract each fabric as a graph — a collection of fabric routers (nodes) connected by directed edges (channels). The set of nodes is denoted as $\mathcal{N}$ and the size of this set is N. Likewise, the set of channels is $\mathcal{C}$ and its size if C. A channel c's bandwidth is b_c. Without loss of generality, each of the fabric routers has an associated line card. We do not explicitly include these line cards in our figures to avoid unnecessary clutter. Routing is the process of finding a path (a sequence of channels) through the graph from a particular source line card to a destination line card. The source and destination of a particular IP packet is predetermined as part of how routing is performed in IP networks.

For this chapter and the next, we determine the throughput of a fabric solely by the bandwidth of the channels. That is, as long as all the channels are not *saturated*, or being asked to deliver more packets than any channel's bandwidth supports, the fabric can deliver all the packets causing the demand. This model assumes an ideal flow control method that is able to perfectly schedule these packet without ever causing idle time on the channels. Thus, any performance determined by this model is an upper bound on what is practically obtainable.

2.2 Linearity of oblivious routing algorithms

Oblivious routing algorithms determine the path an IP packet takes through the fabric without taking into account any network state. They can, though, use randomization to select amongst several paths probabilistically. For example, consider a packet routing from source 5 to destination 8. A simple oblivious routing algorithm might have two possible paths, A and B, from 5 to 8 and, for a particular packet, could select the path to use with a coin flip — on heads, path A is used and, on tails, path B is used.

As each packet arrives at its input line card and is routed through the fabric, demand is placed on the fabric's channels. We refer to this as channel load. The average load on a particular channel c can be expressed using a set of random processes: the arrival process $a(t)_{ij}$ is one if a bit arrives at input i destined for node j at time t and zero otherwise; the routing process $r(t)_{cij}$ is one if the routing algorithm uses channel c as part of a route from node i to node j at time t and zero otherwise. Using these definitions, let the system begin at an arbitrary time $t = 0$ and then the time average load γ_c on channel c is

$$\gamma_c = \lim_{n \to \infty} \left[\frac{1}{n} \sum_{t=0}^{n-1} \sum_{i,j \in \mathcal{N}} a(t)_{ij} r(t)_{cij} \right].$$

Based our our network model, the fabric is stable and can deliver a particular arrival sequence as long as the channel load on each channel is less than the channel's bandwidth ($\gamma_c \leq b_c,\ \forall c \in \mathcal{C}$).

Since we are interested in time average channel loads, we make the weak assumptions that both the arrival and routing processes are stationary and ergodic. Then, by linearity of expectation,

$$\gamma_c = \sum_{i,j \in \mathcal{N}} E\left[a(t)_{ij} r(t)_{cij}\right].$$

At this point, the key feature of oblivious routing algorithms comes into play. Since the path selection is independent of network state, the arrival and routing processes must also be independent. So, it follows that

$$\gamma_c(\Lambda, X) = E\left[a(t)_{ij}\right] E\left[r(t)_{cij}\right] = \sum_{i,j} \lambda_{i,j} x_{cij}. \tag{2.1}$$

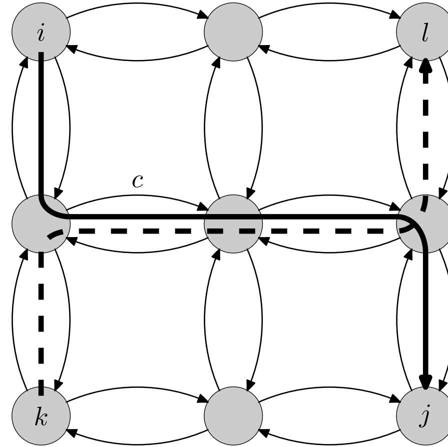


Figure 2.1: An example of two independent contributions to channel c's load. One bit per cycle is being sent from node i to node j, crossing channel c (solid route). Another bit per cycle is sent from node k to node l and also uses channel c (dashed route). Both of these routes contribute a load of one bit per cycle across channel c. The total load on channel c is simply 2 bits per cycle.

Here, λ_{ij} is the average amount of traffic traveling from source i to destination j (bits/s). We typically refer to the matrix of these traffic rates Λ as the traffic matrix or traffic pattern. Each routing variable x_{cij} gives the probability of using channel c when routing a packet from source i to destination j. As is evident in the form of (2.1), the channel load of an oblivious routing algorithm is a sum of the individual contributions due to each source-destination pair. An example of this property is also shown in Figure 2.1. We will make heavy use of this linearity in the following sections.

2.3 The homogeneous case

Using the linearity principle developed in the previous section, the guaranteed line rate of a particular fabric can be found. For simplicity, we focus on a homogeneous case in this section: all injection (ejection) bandwidths into (out of) the network are considered to be equal. Also, the number of sources and destinations are considered to be equal. In practice, this is a common case — all the line cards in a given router often have the same injection and ejection bandwidth. The analysis is generalized to heterogeneous cases in

Section 2.5.1.

The guaranteed line rate of a fabric is, in essence, a worst-case guarantee made by the fabric, that is, a guarantee on the throughput over the set of traffic patterns that do not oversubscribe the input or output ports of the fabric. The basic approach for finding the worst-case throughput is straightforward: focus on a particular channel, find the worst-case traffic pattern for that channel, and then repeat this process for the remaining channels. Still, we are faced with searching the infinite set of admissible traffic patterns. This problem is addressed by first showing that it is sufficient to search only the permutation traffic patterns (Section 2.3.1). Then, by constructing a bipartite graph whose edge weights are determined by the incremental load placed on a corresponding channel in the network (Section 2.3.2), we show that the solution to a maximum-weight matching problem defines the worst-case traffic for that network channel (Section 2.3.3).

2.3.1 Narrowing the worst-case search

In Section 2.2, we established that, under mild assumptions, the channel load induced by a particular traffic pattern and a particular routing algorithm can be expressed as a sum of the contributions from each source-destination pair. Moreover, for a fixed routing algorithm, channel load is linear in the traffic pattern. This has several importance consequences, which we take advantage of in this section to simplify the search for the worst case.

First, remember that our goal is to find the largest injection and ejection bandwidths that can be supported by the switch fabric over all admissible traffic patterns. In the homogeneous case, all injection and ejection bandwidths are equal to a throughput Θ, so the admissible patterns are those whose corresponding traffic matrix has row and column sums of at most Θ. That is,

$$\begin{aligned}
\sum_{j\in\mathcal{N}} \lambda_{ij} &\leq \Theta, \quad \forall i \in \mathcal{N} \\
\sum_{i\in\mathcal{N}} \lambda_{ij} &\leq \Theta, \quad \forall j \in \mathcal{N} \\
\lambda_{ij} &\geq 0, \quad \forall i, j \in \mathcal{N}
\end{aligned}$$

While we are ultimately interested in finding the largest value of Θ supported by the switch fabric over all admissible patterns, this task can be simplified by momentarily assuming unit injection and ejection bandwidths (letting $\Theta = 1$ bit/s). This assumption reduces the search space to doubly-substochastic traffic matrices — matrices with row and column sums less than or equal to one. Due to the result of von Neumann [72], any such doubly-substochastic matrix can be made doubly-stochastic (rows and columns of exactly one) by strictly increasing the matrix entries. Since we are searching for worst-case traffic and adding traffic can never increase throughput, it is sufficient to consider only the doubly-stochastic matrices in the worst case.

Then, for a particular channel c, assume that a doubly-stochastic traffic matrix Λ^* maximizes the channel load on c. The traffic matrix Λ^* may induce more or less load on the channel than its capacity b_c. However, by scaling the traffic pattern by the ratio of channel bandwidth to the channel load induced by Λ^*, channel c can be exactly saturated,

$$\gamma_c\left(X, \frac{b_c\Lambda^*}{\gamma_c(X,\Lambda^*)}\right) = b_c,$$

as a direct result of the linearity of channel load. Moreover, as proved in the following theorem, this scaling factor times the unit bandwidth used for Λ^* gives the largest value of Θ that the switch fabric can support.

Theorem 1. *The scaling factor $b_c/\gamma_c(X,\Lambda^*)$ is the smallest fraction of the unit injection rate needed to saturate channel c.*

Proof. Assume a scaling factor $\alpha < b_c/\gamma_c(X,\Lambda^*)$ saturates channel c for some different doubly-stochastic traffic matrix Λ:

$$\gamma_c(X, \alpha\Lambda) = b_c.$$

Applying linearity and substituting,

$$\alpha = \frac{b_c}{\gamma_c(X,\Lambda)} < \frac{b_c}{\gamma_c(X,\Lambda^*)}.$$

It follows that $\gamma_c(X,\Lambda^*) < \gamma_c(X,\Lambda)$, but this is a contradiction because Λ^* maximizes

channel load. Therefore, $b_c/\gamma_c(X, \Lambda^*)$ is the smallest fraction of the injection rate needed to saturate c. □

Based on this result, our previous assumption of considering just doubly-stochastic matrices to find a worst-case pattern is justified because we can later scale this pattern and find the actual throughput supported by the network. Consequently, we can further simplify the search to include only permutation matrices as shown by the following theorem.

Theorem 2. *For any oblivious routing algorithm X, a permutation matrix can always load a channel c as heavily as a doubly-stochastic traffic matrix Λ.*

Proof. Assume that Λ gives a throughput lower than any permutation matrix. This implies Λ loads channel c more heavily than any permutation. By the result of Birkhoff [9], any doubly-stochastic traffic matrix Λ can be written as a weighted combination of permutation matrices:

$$\Lambda = \sum_{i=1}^{n} \phi_i P_i, \quad \text{s.t.} \sum_{i=1}^{n} \phi_i = 1 \text{ and } \phi_i \geq 0.$$

A permutation P^* is found such that

$$P^* = \underset{P \in \{P_1, \ldots, P_n\}}{\text{argmax}} \ \gamma_c(X, P).$$

The corresponding total load on c can be written using linearity as

$$\begin{aligned}\gamma_c(X, \Lambda) &= \sum_{i=1}^{n} \phi_i \gamma_c(X, P_i) \\ &\leq \sum_{i=1}^{n} \phi_i \gamma_c(X, P^*) = \gamma_c(X, P^*).\end{aligned}$$

P^* loads channel c at least as heavily as Λ, but this is a contradiction. Therefore, a permutation matrix can always give the same load on c as a doubly-stochastic traffic matrix. □

Using these two theorems, worst-case traffic patterns can be found by first searching all permutation matrices while momentarily ignoring the feasibility of the solutions. Then,

the permutation matrix that most heavily loads a channel is scaled to account for the actual channel bandwidth, and the scaling factor gives the smallest injection and ejection bandwidths needed to saturate that channel.

2.3.2 Bipartite graph representation

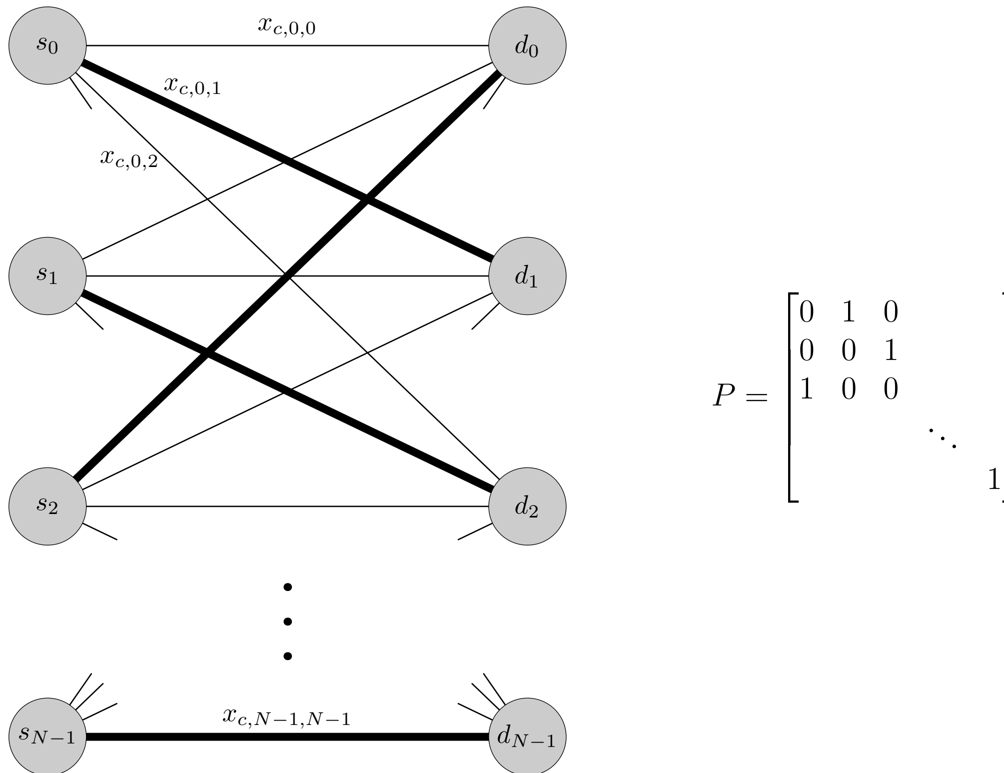
Given the results from the previous section, the search for worst-case traffic patterns can be restricted to just searching permutation patterns. To facilitate this, a bipartite graph can be used to represent the load on a single channel due to any particular permutation. For our graph, the first set of N nodes are used to represent packet sources and the second set of N nodes represent the packet destinations. Edges are added between every source and destination node for a total of N^2 edges, as shown in Figure 2.2. There is a one-to-one correspondence between permutation matrices and *perfect matchings* of this bipartite graph, where a perfect matching is a subset of the graph edges such that each node is incident with exactly one edge in the subset. Also, note that this graph's structure is unrelated to the topology of the underlying interconnection network.

The graph's construction is finished by weighting each edge from source node i to destination node j with the amount of load contributed to a particular channel c when packets are routed from i to j, which is x_{cij} (Figure 2.2). Techniques for finding the edge weights are discussed in Section 2.4.1. Using these weights, the amount of load due to a specific permutation is just the sum of the edge weights in its corresponding bipartite matching. This sum is called the *weight* of that matching.

2.3.3 Maximum-weight matching

Using the bipartite construction, a *maximum-weight matching* of the graph is found. From the correspondence between matchings and permutations, finding a maximum-weight matching is equivalent to evaluating

$$\gamma_{\text{c,max}}(X) = \max_{P \in \mathbb{P}} \gamma_c(X, P),$$



$$P = \begin{bmatrix} 0 & 1 & 0 & & \\ 0 & 0 & 1 & & \\ 1 & 0 & 0 & & \\ & & & \ddots & \\ & & & & 1 \end{bmatrix}$$

Figure 2.2: Construction of the bipartite graph for finding channel load due to a particular permutation. A perfect matching (bold edges) and its corresponding permutation P are also shown. The rows (columns) of P correspond to the source (destination) nodes of the bipartite graph. As an example, the matching's edge from source node 0 to destination node 1 corresponds to the 1 in entry $(0, 1)$ of the permutation matrix.

where $\mathbb{P}$ is the set of all permutation matrices. By repeating this operation over all the channels, the ideal worst-case throughput can be determined as

$$\Theta_{\text{ideal,wc}}(X) = \min_{c \in C} \left[b_c / \gamma_{\text{c,max}}(X) \right].$$

It is important to realize that while the worst-case throughput is certainly unique, the traffic pattern that causes this worst case may not be. There may also be non-permutation patterns that realize the worst-case throughput.

An $O(N^3)$ maximum-weight, bipartite matching algorithm exists [42], and therefore, finding the worst-case channel load requires $O(CN^3)$ time. For typical fixed-degree networks, such as tori or meshes, the size of C is proportional to N and the run time is $O(N^4)$. The maximum size of C is N^2, corresponding to a fully-connected network, which bounds the time of the overall algorithm to $O(N^5)$. So, the worst-case traffic and, thus, the the guaranteed line rate, of a network can be found in time polynomial in the size of that network.

2.4 Implementation and experiments

As an illustration of both how to implement the worst-case algorithm described in Section 2.3 and the practical utility of this approach, we present a comparison of two minimal, oblivious routing algorithms for k-ary n-cube (torus) networks. (See Chapter 5 of Dally and Towles [26].) Torus networks contain k^n nodes placed in an n-dimensional grid with k nodes in each row of a dimension. Channels connect nodes whose address differs by ± 1 in one dimension modulo k. Also, since tori are direct networks, there is a source and destination associated with each of the nodes in the network. Figure 2.3 shows an example of a 4-ary 2-cube network. Low dimensional torus networks are popular in implementation because they keep channel lengths short and are used in systems such as the Avici TSR IP router [23] and the Cray T3E supercomputer [57].

To find exact worst-case traffic patterns in tori and other network topologies, exact edge weights are needed for the bipartite graphs. Section 2.4.1 outlines a simple implementation for finding these weights. Once edge weights are determined, a standard maximum-weight

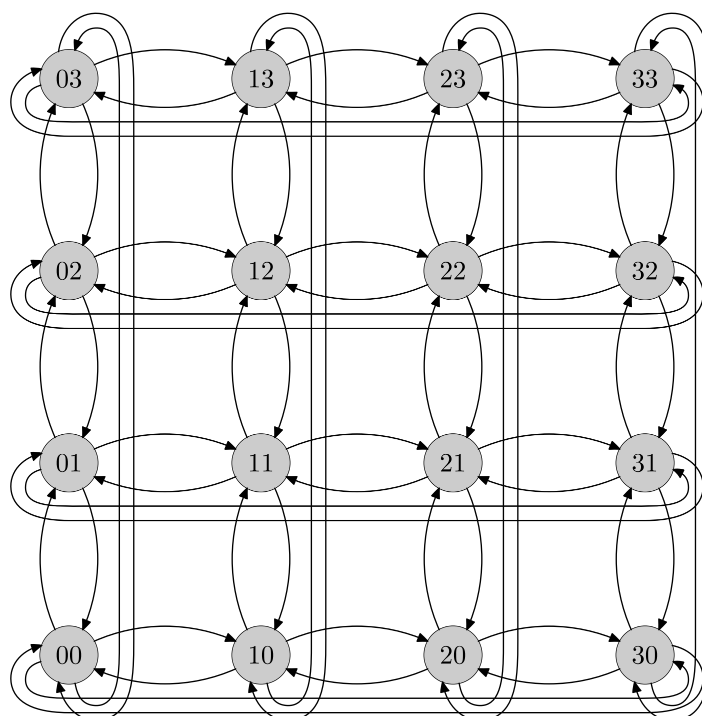


Figure 2.3: A 4-ary 2-cube (torus) network.

matching algorithm can be applied. (See Section 12.4 of Ahuja et al. [2], for example.) An analysis of the worst cases for two example algorithms is then presented in Section 2.4.2. Finally, we can use the infrastructure created for finding the worst case to also compute the distribution of throughputs and answer questions about how common the worst case is (Section 2.4.3).

2.4.1 Computing bipartite edge weights

To compute the edge weights of the bipartite graph used to find the worst case, a conceptually simple experiment must be run: for each source-destination pair in the network, route 1 bit per second of traffic between the source and destination and measure the average load placed on a focus channel in the network. While simple, strictly speaking, this procedure may take time exponential in the size of the network to complete. For example, if a routing algorithm spreads load between all minimal paths in the network (the number of these paths grows exponentially), the time required to visit each path and determine the average load on a channel will be exponential. Fortunately, practical algorithms rarely use an exponential number of paths. For some implementation approaches this would imply an exponential amount of hardware, but there is a more fundamental reason that a very large number of paths is not necessary: any set of channel loads induced by a routing algorithm between a particular source-destination pair can always be realized with at most C paths, where C is the number of channels in the network. (See Section 3.5 of Ahuja et al. [2].) Informally, these C paths provide just enough free variables to set the load on each of the channels in the network. A formal treatment of these issues and the associated time complexity in computing edge weights is discussed by Towles and Dally [69], but for the remainder of this section we focus on a practical method for finding edge weights.

For this section and the next, we consider two minimal routing algorithms for torus networks. The first algorithm is dimension-order routing (DOR) [66]. DOR deterministically routes a packet along a shortest path, routing completely in one dimension before moving on to the next dimension. An example dimension-order route from source i to destination j for a 2-dimensional network is shown as a solid line in Figure 2.4. In this example, DOR routes first in the horizontal dimension and then in the vertical dimension.

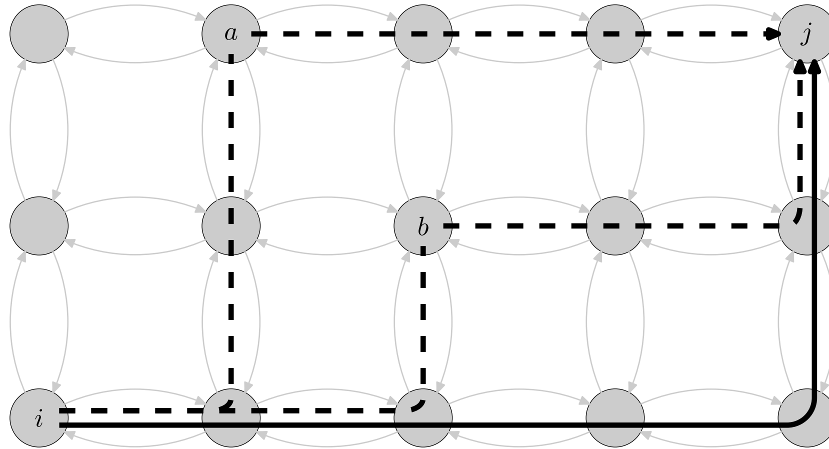


Figure 2.4: Example dimension-order (solid line) and ROMM routes (dashed lines) in a portion of a 2-dimensional network.

The second algorithm is the two-phase variant of the randomized algorithm (ROMM) described by Nesson and Johnson [48]. ROMM routes a packet from source to destination by uniformly choosing a random intermediate node within the minimal quadrant. The minimal quadrant is the set of nodes along any minimal length path between the source and destination. The packet then uses DOR from the source to the intermediate and repeats the same algorithm from the intermediate to the destination. Two example ROMM routes, which use intermediate nodes a and b respectively, are shown in Figure 2.4 as dashed lines.

Given that DOR is a deterministic and simple routing algorithm, it is not difficult to develop a procedure for directly computing its corresponding bipartite graph's edge weights. This task is significantly more difficult for ROMM and, in general, must be approached on a per-case basis for each routing algorithm to be analyzed. To avoid developing specialized methods for each routing algorithm, we adopt a general approach based on a "C" implementation of an algorithm.

For example, a C implementation of DOR is shown in Figure 2.5. The routing function takes as inputs a pointer to the network class (`net`), a randomization class (`rs`), a current node (`current`), the destination node (`dest`), the network radix (`k`), and the network dimension (`n`). A single call of the `DOR` function is responsible for routing the packet from the current node to the destination. Load is added to each channel along the path using the `AddLoad` method. The amount of load added to each channel is determined by calling

```
1 : void DOR( Network *net, Random *r,
2 :           int *current, int *dest, int k, int n )
3 : {
4 :   Channel *c;
5 :   int     steps, dim, dir;
6 :
7 :   for ( dim = 0; dim < n; dim++ ) {  // Traverse dims in order
8 :
9 :     // Determine the minimal direction in the dim of the network
10:     steps = ( dest[dim] - current[dim] + k ) % k;
11:     if ( steps > k/2 ) { // CCW around the ring is minimal
12:        steps = k - steps;
13:        dir = -1;
14:      } else {           // CW around the ring is minimal
15:        dir  = 1;
16:      }
17:
18:     // Follow the minimal direction in the current dimension and
19:     // add load to the channels traversed
20:     for ( int s = 0; s < steps; s++ ) {
21:       c = net->GetChannel( current, dim, dir );
22:       c->AddLoad( r->BranchProb( ) );
23:       current[dim] = ( current[dim] + dir + k ) % k;
24:     }
25:   }
26: }
```

Figure 2.5: Code for the DOR algorithm.

the `BranchProb` method of the randomization class. In the case of DOR, the result of `BranchProb` is always 1 bit/s and one call to `DOR` per source-destination pair is sufficient to determine the edge weights of the bipartite graph. While it may seem unnecessary to query a method that will always return 1 bit/s, the power of this approach is better appreciated in the case of the ROMM routing algorithm.

Figure 2.6 shows the C code for the ROMM algorithm following the same format used with DOR. The key difference in this case is that ROMM also uses randomization to determine its path choice. Each random intermediate requires a random number for each dimension in the network. So, the ROMM code calls the `RandInt` method of the randomization class — the method returns a random integer between its first and second arguments, inclusive. Then, one technique for determining average channel loads for a

```
1 : void ROMM( Network *net, Random *r,
2 :            int *current, int *dest, int k, int n )
3 : {
4 :   int steps, im[n];
5 :
6 :   // Select the random intermediate node (im) one dim at a time
7 :   for ( int dim = 0; dim < n; dim++ ) {
8 :
9 :     // Find the extend of the minimal quadrant in the current dim
10:     // (steps) and select a random number in that range.
11:     steps = ( dest[dim] - current[dim] + k ) % k;
12:     if ( steps > k/2 ) { // CCW is minimal
13:       steps = k - steps;
14:       im[dim] = ( current[dim] - rs->RandInt( 0, steps ) + k ) % k;
15:     } else {             // CW is minimal
16:       im[dim] = ( current[dim] + rs->RandInt( 0, steps ) ) % k;
17:     }
18:   }
19:
20:   // Two-phases of dimension-order routing
21:   DimOrder( net, rs, current, im,   k, n );
22:   DimOrder( net, rs, im,      dest, k, n );
23: }
```

Figure 2.6: Code for the ROMM algorithm.

particular source-destination pair (and thus edge weights), would be to perform numerical integration over several calls to `ROMM`. If 1000 samples where used in the integration, for example, the `BranchProb` method would be set to return 0.001 — the probability of a single sample.

The same code can also be used to exactly determine channel loads by systematically visiting each path of the routing algorithm once. To determine loads in the case of ROMM, the routing function is called just as before. However, now each call to `RandInt` pushes an element onto a stack stored in the randomization class (`r`) and returns zero — the smallest value in the requested range. In the next invocation of the routing function, the return value of the last call to `RandInt` is incremented to one and all other calls again return zero. The last call to `RandInt` continues to increment with each invocation until it reaches the maximum value of its range. At this point, it is popped off of the stack and the return value of the second to last call to `RandInt` is incremented. The routing function continues to

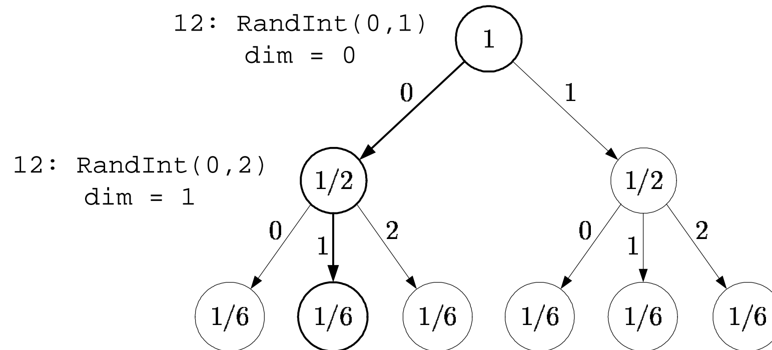


Figure 2.7: Possible executions of `ROMM` when routing from $(0, 0)$ to $(1, 2)$ in a 8-ary 2-cube network.

be invoked until the entire stack is emptied. This ensures each possible sequence of values return by the calls to `RandInt` is evaluated exactly once.

As illustrated in Figure 2.7, this approach is equivalent to a depth-first traversal of the tree containing all the possible executions of the routing algorithm for a particular source-destination pair. The channel load returned by `BranchProb` is determined by the probability of a particular execution — this is simply equal to one over the product of all the ranges in the `RandInt` calls used in the corresponding branch. For example, the branch in Figure 2.7 shown in bold represents the choice of $(0, 1)$ as an intermediate node, which occurs with probability $1/6$. By manipulating these probabilities as rational numbers (an integer numerator and denominator), exact values for each edge weight can be determined.

2.4.2 Worst-case example

Qualitatively, one might except ROMM to have better throughput than DOR because it more evenly distributes its traffic through the network whereas DOR concentrates all the traffic between each source-destination pair along a single path. In fact, several theoretical results [12, 37] have indeed shown that deterministic (single path) routing is provably bad in the worst case and reinforce the intuition about the benefits of spreading load.

To test this intuition, the performance of these two algorithms was compared against uniform random traffic and two permutations that are typically relied upon to demonstrate poor performance [11, 48]: bit-complement and transpose. The tornado pattern was also

Table 2.1: Ideal throughput as a fraction of network capacity for DOR and ROMM over several traffic patterns on an 9-ary 2-cube.

Pattern	DOR	ROMM
Uniform	1	1
Bit-complement	0.556	0.332
Transpose	0.278	0.421
Tornado	0.278	0.278
Worst of 10^4 permutations	0.278	0.302
Worst-case	0.278	0.173

considered, where each node sends packets $(k-1)/2$ hops to the right in the horizontal dimension. In addition to these patterns, a trial of 10^4 random permutation matrices was generated and the worst-case throughput for both algorithms over the 10^4 permutations was determined. Each permutation was chosen uniformly from the space of all permutations using Durstenfeld's algorithm [27]. As shown in Table 2.1, ROMM performed as well as DOR on most of these conventional metrics. The notable exception was the bit-complement pattern for which dimension-order routing happens to be particularly well used. Moreover, despite its additional load balancing, these ad-hoc tests reveal little worst-case advantage for ROMM over DOR.

Next, the algorithm presented in Section 2.3 was used to determine the worst case for both DOR and ROMM (Table 2.1). All calculations were performed using integer arithmetic and the worst-case results are exact. Values shown in the table have been rounded to three significant digits. The worst-case of DOR matched the result of 0.278 of capacity found in the random permutations. However, ROMM's exact worst-case of 0.173 was significantly less — only 62.3% of DOR's worst-case throughput.

The reason for ROMM's lower worst-case throughput is illustrated by constructing an adversarial traffic pattern (Figure 2.8). The pattern is started with the tornado pattern in a single row of the network. This row is outlined with a dashed box in the figure. The source-destination pairs in the tornado pattern send all their traffic within a single dimension, resulting in a single minimal path between each pair. Because both ROMM and DOR are minimal routing algorithms, they both route all their traffic along the single minimal path

between each pair of the tornado pattern. In our example of a 9-ary 2-cube, this gives a channel load of 4 bits/s in the tornado row. For DOR, this is a worst case — their are only 4 nodes in any row that can send traffic along a dimension-order route crossing a particular horizontal channel. Similarly, their are only 4 nodes in any column that can receive traffic along a route crossing a particular vertical channel. However, the channel load in ROMM can be increased further by selecting source-destination pairs outside the tornado row whose minimal quadrants include a channel of the tornado row. By setting up a large number of these crossing patterns, as shown in Figure 2.8, the channel load can be increased to almost twice that of DOR's worst case.

A further comparison of the worst case throughput of ROMM and DOR on k-ary 2-cubes shows that for odd values of k beyond 9, DOR approaches approximately 25% of capacity, while ROMM approaches 12.5% of capacity — half that of DOR (Figure 2.9). Similar results hold for even values of k. So, although ROMM might qualitatively seem to be a more balanced routing algorithm, these experiments show that simple DOR has superior worst-case performance on k-ary 2-cubes.

However, ROMM does perform better relative to DOR for higher dimensional networks (larger values of n). For example, in the case of a 5-ary 3-cube, ROMM has a worst-case throughput which is approximately two times that of DOR (Figure 2.9). Much of ROMM's advantage for higher dimensional networks comes from the fact that DOR's worst-case load must grow with $\sqrt{N} = k^{n/2}$ because it is a deterministic routing algorithm [12, 37]. Since the capacity of a torus network is only a function of k, increasing n for a fixed value of k reduces DOR's throughput as a fraction of capacity exponentially. Specifically, for odd k, DOR's exact worst-case is

$$\frac{k+1}{4k^{\lceil n/2 \rceil}}$$

of capacity. While ROMM is not subject to the $\sqrt{N}$ bound because it is a randomized algorithm, its performance is still only marginally better for larger values of n.

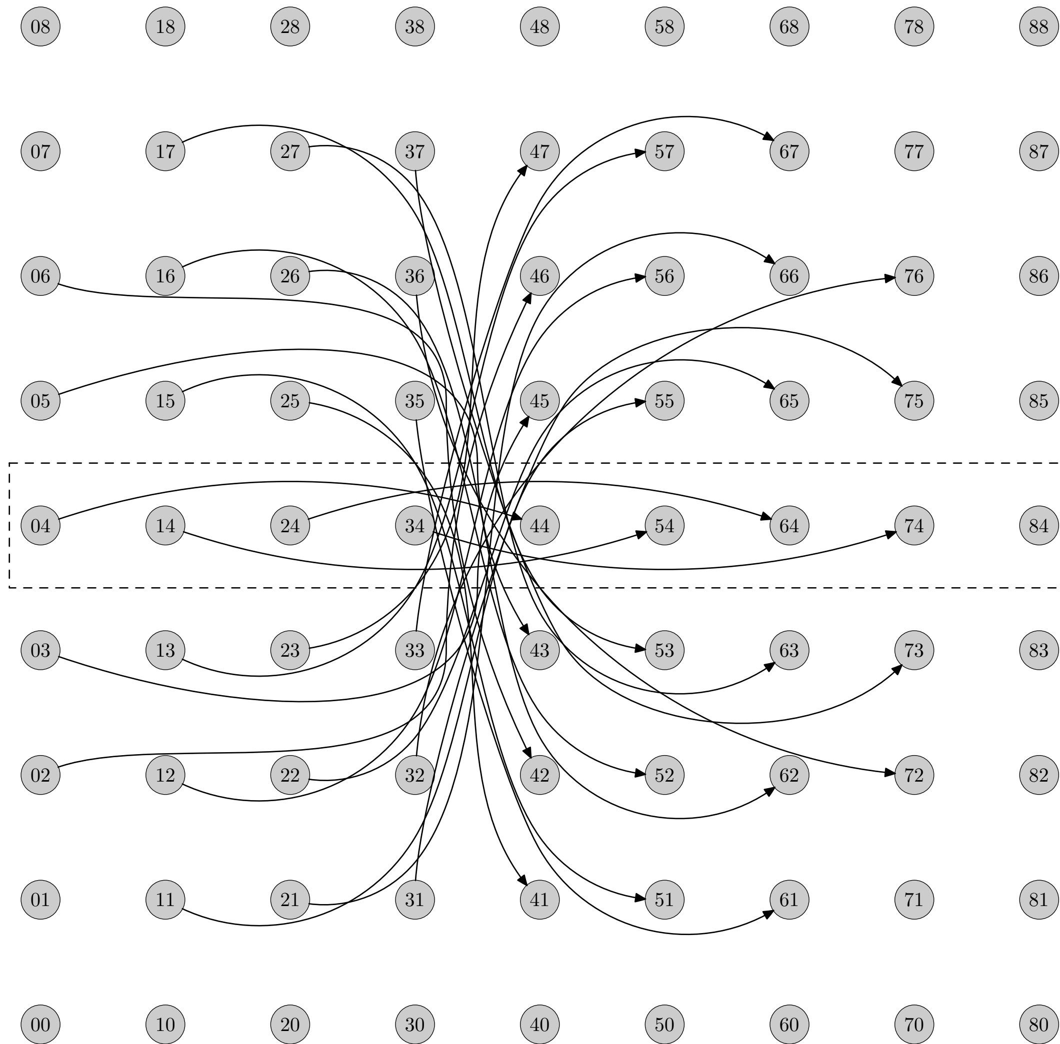


Figure 2.8: Adversarial pattern for ROMM in a 9-ary 2-cube. The network channels are not shown for clarity and arrows connect source-destination pairs in the worst-case traffic pattern. The nodes in the middle row (dashed box) run the tornado traffic pattern and nodes outside this row send across the row, maximizing load on the channel from node 34 to node 44.

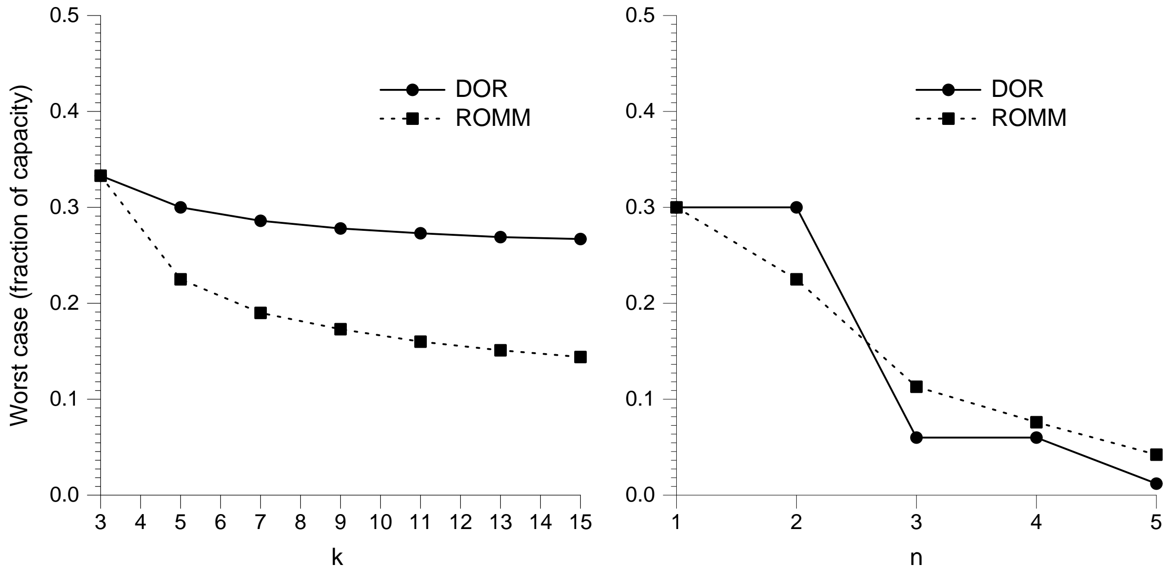


Figure 2.9: Worst-case throughput of DOR and ROMM for k-ary 2-cubes (left) and for 5-ary n-cubes (right).

2.4.3 Throughput distributions

While worst-case throughput is the primary design consideration for IP router applications of interconnection networks, in other situations it may also be useful to know the distribution of throughput over a particular set of a traffic patterns. For example, these distributions also reveal how common the worst case is over a particular traffic distribution.

Figure 2.10 shows the distribution of DOR's throughput when sampling uniformly from the set of permutation traffic patterns (10^4 samples). Since each source sends all of its traffic to a single destination in a permutation and DOR does not use randomization, channels are always loaded with 1, 2, 3, or 4 bits/s of traffic. These loads correspond directly to the 4 vertical bars shown in the figure. On average, DOR's throughput is 78.6% of capacity, but the worst-case throughput is also fairly common, occurring for about 0.7% of the permutations.

As shown in Figure 2.11, ROMM performs much better on average than DOR in the same test — ROMM's average throughput is approximately 96.8% of capacity. While we have already seen that ROMM has a low worst-case throughput of 17.3% of capacity, the distribution also shows that the worst case is very rare. More than 99% of the traffic can be

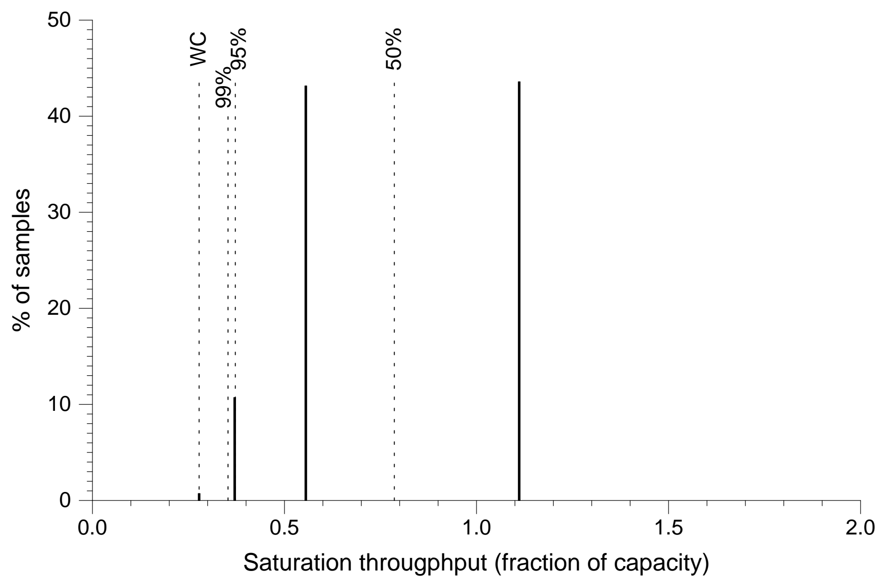


Figure 2.10: Distribution of throughput over the set of permutation traffic patterns for DOR on a 9-ary 2-cube. Dotted vertical lines indicate the expected throughput achieved on a given percent of the patterns. For example, 99% of the patterns achieve a throughput of at least 35.3% of the network's capacity. The worst-case throughput is marked with the line labeled "WC".

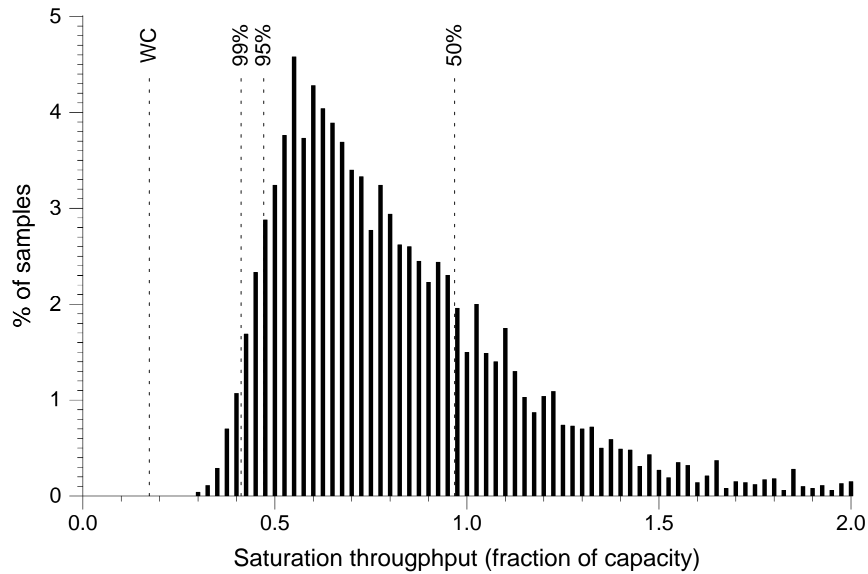


Figure 2.11: Distribution of throughput over permutation traffic patterns for ROMM on a 9-ary 2-cube.

delivered at twice the throughput of the worst case. So, while the intuition that spreading load should improve performance did not hold in terms of the worst case, it did improve the average-case performance for ROMM. This opens an interesting question about the tradeoff between average- and worst-case performance, which we will explore quantitatively once we develop the tools of Chapter 3.

2.5 Extensions

In this section, we develop extensions to the basic method for finding the worst case presented in the previous sections. As described in Section 2.5.1, the worst case for heterogeneous sources and destinations can be found by solving a generalized version of the maximum-weight matching problem. Also, many common networks and routing algorithms exhibit some symmetry. Section 2.5.2 shows how this symmetry can be used to reduce both the number of channels considered for the worst case and the time required to compute edge weights.

2.5.1 Heterogeneous cases

In a heterogeneous worst-case problem, both the restriction that each source (destination) injects (ejects) at the same bandwidth is lifted along with the restriction that there are an equal number of sources and destinations. However, the basic objective still remains: find a traffic pattern that is both admissible and induces the maximum possible load on a given channel.

To approach the heterogeneous case, we can use the idea of a maximum cost flow problem. (See Ahuja et al. [2].) The flow problems we are interested in begin with a directed graph. Each edge of the graph has both a capacity (bits/s) and a cost ($ per bit/s) and each node of the graph can either sink or supply flow to the graph. Then, the objective in a maximum-cost flow problem is finding a flow from supplies to sinks that maximizes the total cost (sum of flow cost over all edges).

Figure 2.12 shows a maximum-cost flow problem which can be used to determine the worst-case traffic for a channel in the heterogeneous case. As before, a bipartite graph is created with nodes corresponding to the sources and destinations in the network. The cost of each edge in the bipartite graph (x_{cij}) is also the same as before and the capacity of these edges is infinite. An additional "master" source node S is added to the network and connected to each source node in the bipartite graph. These edges have zero cost, but a capacity equal to the bandwidth of the corresponding source. So, for example, the edge from S to source node s_1 has a capacity of b_{s_1} bits/s, where b_{s_1} is the injection bandwidth of the linecard at source 1. Similarly, the destination nodes in the bipartite graph are connected to a master sink node T. Node S can supply an infinite amount of flow, node T can sink an infinite amount of flow, and the remaining nodes do not source or sink traffic.

Because of the addition of the capacity constrained edges to (from) each source (destination), any flow in the graph is an admissible traffic pattern and any admissible traffic pattern is a valid flow. If we refer to the flow going from s_i to d_j in the graph as λ_{ij}, the total cost of a particular flow is

$$\sum_{i,j} \lambda_{ij} x_{cij},$$

which is our expression for channel load. Thus, by maximizing cost over all valid flows, we also maximize the channel load over all admissible traffic patterns. For a network with

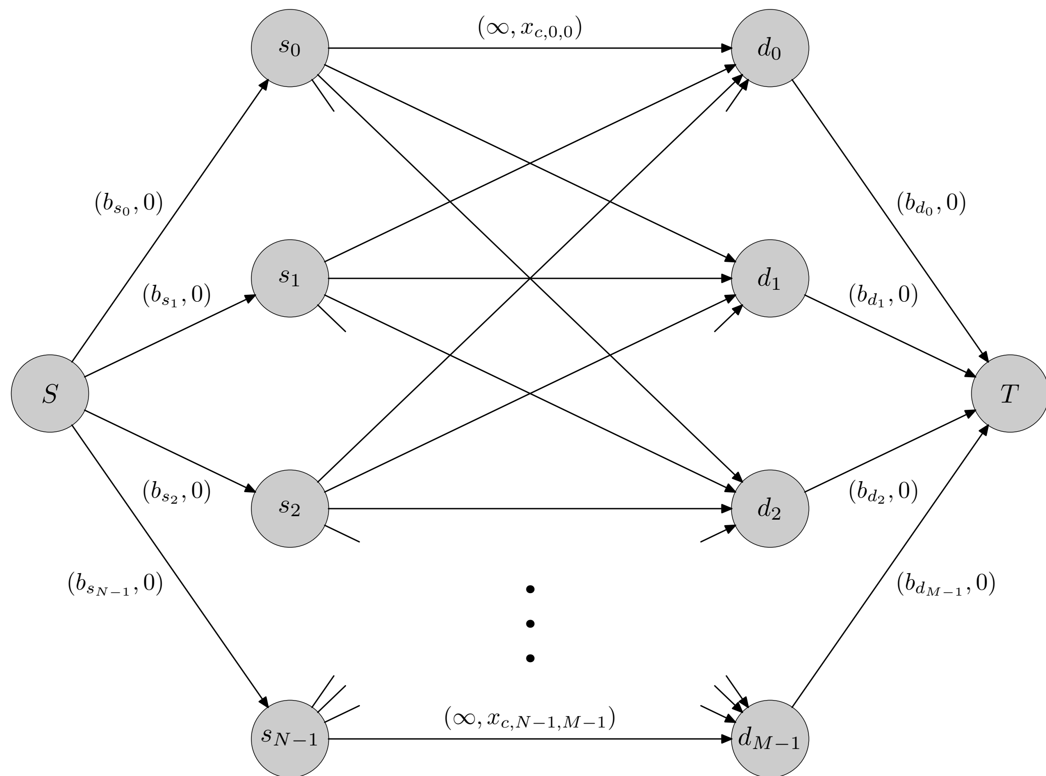


Figure 2.12: Maximum-cost flow problem used to solve heterogeneous cases of the worst-case traffic problem. Edges are labeled with a (capacity,cost) pair.

N sources and M destinations, a maximizing flow can be found in $O((N+M)NM)$ time.

There are obviously large similarities between this maximum-cost flow problem and the maximum-weight matching problem used in the homogeneous case. In fact, if the source and destination bandwidths were set to 1 bit/s and if $N = M$, the problems would be exactly the same. What might not be obvious when approaching the homogeneous case from this point of view is that flows corresponding to permutation traffic can always obtain the maximum cost. However, this result can be reached by viewing this maximum-cost flow problem in its linear programming form:

$$\begin{array}{lll} \text{maximize} & \sum_{i,j\in\mathcal{N}} \lambda_{ij} x_{cij} & \\ \text{subject to} & \sum_{j\in\mathcal{N}} \lambda_{i,j} \leq b_{s_i}, & \forall i \in \mathcal{N} \\ & \sum_{i\in\mathcal{N}} \lambda_{i,j} \leq b_{d_j}, & \forall j \in \mathcal{N} \\ & \lambda_{ij} \geq 0, & \forall i,j \in \mathcal{N} \end{array}$$

Since this optimization occurs over a bounded set of traffic patterns, the fundamental theorem of linear programming states that an extreme point of this set of traffic patterns can always realize the maximum. When the source and destination bandwidths are 1 bit/s, the set extreme points become the permutation traffic patterns by the result of Birkhoff [9] used in Section 2.3.1. For heterogeneous cases, the extreme points can still realize the maximum, but, in general, the extreme points would not necessarily have a structure that can be described as simply as the permutations.

Finally, as long as the possible traffic patterns form a convex set, maximizing the worst-case channel load can be cast as a general convex program, which can be solved efficiently (in polynomial time).

2.5.2 Symmetry optimizations

Both the maximum-weight matching algorithm (Section 2.3.3) and the maximum-cost flow algorithm (Section 2.5.1) run in polynomial time, but the large powers of N can still restrict the practical size of networks that can be analyzed. To reduce the complexity of finding a worst-case traffic pattern, symmetry in the network and routing algorithm can be used. In this section, we give an overview of how to reduce the number of channels considered for

the worst case and how to reduce the time required to find the edge weights. The application of these techniques to our previous example of DOR in a 2-dimensional torus network is then described.

Intuitively, if we are dealing with a completely symmetric network, each edge in the network is equivalent to any other edge and, thus, it should be sufficient to examine just one edge when searching for the worst-case throughput. Then, the worst-case throughput for any other edge should be the same by symmetry.

More formally, let Γ be a group whose elements define a mapping of the nodes of the network onto themselves. That is, for $g \in \Gamma$, $g : N \mapsto N$. The worst-case channel load of a network is said to be Γ-invariant if for every $g \in \Gamma$ the following conditions hold:

1. g is defines an automorphism of the network.

2. g preserves bandwidth of the sources and destinations: the bandwidth of source node s_i equals the bandwidth of the mapped source node $g(s_i)$, $b_{s_i} = b_{g(s_i)}$, and likewise for the destination nodes.

Also, denote the mapping of of a routing algorithm X using g as $g(X)$ where, for all sources i, destinations j, and channels $c = (u, v)$, $x_{cij} = x_{(g(u),g(v)),g(i),g(j)}$. Then, these conditions are sufficient to ensure that the worst-case channel load of algorithm X is unchanged by any mapping $g \in \Gamma$,

$$\max_{\Lambda \in \mathcal{T}} \gamma_c(\Lambda, X)/b_c = \max_{\Lambda \in \mathcal{T}} \gamma_{c'}(\Lambda, g(X))/b_{c'}, \tag{2.2}$$

where $\mathcal{T}$ is the set of admissible traffic patterns being considered and c' is the mapping c using g.

While the above conditions express symmetry in the network itself, there must also be symmetry in the routing algorithm to reduce the complexity of finding the worst-case throughput. Specifically, a particular algorithm is said to be Γ-invariant if for every $g \in \Gamma$,

$$X = g(X). \tag{2.3}$$

For a group Γ, if both the worst-channel load and routing algorithm are Γ-invariant, then combining (2.2) and (2.3) reveals that the worst-case channel load on a particular channel

$c = (u, v)$ is the same as the worst-case on the set of channels $\{g \in \Gamma | (g(u), g(v))\}$. Thus, the number of channels whose worst-case traffic must be found can be reduced by a factor $\leq |\Gamma|$. The reason the reduction is not always exactly $|\Gamma|$ is that some (non-identity) mappings may map c onto itself.

Symmetry can also be used to reduce the time required to compute the edges weights. When using the technique described in Section 2.4.1, the complexity of edge weight computation is largely determined by the number of source-destination pairs considered — for each source i and destination j, the technique finds x_{cij} for all $c \in \mathcal{C}$. By applying a mapping $g \in \Gamma$, the computed values of x_{cij} can be used to find the values of $x_{c',g(i),g(j)}$ for all $c' \in \mathcal{C}$ (routing algorithm invariance). As before, some mappings may map a source-destination pair onto itself, so the reduction in the number of pairs considered is at most $|\Gamma|$.

As an example of how symmetry can be used, consider the example of finding a worst case in a 2-dimensional torus network (k-ary 2-cube) that uses DOR. Define the group Γ_T using the generators for a horizontal shift (S_X), vertical shift (S_Y), horizontal flip (F_X), and vertical flip (F_Y). For any node in the network given by its radix-k address (x, y), these generators are defined as w

$$
\begin{aligned}
S_X(x, y) &= (x + 1, y) \qquad & S_Y(x, y) &= (x, y + 1) \\
F_X(x, y) &= (k - x, y) \qquad & F_Y(x, y) &= (x, k - y),
\end{aligned}
$$

where all operations are performed modulo k. This set of generators produce mappings that allow any horizontal (vertical) channel to be mapped to any other horizontal (vertical) channel in the torus. For example, mapping the channel from (0,0) to (1,0) to the channel from (3,3) to (2,3) can be achieved with a horizontal flip, three horizontal shifts, and three vertical shifts. The mappings also allow a particular source (destination) to be mapped to any other source (destination) in the network. The resulting group Γ_T preserves the conditions listed above and thus the worst-case channel load is Γ_T-invariant. Also, note that for this example adding a generator that swaps the x and y coordinates would have preserved worst-case invariance, but would not have preserved the routing algorithm invariance because DOR treats the horizontal and vertical dimensions differently.

When using these symmetries for finding the worst-case traffic, only one representative horizontal channel and one representative vertical channel need to be considered. This decreases the number of channels whose worst case must be found by a factor of $2N$ and reduces the complexity of finding the worst-case throughput to $O(N^3)$ in this example. When computing edge weights, a canonical source can be considered, such as the origin, and then only destinations whose minimal routes are in a particular quadrant of network, such as the $(+x, +y)$ quadrant, need to be considered. Then, by applying the symmetry mappings, this limited set of source-destination pairs can be mapped to any source-destination pair in the network. This reduces the number of pairs that need to be considered by a factor of approximately $4N$.

2.6 Related work

This approach compliments work in charactering of the worst case from a theoretical perspective. For example, Borodin and Hopcroft [12] showed that deterministic routing algorithms induce a channel load of $\Omega(\sqrt{N}/d^{3/2})$, where d is the maximum degree of any node in the network. Kaklamanis et al. [37] later strengthened this bound to $\Omega(\sqrt{N}/d)$ and the ideas were extended to bit-serial routing by Aiello et al. [3]. While these results hold for the class of all deterministic routing algorithms, they only provide lower bounds on performance. Rather, our approach is tailored for use by the network designer and provides an exact worst case for a specific problem instance. As illustrated in Section 2.4, finding the exact worst case significantly improves over the characterization given by existing ad-hoc testing methods such as those used by Nesson and Johnson [48], Bolding et al. [11], for example. As shown, these methods overestimated the throughput of ROMM by approximately 47% in our case.

2.7 Summary

An IP router can guarantee a particular throughput only if it can deliver packets at that rate over all admissible traffic patterns. In a distributed fabric, a routing algorithm determines how these traffic demands are spread over the fabric channels. Therefore, to meet this

throughput guarantee, the fabric's channels must not become overloaded, even for worst-case traffic. In this chapter, we showed that the worst-case traffic for an oblivious routing algorithm in a particular fabric can be found by solving a series of maximum-weight matching problems. This traffic pattern then determines the throughput that can be guaranteed by the fabric. The resulting algorithm runs in polynomial time, making exact worst-case analysis tractable.

Finally, computing the exact worst-case does have practical limits. While the underlying algorithms for solving the maximum-weight matching algorithms are polynomial in the network size, this dependence is cubic and grows quickly. Our implementation can analyze networks of around 500 nodes without exploiting symmetry. The symmetry optimizations described in Section 2.5.2 extend the approach to networks of several thousand nodes.

Chapter 3

Design of worst-case optimal routing algorithms

In the previous chapter, we showed how the worst-case throughput of an oblivious routing algorithm could be found by solving a series of maximum-cost flow problems. While this significantly improved the characterization of routing algorithms compared to existing ad-hoc approaches, it still leaves unanswered the more fundamental question of how to design oblivious routing algorithms that perform well in the worst case. As we show in this chapter, the maximum-cost flow formulation of the worst case is a convex function of the routing algorithm (Section 3.1). This, combined with the knowledge that the set of all oblivious routing algorithms is convex, allows us to formulate the design of good worst-case routing algorithms as convex optimization problems. *Globally optimal* solutions to these problems can be found efficiently (in polynomial time) giving us provably optimal oblivious routing algorithms. We present two practical approaches for finding solutions to these optimization problems in Section 3.2.

Once worst-case optimal routing algorithm design is cast as an optimization problem, a host of interesting applications are possible. Of course, routing algorithms can be customized to a particular topology. Also, the tradeoff between the worst-case and other design metrics can be explored. Specifically, we look at the tradeoff between worst-case throughput and both locality and average-case throughput in Section 3.3. These experiments reveal that, in torus networks, current routing algorithms give up too much locality

to achieve optimal worst case performance, leading to the development of two new routing algorithms. Also, for the same torus networks, a weak tradeoff between average- and worst-case throughput is demonstrated. Another set of experiments explores the performance advantages of worst-case optimal oblivious algorithms in irregular topologies. For example, in the comparison of 15 random topologies, optimal oblivious routing improves average worst-case throughput by 46.4% over Valiant's algorithm [71].

The one caveat in the routing algorithms developed through optimization is that they are specified in terms of real numbers. If these results are to be used to implement a routing algorithm in hardware, they must be converted to a finite representation (Section 3.4). A randomized rounding approach is developed and it is experimentally shown that approximating an oblivious routing algorithm's specification using integer multiples of a value ϵ decreases the worst-case throughput by a factor of $1 - O(\epsilon N)$. Two different hardware approaches for storing the finite representation of a routing algorithm are also presented and compared.

Finally, Section 3.5 discusses the key differences, in terms of both performance and implementation, between oblivious and adaptive routing algorithms.

3.1 Worst-case routing design as a flow problem

To formulate the design of routing algorithms for the worst case, we first develop a flow-based description of oblivious routing algorithms (Section 3.1.1). Then, this description is combined with the worst-case channel load as an objective function forming a convex optimization problem (Section 3.1.2).

3.1.1 Routing algorithm representation

A common way to formalize routing algorithms and the movement of data through a network is as a set of data flows through its topology. Early examples of this formulation include Fratta et al.'s study of routing algorithm design [29]; Bertsekas [6] contains an thorough overview of this and related formulations. We adopt the flow description, but point out one key difference between our approach and more common approaches. Here,

the flows through the network are not used to represent the actual flow of data through the network, but rather the flow of the probability of routing along the channels. This allows us to separate the description of the routing algorithm from the demands induced by a particular traffic pattern — a critical point, as we will ultimately optimize our routing algorithms over many traffic patterns simultaneously.

We have previously defined the probability of a packet using a particular channel c when routing from a source i to a destination j as x_{cij}. For now, we assume x_{cij} is a non-negative real number. Then, for a particular node k in the network, where $k \neq i$ and $k \neq j$, the probability of using any channel entering that node must equal the probability of using any channel leaving the node,

$$\sum_{\{l|(k,l)\in\mathcal{C}\}} x_{(k,l),i,j} - \sum_{\{l|(l,k)\in\mathcal{C}\}} x_{(l,k),i,j} = 0, \tag{3.1}$$

by the conservation of packets through that node.

However, (3.1) is nothing but a conservation of flow constraint through the node k. Considering that packets are injected at source nodes with probability one, there should be a surplus of one unit of flow reflected in the flow constraints. Likewise, destinations should have a deficit of one unit of flow. Combining the flow constraints for each node and source-destination pair with non-negativity constraints, any flows for which

$$\begin{aligned} \sum_{\{l|(k,l)\in\mathcal{C}\}} x_{(k,l),i,j} - \sum_{\{l|(l,k)\in\mathcal{C}\}} x_{(l,k),i,j} = [k=i] - [k=j] \qquad & \forall i,j,k \in \mathcal{N} \\ x_{cij} \geq 0, \qquad & \forall i,j \in \mathcal{N},\ c \in \mathcal{C} \end{aligned} \tag{3.2}$$

define a complete oblivious routing algorithm.[1] In the language of network optimization, these equations specify a multicommodity flow with one commodity per source-destination pair. Such flows can also be described as a probability distribution over the paths between each source-destination pair and this formulation is used later in Section 3.2.2.

Building on these multicommodity flow constraints, we can pose problems about designing optimal routing algorithms for a particular traffic pattern. So, for example, the

[1]The notation $[x = y]$ denotes a function that is 1 if $x = y$ and 0 otherwise, as adapted by Graham et al. [31] from Iverson's APL programming language [35].

throughput for the traffic pattern Λ can be maximized by solving

$$\text{minimize} \quad \max_{c \in C} \sum_{i,j \in \mathcal{N}} \lambda_{ij} x_{cij} / b_c, \tag{3.3}$$

where the routing algorithm X is subject to the constraints of (3.2). The optimal value of the optimization is a scaling factor that gives the reciprocal of the fraction of the original traffic pattern that can be supported by the network. For example, if the optimal value of (3.3) is 1.25, the network can support $1/1.25 = 80\%$ of the original traffic pattern before a channel becomes overloaded. In general, this form of a multicommodity optimization is known as a maximum concurrent flow problem (MCFP) and can be solved using linear programming methods as discussed by Ros Peran [50] or quickly approximated with primal-dual methods as introduced by Shahrokhi and Matula [60]. We will shortly extend the structure of this problem to solve our worst-case routing algorithm design problem (Section 3.1.2), but, first, we consider a particularly useful MCFP.

In the *uniform capacity problem* [26], or simply the *capacity problem*, the traffic pattern of (3.3) is defined to be

$$\lambda_{ij} = \frac{b_{s_i} b_{d_j}}{\max\left(\sum_{k \in \mathcal{N}} b_{s_k}, \sum_{k \in \mathcal{N}} b_{d_k}\right)}, \tag{3.4}$$

for all $i, j \in \mathcal{N}$. This traffic pattern is uniform in that each node spreads its traffic over each of the destinations in proportion to the destination bandwidths. In the homogeneous case (Section 2.3), where all injection and ejection bandwidths are equal to one, the pattern simplifies to $\lambda_{ij} = 1/N$. The fraction of the uniform pattern supported by a particular network is commonly used to normalize the performance of the network on other patterns. So, for example, if a network achieves 50% of capacity in the worst case, then the channel load induced by a worst-case pattern is twice that of the uniform pattern specified by (3.4). Since the uniform pattern is well-defined for any network, expressing throughputs as a fraction of capacity allows a meaningful comparison between networks with potentially different bandwidths. We will make use of this definition of uniform capacity in the later experimental sections.

3.1.2 Worst-case problem

The maximum concurrent flow problem (MCFP) introduced in the previous section closely resembles our goal of designing good worst-case routing algorithms, but rather than optimizing over the set of all admissible traffic patterns, the MCFP optimizes for a single traffic pattern. The logical extension to this is to augment the objective to include all the admissible traffic patterns,

$$\text{minimize} \quad \max_{\Lambda \in \mathcal{T}} \max_{c \in C} \sum_{i,j \in \mathcal{N}} \lambda_{ij} x_{cij} / b_c, \tag{3.5}$$

where $\mathcal{T}$ defines the set of admissible traffic patterns,

$$\begin{aligned} \mathcal{T} = \big\{ \Lambda \big| \textstyle\sum_j \lambda_{ij} \leq b_{s_i},\ \forall i \in \mathcal{N} \\ \textstyle\sum_i \lambda_{ij} \leq b_{d_j},\ \forall j \in \mathcal{N} \\ \lambda_{ij} \geq 0,\ \forall i, j \in \mathcal{N} \big\}. \end{aligned} \tag{3.6}$$

Before considering the tractability of even computing the objective (3.5), let us carefully consider its form. First, the sum and innermost maximum compute the fraction of a traffic pattern that can be supported by the network by summing to find the channel loads and then taking the maximum to find the bottleneck channel. The major change compared to the MCFP formulation is that the traffic pattern under consideration is no longer given as part of the problem setup, rather the set of admissible patterns $\mathcal{T}$ is specified. Then, the outermost maximum of (3.5) finds the worst-case traffic pattern amongst all the admissible traffic patterns. So the goal of the optimization is exactly what we want — a routing algorithm that minimizes the worst-case channel load, and thus maximizes throughput, over all admissible traffic patterns. We call this optimization problem the worst-case oblivious routing design problem (WCORDP).

Fortunately, despite its potentially intimidating form, the objective (3.5) can also be minimized efficiently. That is, for any network the *globally* best oblivious routing algorithm for minimizing worst-case channel load can be found. One key to a solution is the understanding that the objective is convex. This implies that any local minimum of the

objective function $f(X)$, where

$$f(X) = \max_{\Lambda \in \mathcal{T}} \max_{c \in C} \sum_{i,j \in \mathcal{N}} \lambda_{ij} x_{cij} / b_c$$

is also a global minimum. The convexity of f follows from the fact that it can be viewed as a point-wise maximum over linear functions, one for each possible traffic pattern-channel pair. A point-wise maximum of any number of convex functions is convex.

Stated another way, convexity guarantees that for any two valid routing algorithms X_1 and X_2 and a scalar $0 \leq \alpha \leq 1$,

$$f(\alpha X_1 + (1-\alpha)X_2) \leq \alpha f(X_1) + (1-\alpha) f(X_2). \tag{3.7}$$

So, any routing algorithm formed by interpolating between two routing algorithms always has a worst case that is at most equal to the interpolated worst cases of the two routing algorithms. It is also important that the interpolated algorithm itself is a valid algorithm, but it is not difficult to verify that this is indeed the case using the constraints of (3.2). More formally, the set of feasible routing algorithms form a convex set. This fact, combined with the convexity of f, means the task of designing worst-case optimal oblivious routing algorithms can be cast as a convex program.[2]

The final condition for any such convex program to be solved efficiently is that the objective can be evaluated easily (in time polynomial in the number of variables). Since we showed that the worst-case objective can be solved as a series of maximum-cost flow problems in Chapter 2, any WCORDP can be solved efficiently. We explore two specific solution methods in the following section.

3.2 Optimization approaches

As described in the previous sections, a worst-case oblivious routing design problem (WCORDP) can be cast as a convex program that, in theory, can be solved efficiently. However, as might be expected, the particular optimization method chosen has a large impact on the size of

[2] See Boyd and Vandenberghe [15] and Bertsekas et al. [8] for a detailed treatment of convex optimization.

problems that can be solved practically. In this section, several methods that take advantage of the structure of WCORDPs are described.

Section 3.2.1 shows that a WCORDP can be reformulated as a linear program (LP), allowing the use of specialized, off-the-shelf LP solvers. Alternatively, by taking advantage of the fact that the worst case for a particular routing algorithm can be found by solving maximum-cost flow problems, as described in Chapter 2, a subgradient method can provide fast solutions (Section 3.2.2). Finally, the symmetry of the underlying network can be used to constrain the search space of optimal routing algorithms and, thus, simplify the problem (Section 3.2.3).

3.2.1 Linear programming representation

A linear program (LP) is an optimization problem with a linear objective, linear constraints, and real-valued variables. Since linear functions are also convex, an LP is also a convex program — that is, linear programming is a special case of convex programming. The advantage in expressing our WCORDP as a simpler LP is that the solution methods for LPs are correspondingly simpler and faster. This allows us to extend the range of problem sizes that we can practically solve.

The first step in expressing a WCORDP as an LP is to take advantage of an observation from Section 2.5.1: for a particular routing algorithm, the maximum channel load can always be realized with an extreme point from the set of admissible traffic patterns $\mathcal{T}$ defined by (3.6). As a reminder, this was a direct consequence of the fundamental theorem of linear programming. Because the set of admissible traffic patterns describes a bounded polyhedron, there are a finite, although exponential, number of extreme points, $T_1, \ldots, T_E \in \mathcal{T}$. This allows us to rewrite our original convex objective (3.5) as a linear objective with linear constraints by introducing an inequality for each extreme point-channel pair as

$$\begin{array}{ll} \text{minimize} & w \\ \text{subject to} & \gamma_c(X, T_e)/b_c \leq w, \quad \forall c \in \mathcal{C},\ e = 1, \ldots, E, \end{array} \tag{3.8}$$

where the routing algorithm X is also subject to the constraints of (3.2). While this gives us a linear program, there are an exponential number of constraints associated with the

extreme points. However, these constraints can be simplified by first considering the dual of this linear program.

Substituting the definition of $\gamma_c(X, \Lambda)$, the Lagrange dual function corresponding the optimization problem defined by (3.8) and constrained by (3.2) is

$$g(D, Y, Z) = \inf_{w,X} \left\{ \sum_{i,j,k \in \mathcal{N}} d_{ijk}([k = j] - [k = i]) + w \left(1 - \sum_{c \in \mathcal{C}} \sum_{e=1}^{E} y_{ce} \right) + \right.$$
$$\left. \sum_{i,j \in \mathcal{N}} \sum_{(k,l) \in \mathcal{C}} x_{(k,l),i,j} \left(d_{ijk} - d_{ijl} - z_{(k,l),i,j} + \sum_{e=1}^{E} y_{ce} t_{eij} / b_c \right) \right\}, \quad (3.9)$$

where D is the dual variable associated with the conservation of flow constraints of (3.2) and has dimension N^3, Y is the variable associated with the channel-load constraints of (3.8) and has dimension CE, and Z is the dual variable associated with the flow non-negativity constraints and has dimension CN^2. While an explanation of Lagrangian duality is well beyond the scope of this thesis, the dual function g is a lower bound on the optimal value of the primal optimization (3.8) for any value of D and any values of $Y, Z \geq 0$.[3] Moreover, this bound is tight and, for some dual variable values, g is exactly equal to the optimal primal value.

The Lagrange dual (3.9) can be simplified by focusing on the third summation. The final term of this summation is a sum of extreme points weighted by Y. By again applying the fundamental theorem of linear programming, this sum can be replaced by a single, scaled traffic pattern Λ_c associated with each channel c,

$$\Lambda_c = \sum_{e=1}^{E} y_{ce} T_e, \qquad \phi_c = \sum_{e=1}^{E} y_{ce},$$

where ϕ_c is the scaling factor. The scaled traffic patterns are always admissible traffic patterns, $\Lambda_c / \phi_c \in \mathcal{T}$. Then, applying this simplification, the problem of finding the maximum value of the Lagrange dual, and thus the minimum value of the primal problem, can itself

[3] See Boyd and Vandenberghe [15] Chapter 5 for an introduction to duality.

be written as a linear program

$$
\begin{array}{lll}
\text{maximize} & \sum_{i,j \in \mathcal{N}} d_{ijj} - d_{iji} & \\
\text{subject to} & \lambda_{(k,l),i,j}/b_{(k,l)} \geq d_{ijl} - d_{ijk}, & \forall i,j \in \mathcal{N},\ c \in \mathcal{C}, \\
& \sum_{j \in \mathcal{N}} \lambda_{cij} \leq \phi_c b_{s_i}, & \forall i \in \mathcal{N},\ c \in \mathcal{C}, \\
& \sum_{i \in \mathcal{N}} \lambda_{cij} \leq \phi_c b_{d_j}, & \forall j \in \mathcal{N},\ c \in \mathcal{C}, \\
& \lambda_{cij} \geq 0, & \forall i,j \in \mathcal{N},\ c \in \mathcal{C}, \\
& \sum_{c \in \mathcal{C}} \phi_c = 1. &
\end{array}
\tag{3.10}
$$

Notice that (3.10) has a polynomial number of variables and constraints. Then, finding the dual of (3.10) recovers a simplified version of the primal problem

$$
\begin{array}{lll}
\text{minimize} & w & \\
\text{subject to} & x_{ijc} \leq v_{cj} - u_{ci} & \forall i,j \in \mathcal{N},\ c \in \mathcal{C} \\
& \sum_{j \in \mathcal{N}} v_{cj} b_{d_j} - \sum_{i \in \mathcal{N}} u_{ci} b_{s_i} = b_c w, & \forall c \in \mathcal{C},
\end{array}
\tag{3.11}
$$

where X is also subject to the constraints of (3.2). By introducing the extra variables U and V, each of dimension CN, the exponential number of constraints of (3.8) have been reduced to a polynomial number. This LP formulation of the WCORDP requires a total of $CN^2 + 2CN + 1$ variables and $2CN^2 + N^3 + C$ constraints.

It is worth noting that the reformulated primal problem (3.11) is closely related to the dual of the maximum-cost flow problem introduced in Section 2.5.1. In this context, the values U and V are commonly referred to as node potentials and minimizing the weighted sum of node potentials is equivalent to maximizing channel load. See Ahuja et al. [2], Section 9.4 for further information.

Also, the dual optimization problem (3.10) can be interpreted as optimizing many simultaneous shortest-path problems. Each channel c is assigned a "distance" λ_{cij} associated with the source-destination pair (i, j). Then, for each source i and destination j, every node k is labeled with the value d_{ijk} that defines the shortest distance to that node as measured by the channel lengths. To find the shortest distance from a node p to a node q, the absolute

distances are subtracted: $d_{ijq} - d_{ijp}$. The objective seeks to maximize the sum of shortest distances between each source-destination pair. Since the distances for each channel are determined by the traffic pattern Λ_c, the dual problem seeks to design traffic patterns that maximize the shortest-path distances between the source-destination pairs. Thus, any heuristic for choosing the traffic patterns gives a lower bound on the performance of any routing algorithm for the network.

3.2.2 Projected subgradient method

An alternative to reformulating the worst-case routing algorithm design problem as a linear program is to solve the convex program defined by (3.5) directly using a projected subgradient method. Subgradient methods [8] can be thought of as a generalization of gradient descent methods to non-smooth objective functions. The basics of the subgradient method are introduced in this section and a detailed implementation of the method is left for Appendix B.

For our case, the objective function of the optimization f is defined as

$$f(X) = \max_{\Lambda \in T} \max_{c \in C} \gamma_c(X, \Lambda)/b_c. \tag{3.12}$$

Then, the subgradient method repeats the following steps to minimize f:

1. Initialize with any valid routing algorithm $X^{(1)}$ and set $k \leftarrow 1$.
2. For any subgradient $G^{(k)}$ of f at $X^{(k)}$, update the routing algorithm as

   $$x_{cij}^{(k+1)} = P\left(x_{cij}^{(k)} - \alpha^{(k)} g_{cij}^{(k)}\right),$$

   where $\alpha^{(k)}$ is a step size and P is the Euclidean projection onto the set of feasible routing algorithms.
3. $k \leftarrow k + 1$; go to Step 2

Roughly speaking, the subgradient defines a direction of improvement based on the shape of the objective function at the current iterate of the routing algorithm X. Formally,

a subgradient of f at X is any G for which

$$f(Y) \geq f(X) + \sum_{c \in \mathcal{C}} \sum_{i,j \in \mathcal{N}} g_{cij}(y_{cij} - x_{cij})$$

holds for all Y. Geometrically, a subgradient defines a supporting hyperplane that passes through the point X. For any point Y on the side of the hyperplane where the inner-product of G and Y is positive, the value of $f(Y)$ is guaranteed to be larger than $f(X)$. Thus, the direction of descent in Step 2 is $-G$. In the special case where f is differentiable at X, the supporting hyperplane is equal to the tangent plane at X and the subgradient is equal to the gradient.

For the worst-case objective function defined by (3.12), any G for which

$$f(Y) \geq f(X) + \sum_{c \in \mathcal{C}} \sum_{i,j \in \mathcal{N}} g_{cij}(y_{cij} - x_{cij})$$

over all Y is a subgradient of f at X and, since f is convex, at least one such subgradient exists at all points. The shape of the worst-case objective function is determined by the elements of the maximum that are "active" for a particular value of the routing function X. That is, possible subgradients are determined by the traffic pattern-channel pairs that maximize the objective. It is easily shown[4] that any G for which

$$\sum_{c \in \mathcal{C}} \sum_{i,j \in \mathcal{N}} g_{cij} x_{cij} / b_c = \max_{\Lambda \in \mathcal{T}} \max_{c \in C} \gamma_c(X, \Lambda)/b_c$$

is a subgradient of f at X. For example, if for a particular routing algorithm X, if a worst-case traffic pattern Λ^* overloads channel c^*, defining G as

$$g_{cij} = \begin{cases} \lambda^*_{ij}, & \text{if } c = c^* \\ 0, & \text{otherwise,} \end{cases} \tag{3.13}$$

gives a subgradient of f at X. This result has a very simple interpretation when connected to its use in the subgradient method — the current routing algorithm X is improved by

[4]For example, see Bertsekas et al. [8] Proposition 4.5.1

shifting demand away from the current worst-case channel. Moreover, since the subgradient can be described in terms of a worst-case traffic pattern, the techniques described in Chapter 2 can be used to efficiently find a subgradient.

The two remaining aspects of the subgradient method are the selection of the step size α and the computation of the projection back to the set of feasible routing algorithms. First, there are many simple rules for selecting the step size, such as $\alpha^{(k)} = 1/k$, that ensure the method will converge to a global minimum of f. See Bertsekas et al. [8] Section 8.2, for more information. Second, while a naive formulation of the projection requires solving a quadratic minimization problem with similar complexity to the WCORDP, significantly more efficient approaches are possible.

For example, projection can be greatly simplified if a path-based description of the routing algorithm X is adopted. In this description, the algorithm is expressed in terms of paths between each source-destination pair. Denoting the probability of routing from i to j along path k as p_{kij}, the original objective function is preserved because

$$x_{cij} = \sum_{\substack{\text{paths } k \\ \text{from } i \text{ to } j \\ \text{containing } c}} p_{kij}. \tag{3.14}$$

Then, the routing algorithm constraints of (3.2) can be replaced by

$$\sum_{\substack{\text{paths } k \\ \text{from } i \text{ to } j}} p_{kij} = 1, \quad \forall i, j \in \mathcal{N}.$$

Computing the projection for each source-destination pair is independent and can be expressed as a quadratic program

$$\begin{array}{ll} \text{minimize} & ||p - q||_2 \\ \text{subject to} & \sum_{k=1}^{n} p_k = 1, \\ & p_k \geq 0, \qquad \forall k = 1, \ldots, n, \end{array} \tag{3.15}$$

where q is a vector of n path probabilities for a particular source-destination pair and the

variable p is the projection of these probabilities back onto the probability simplex. Fortunately, solving this optimization directly can be avoided by considering how the optimality conditions constrain the set of possible solutions.

Let p^* be a primal optimal point of (3.15). Then, the Karush-Kuhn-Tucker (KKT) conditions require the following to also be true:

$$\begin{aligned} \lambda_i^* \geq 0, &\quad i = 1, \ldots, n \\ \lambda_i^* p_i^* = 0, &\quad i = 1, \ldots, n \\ p_i^* = q_i - \lambda_i^* - \nu^*, &\quad i = 1, \ldots, n \end{aligned} \tag{3.16}$$

where λ^* and ν^* form a dual optimal point. Combining the second and third KKT conditions with the fact that p^* is non-negative, if $p_i^* > 0$ then $\lambda_i^* = 0$ and $p_i^* = q_i - \nu^*$. Otherwise, $p_i^* = 0$. Therefore p^* can be written as

$$p_i^* = \max(0, q_i - \nu^*), \quad i = 1, \ldots, n. \tag{3.17}$$

The fact that p^* can be expressed in terms of a single scalar variable ν^* greatly simplifies the computation and avoids the need to solve a quadratic program. Figure 3.1 shows C code for computing ν^* given the elements of q in sorted order. Conceptually, the code works by first setting ν^* to the largest element of q — at this point, p^* would be the zero vector. If ν^* is then decreased, the element of p^* corresponding to the largest element of q will increase. This continues until the value of ν^* reaches the second largest element of q. Then, decreasing ν^* further causes two elements of p^* to increase while the rest remain at zero. This process of decreasing ν^* should continue until the sum of all the p^* values exactly equals one. The rate of change in this sum is exactly equal to the number elements in q greater than the current value of ν^*. In the code of Figure 3.1, this rate of change is stored in the variable `i`. Each iteration of the `while` loop corresponds to ν^* falling below another element of q. Sorting dominates the complexity of computing p^* and the overall complexity of the projection is $O(n \log n)$. This procedure is closely related to the waterfilling algorithms that appear in applications such as allocating power to communications channels. See Boyd and Vandenberghe [15] Section 5.5.3, for an example.

```
1 : double FindNu( double *qs, int n )
2 : {
3 :   double sum = 0.0;
4 :   double nu  = qs[0];
4 :   int    i   = 1;
5 :
6 :   while ( ( i < n ) && ( sum < 1.0 ) ) {
7 :     sum = sum + ( (double)i )*( nu - qs[i] );
8 :     nu = qs[i++];
9 :   }
10:
11:   if ( sum > 1.0 ) { i--; }
12:
13:   return ( nu + ( sum - 1.0 ) / (double)i );
14: }
```

Figure 3.1: Code to compute ν^* for projection onto a probability simplex. The values of q are stored in sorted order, from largest to smallest in `qs` and the dimension of q is stored in `n`.

Combining the efficient projection algorithm with the fact that subgradients can be easily computed results in a fast projected subgradient approach to finding worst-case optimal routing algorithms. Additionally, for homogeneous networks, as described in Section 2.3, worst-case traffic patterns at each iteration can be permutation patterns and only N of the N^2 source-destination pairs need to be updated per iteration. A potential drawback of this method are the limitations of describing the routing algorithms using path probabilities. Since there are an exponential number of paths, only a subset can be considered in any practical optimization. While this offers an advantage when also considering the deadlock properties of the resulting routing algorithm (see Dally and Towles [26]), it also leaves the potentially difficult problem of selecting appropriate paths to the designer. An alternative to the path-based approach is described in Appendix B.

3.2.3 Symmetry optimization

Just as symmetry allowed simplifications in finding the worst case (Section 2.5.2), network symmetry can also be used to simplify the search for worst-case optimal routing algorithms. First, the definition of a worst-case invariant group from Section 2.5.2 is augmented to include channel symmetry. The worst-case throughput of a network is said to be Γ-invariant

if for every mapping $g \in \Gamma$ the following conditions hold:

1. g is defines an automorphism of the network.

2. g preserves bandwidth of the sources and destinations: the bandwidth of source node s_i equals the bandwidth of the mapped source node $g(s_i)$, $b_{s_i} = b_{g(s_i)}$, and likewise for the destination nodes.

3. g preserves bandwidth of the channels: the bandwidth of a channel $c = (u, v)$ equals the bandwidth of the mapped channel $c' = (g(u), g(v))$, $b_c = b_{c'}$.

These conditions are sufficient to ensure that

$$\max_{c \in \mathcal{C}} \max_{\Lambda \in \mathcal{T}} \gamma_c(\Lambda, X)/b_c = \max_{c \in \mathcal{C}} \max_{\Lambda \in \mathcal{T}} \gamma_c(\Lambda, g(X))/b_c,$$

for any $g \in \Gamma$. Moreover, $g(X)$ is valid routing algorithm because the automorphism condition preserves (3.2).

Then, for a particular routing algorithm X, let $\overline{X}$ be the average of X over its Γ-orbit,

$$\overline{X} = \frac{1}{|\Gamma|} \sum_{g \in \Gamma} g(X).$$

$\overline{X}$ is a valid routing algorithm by the convexity of the set of oblivious routing algorithms and, since Γ is a group, $\overline{X} = g(\overline{X})$ for any $g \in \Gamma$. Thus, $\overline{X}$ shares the same symmetry as the underlying network.

By applying the convexity of the worst case,

$$\max_{c \in \mathcal{C}} \max_{\Lambda \in \mathcal{T}} \gamma_c(\Lambda, \overline{X})/b_c \le \frac{1}{|\Gamma|} \sum_{g \in \Gamma} \max_{c \in \mathcal{C}} \max_{\Lambda \in \mathcal{T}} \gamma_c(\Lambda, g(X))/b_c = \max_{c \in \mathcal{C}} \max_{\Lambda \in \mathcal{T}} \gamma_c(\Lambda, X)/b_c.$$

So, routing algorithms that share the symmetry of the network have worst-case performance that is as least as good as their non-symmetric counterparts. This allows the search for optimal routing algorithms to be restricted to those that have the same symmetries as the network. Specifically, a set of variables can replaced by a single value

$$\overline{x}_{(u,v),i,j} = x_{(g_1(u),g_1(v)),g_1(i),g_1(j)} = \dots x_{(g_n(u),g_n(v)),g_n(i),g_n(j)},$$

for $g_1, \ldots, g_n \in \Gamma$. Also, because the routing algorithm being optimized is Γ-invariant once these reductions are applied, all the symmetry optimizations for finding the worst case described in Section 2.5.2 are applicable.

Revisiting the example of the 2-dimensional torus network (k-ary 2-cube) network from Section 2.5.2, routing variables only need to be tracked for routing from the origin to any destination whose minimal quadrant is increasing in both dimensions relative to the origin. Instead of considering all N^2 source-destination pairs, this reduces the number of pairs that need to be considered to approximately $N/4$.

3.3 Experiments

As we have shown, worst-case optimal routing algorithm design can be cast as a convex optimization problem and then solved, for example, via linear programming or subgradient methods. In this section, we apply these ideas to several different design problems. The tradeoff between worst-case throughput and both locality and average-case throughput is explored in Section 3.3.1 for torus networks. These experiments reveal that current routing algorithms give up too much locality to achieve optimal worst case performance, leading to the development of two new routing algorithms. In addition to increasing locality while maintaining optimal worst-case performance, these new algorithms achieve good average-case throughput. This demonstrates a weak tradeoff between average- and worst-case throughput in the torus.

In Section 3.3.2, worst-case optimal oblivious algorithms are compared to Valiant's [71] routing algorithm in irregular networks. While Valiant's algorithm performs as well as an optimal oblivious algorithm in many symmetric networks, the experiments show the same result does not hold for irregular networks. For example, in a comparison of 15 random topologies, optimal oblivious routing improves average worst-case throughput by 46.4% over Valiant's algorithm.

3.3.1 Design tradeoffs

In addition to simply designing routing algorithms for the worst case, our framework can be extended to optimize other convex properties of a routing algorithm. For example, both the average distance a packet travels, a measure of locality, and the average-case throughput are convex in the routing algorithm. In this section, we explore how these two measures of a routing algorithm's performance can be traded off against worst-case throughput.

To define the average distance a packet travels, we assume a uniform traffic pattern — each packet picks a particular destination from the N possible destinations with probability $1/N$. Then, the average distance (number of channels) a packet travels is found by summing over all source-destination pairs and paths. In terms of the routing algorithm, the average distance is

$$\frac{1}{N^2} \sum_{i,j \in \mathcal{N}} \sum_{\substack{\text{paths } k \\ \text{from } s \text{ to } d}} p_{kij} \cdot \mathbf{len}(i,j,k), \tag{3.18}$$

where p_{kij} is the probability of routing from i to j along path k and $\mathbf{len}(i,j,k)$ is the length of that path in channels. From (3.14) this expression can be rewritten in terms of the channel variables x_{cij} and, since the path lengths are just a constant, the average packet distance given by (3.18) is a linear function of the routing algorithm. Linear functions are convex, so average packet distance can be efficiently optimized.

It is certainly possible to design a routing algorithm that minimizes (3.18), but the result of this optimization is simply a routing algorithm that uses only shortest paths. A more interesting and informative optimization explores the tradeoff between average packet distance and worst-case throughput by incorporating (3.18) as a linear constraint in the worst-case optimization (3.5). Then, for a fixed average packet distance, the optimal worst case can be found.

For example, Figure 3.2 shows the result of this tradeoff problem for an 8-ary 2-cube network. Each point in the tradeoff space represents a separate routing algorithm that achieves a particular average packet distance and worst-case throughput. Ideal algorithms would sit in the lower-right corner of the graph. However, only certain algorithms are possible and the shaded area of the graph represents the set of feasible oblivious routing algorithms. Moreover, the guarantees of convex programming ensure that the unshaded

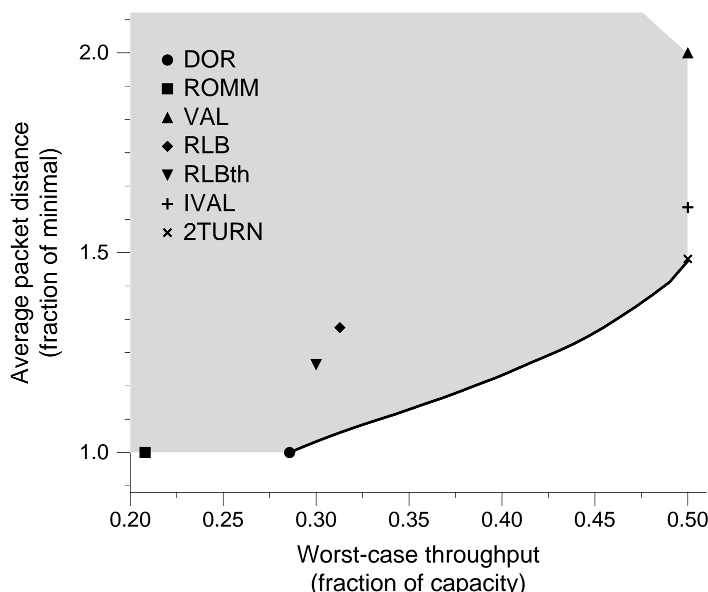


Figure 3.2: Tradeoff between average packet distance and worst-case throughput for a 8-ary 2-cube network. The set of feasible algorithms is shaded and the solid line indicates Pareto optimal algorithms.

areas are provably unobtainable. An important set of routing algorithms, shown as a dark line, are those that are Pareto optimal. Each Pareto optimal algorithm cannot be improved in either average packet distance without sacrificing worst-case throughput and vice versa. Along with the feasible algorithms, Figure 3.2 shows several existing algorithms that are summarized in Table 3.1.

The DOR and ROMM algorithms are minimal routing algorithms and therefore achieve the best possible locality. At the other extreme, Valiant's algorithm (VAL) achieves the best possible worst-case throughput, but sacrifices locality because it doubles the average path length. The two tradeoff algorithms, RLB and RLBth, achieve points between these two extremes. While DOR is at the lower-left end of the Pareto optimal curve, none of the existing algorithms lie near the upper-right of Pareto optimal curve. This observation leads to our development of the IVAL and 2TURN algorithms [70].

IVAL is an improvement over Valiant's algorithm that maintains its worst-case optimality while reducing average path length. As described in Table 3.1, Valiant's algorithm

Table 3.1: Summary of routing algorithms

DOR	Dimension-order routing [66]. Packets are routed minimally in the X dimension first, then in Y. If either direction is minimal in a dimension, routes are split evenly between both directions.
VAL	Valiant's routing algorithm [71]. In the first phase, packets are routed from the source to a randomly chosen intermediate node using a minimal routing algorithm (e.g. DOR). The second phase routes minimally from the intermediate to the destination.
ROMM	ROMM [48]. A two-phase algorithm that uses DOR for both phases, like Valiant's, but routes are kept minimal by always choosing the intermediate from the minimal quadrant.
RLB	Randomized local balance [63]. Another two-phase algorithm where the intermediate is chosen so that minimal routing occurs in the X dimension with probability $(k - \Delta_X)/k$, where Δ_X is the minimum distance in X that must be traveled and minimal routing occurs in Y with probability $(k - \Delta_Y)/k$. DOR is used for both phases.
RLBth	RLB threshold [63]. A modification of RLB where a packet is always routed minimally in X if $\Delta_X < k/4$ and minimally in Y if $\Delta_Y < k/4$.

routes packets using two phases of randomized routing. By randomly picking the intermediate destination node of the first phase, load is balanced over the channels of the network. However, in this strict implementation, many paths revisit nodes, forming loops. Obviously, eliminating these loops cannot degrade the performance of the algorithm because the load placed on the channels is only reduced. IVAL takes this idea further by making the observation that the two phases can use different routing algorithms to try to increase the occurrence of loops. As long as both phases route minimally, the worst-case performance is maintained. Specifically, IVAL uses DOR (X first, then Y) for the first phase and then uses DOR with the dimension order reversed (Y first, then X) for the second phase. This creates large loops as once a packet reaches the intermediate node, it often doubles back over the same channels on its way to the final destination. For an 8-ary 2-cube network, IVAL reduces average path length to about 1.61 times minimal — an almost 20% reduction over the average path length of VAL.

Although IVAL has a simple closed-form (algorithmic) description and improves significantly over Valiant's algorithm, its average path length is still about 9.1% above the optimal point of just below 1.48 times minimal. There may exist other simple algorithms that lie within this gap, however the gap can also be reduced if the designer is willing to abandon a purely closed-form description of the routing algorithm. We introduce such an algorithm, called 2TURN, that does just that.

Instead of requiring that the routing algorithm have a closed-form description, the 2TURN algorithm only uses a closed-form description of the possible paths a packet may take through the network. As its name implies, 2TURN allows any path through the network which contains at most two turns. A turn is defined as any change from routing in one dimension to the other. Also, "u-turns" or changes of direction within dimensions are disallowed in the 2TURN algorithm. Since every path in IVAL also has at most two turns, 2TURN contains all the paths considered by IVAL. 2TURN can also use paths not available to IVAL. For example, IVAL always routes minimally after its final turn, but 2TURN has the option to route non-minimally.

The constraints on path choice imposed by the 2TURN algorithm can be easily incorporated into the path-based formulation of the routing algorithm optimization problem. The result of the optimization is the probability with which each of the possible paths should

be taken. Since the paths of 2TURN are a superset of those of IVAL, 2TURN can match IVAL's worst-case performance. At the same time, the average path length of 2TURN is reduced to approximately 1.48 times minimal, only 0.36% more than the Pareto optimal algorithm (Figure 3.2). The main advantage of adopting the closed-form paths of 2TURN is one of implementation. Since our networks have finite resources, the dependencies between these resources need to be carefully considered to avoid deadlocks [25]. The possible routes a packet takes determines these dependencies, so having a simple description of the paths allows an equally simple approach to deadlock avoidance.[5]

Now that we have routing algorithms near both ends of the Pareto optimal curve, convexity can be taken advantage of to create algorithms between these two points. Oblivious routing algorithms form a convex set and, by definition, any interpolation between routing algorithms is itself a valid routing algorithm. For example, consider the routing algorithms X and Y and an interpolation factor $0 \leq \alpha \leq 1$. Then a new routing algorithm Z can be formed by interpolating between X and Y,

$$z_{cij} = \alpha x_{cij} + (1 - \alpha) y_{cij}.$$

The interpolation factor α controls the relative influence of X and Y on Z. Intuitively, as α sweeps from one to zero, the properties of Z transition from those of X to those of Y. Quantitatively, if we define the average path length of a routing algorithm X as $L(X)$, then it is easily verified that

$$L(Z) = \alpha L(X) + (1 - \alpha) L(Y)$$

by using the linearity of path length. Also, if the worst case of a routing channel load of an algorithm X is defined as $W(X)$, then

$$W(Z) \leq \alpha W(X) + (1 - \alpha) W(Y). \quad (3.19)$$

This inequality is simply a restatement of the convexity of the worst case from (3.7).

For the 8-ary 2-cube, the set of routing algorithms produced by interpolating between

[5] 2TURN can be made deadlock free with 4 virtual channels (VCs). Packets start in VC set zero and the VC set is incremented after a turn from the Y to X dimension. Each VC set has 2 individual VCs to avoid intra-dimension deadlock.

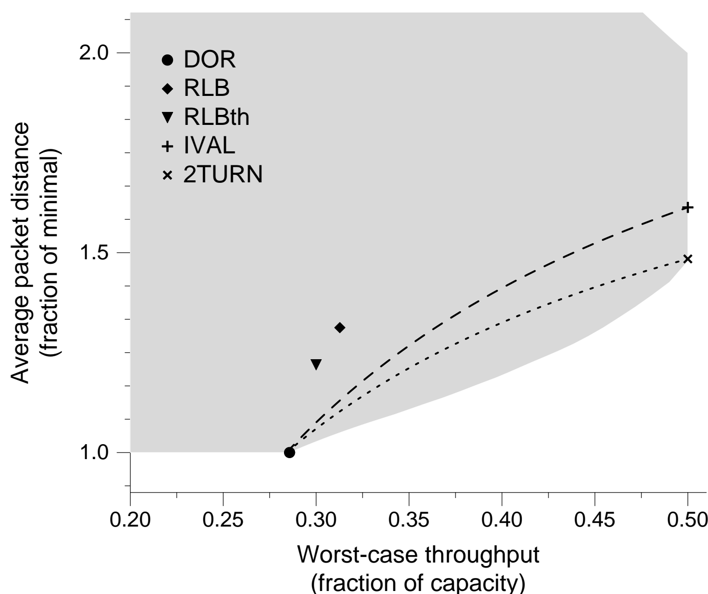


Figure 3.3: Routing algorithms produced by interpolation. The dashed line shows interpolation between DOR and IVAL and the dotted line shows interpolation between DOR and 2TURN.

DOR and IVAL and between DOR and 2TURN is shown in Figure 3.3. Each point in the curve corresponds to a different value of α used in the interpolation. For this example, it also happens that the worst-case throughput of the interpolated routing algorithms is exactly equal to the lower bound of (3.19).[6]

The interpolated algorithms between DOR and IVAL are at most 17% above the optimal locality with the maximum percent difference occurring about 65% of the way between DOR and IVAL. At the same worst-case throughput as RLB, the interpolated algorithm gives a roughly 14% reduction in path length. Also, the reduction in path length over RLBth is about 12%. Interpolating between DOR and 2TURN improves performance further, with the set of interpolated algorithms at most 10% above the optimal locality and offering a 19% and 15% reduction in path length over RLB and RLBth, respectively. Also, the algorithms produced by interpolation are simple to implement. For example, to interpolate between IVAL and DOR a packet is routed using IVAL with probability α and using DOR with

[6]Since DOR and IVAL and DOR and 2TURN share a worst-case traffic pattern, the actual worst-case throughput of the interpolated routing algorithms are equal to the bound. This is always true when interpolating between routing algorithms that share a worst-case traffic pattern.

probability $1 - \alpha$.

Another interesting performance metric that can be approximated by a convex function is the average-case throughput of a routing algorithm. First, the average channel load is a convex function of the routing algorithm and is proportional to

$$\int_{\Lambda \in \mathcal{T}} \max_{c \in \mathcal{C}} \sum_{i,j \in \mathcal{N}} \lambda_{cij} x_{cij} / b_c. \tag{3.20}$$

The true average would be divided by the volume of the traffic set, but we have left out this constant factor for clarity. Then, we approximate the average-case throughput as simply the reciprocal of (3.20).

Figure 3.4 shows the tradeoff between average packet distance and average-case throughput in the 8-ary 2-cube. In this experiment, (3.20) was evaluated by numerically integrating over a sample of the set of all traffic patterns. Unlike the worst-case tradeoff, the existing routing algorithms (Table 3.1) generally lie far from the Pareto optimal set. Computing the average throughput of the two new routing algorithms developed for the worst-case, IVAL and 2TURN, reveals that they are both much closer to the maximum average-case throughput: IVAL is within 8.4% and 2TURN is within 6.4%. This also demonstrates that there is a weak tradeoff between worst-case and average-case throughput in the 8-ary 2-cube and that routing algorithms can be designed to have both good worst-case and good average-case throughput.

Adapting the approach used to develop the 2TURN algorithm for the worst case, a similar algorithm 2TURNA can be designed for the average case. 2TURNA allows all paths with at most two turns and the probabilities for each path are found by first optimizing for the average-case throughput, then for maximizing locality. The resulting algorithm's performance is also plotted in Figure 3.4. As shown, 2TURNA has an average-case throughput within 4.6% of the maximum of approximately 62.8% of capacity. 2TURNA also increases locality over IVAL and 2TURN, sitting roughly 16% above the optimal tradeoff curve in terms of average path length.

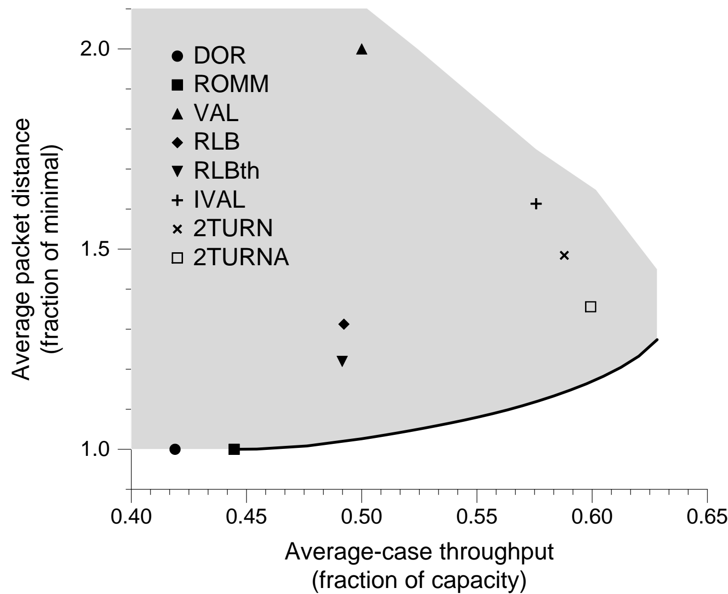


Figure 3.4: Tradeoff between average packet distance and average-case throughput for a 8-ary 2-cube network. The set of feasible algorithms is shaded and the solid line indicates Pareto optimal algorithms.

3.3.2 Comparison to Valiant's algorithm in faulty and irregular topologies

As demonstrated in the previous section, designing routing algorithms through optimization offers a significant improvement in locality for a given worst-case throughput in the two-dimensional torus network. However, in terms of just worst-case throughput, Valiant's algorithm matches the best performance of any oblivious algorithm. It is easily shown that this result extends to any torus network and many other symmetric networks. (See Dally and Towles [26] Section 9.1.1 or Singh et al. [61], for example.) However, in this section, we show that this result does not hold in general, and quantify the performance difference between Valiant's algorithm and an optimal oblivious algorithm in several networks.

The first set of experiments compares the worst-case throughput of Valiant's algorithm to an optimal oblivious algorithm on a 6-ring (one-dimensional torus) network with a variable number of faulty channels. It is assumed that the routing algorithm cannot deliver

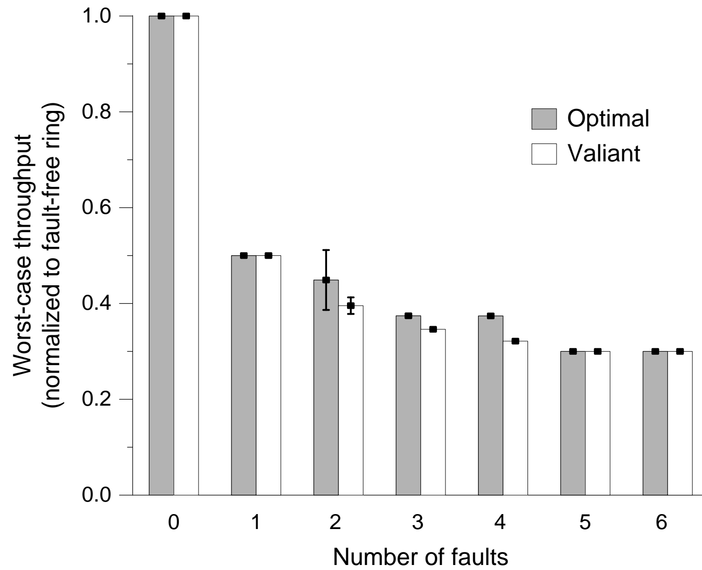


Figure 3.5: The worst-case throughput of both an optimal oblivious algorithm and Valiant's randomized algorithm in a 6-ring (one-dimensional torus) with a variable number of faulty channels. All throughputs are normalized to the fault-free case and only channel fault patterns that leave the network connected are included in the results. For a given number of faults, there are many possible fault patterns that leave the network connected and the bars show the average throughput over these patterns and the error bars show one standard deviation.

any traffic over a faulty channel and only channel fault patterns that leave the network connected are considered. Also, in the faulty cases, the throughput of Valiant's algorithm is always half of the network's capacity — the throughput achieved when the channel load is exactly twice that of uniform traffic (Section 3.1.1).

As shown in Figure 3.5, both Valiant's algorithm and an optimal algorithm have the same performance for the symmetric, fault-free network. In the case of a single fault, the performance of the algorithms again remains the same. Performance diverges for the two-, three-, and four-fault cases, where the optimal algorithm offers a 13.7%, 8.1%, and 16.5% improvement in throughput, respectively. For the five- and six-fault cases, the bidirectional ring has degenerated to essentially a unidirectional ring in which there is a single path between each source and destination and, thus, no variation between the different routing algorithms.

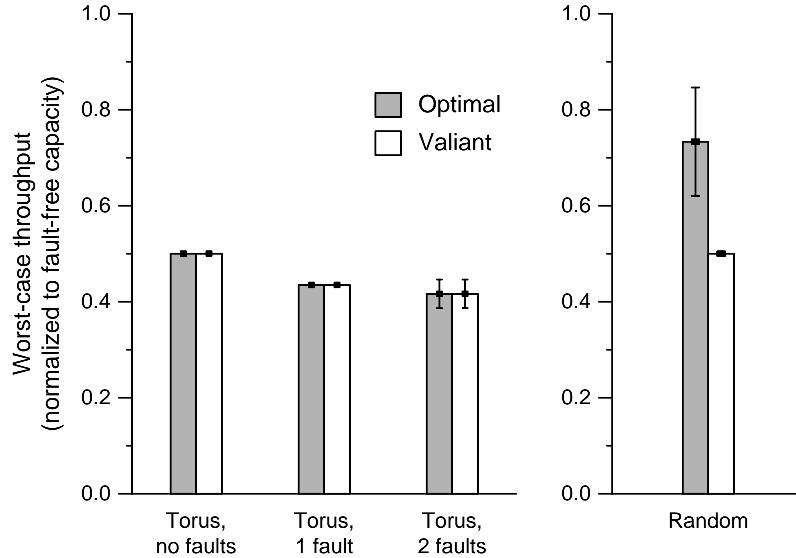


Figure 3.6: The worst-case throughput of both an optimal oblivious algorithm and Valiant's randomized algorithm on faulty torus (left) and random networks (right). All throughputs are normalized to the fault-free capacity of the corresponding network. As in Figure 3.5, the bars show the average case over a sampling of fault patterns and random topologies and the error bars show one standard deviation.

This analysis is extended to two-dimensional torus networks in the left graph of Figure 3.6. In this case, 4-ary 2-cube (torus) networks with zero, one, and two faults are considered. All throughputs on the torus are normalized to the capacity of the non-faulty case. As shown, the performance of Valiant's algorithm and an optimal oblivious algorithm are exactly the same in all cases. Also, the results show that average throughput is reduced by 13.2% when one channel is faulty and 16.8% when a second channel fails in a 4-ary 2-cube.

The lack a performance difference for the one- and two- fault cases in the 4-ary 2-cube combined with the results in the ring could indicate that the worst-case performance gap between Valiant's algorithm and an optimal oblivious algorithm is determined by the "irregularity" of the underlying network — the more irregular a network's topology, the larger the advantage to using an optimal oblivious algorithm. This informal observation is supported further by comparing the two algorithms in random topologies.

A total of 15 random topologies with 16 nodes and 48 channels were constructed using a

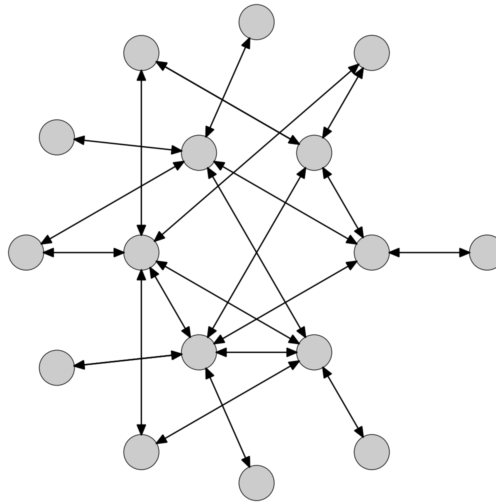


Figure 3.7: A random network with 16 nodes and 48 channels created using linear preferential attachment.

two-step process. First, each node was randomly connected to another node in the network. Then, additional channels were added using linear preferential attachment. That is, the probability of selecting a node as an endpoint of a new channel was proportional to the number of channels currently incident to that node.[7] In this rich-get-richer model, the distribution of node degrees tends to a power law, which has been observed experimentally in many real-world networks. An example of a network generated using this approach is shown in Figure 3.7. See Mitzenmacher's survey [45] for more information on power laws and preferential attachment.

The performance on these random networks is also shown in the right graph of Figure 3.6. Here the advantage of using an optimal oblivious algorithm is very pronounced, offering a 46.4% improvement in throughput over Valiant's algorithm. These results also lend more support to out observation that optimal oblivious algorithms offer the most improvement in irregular networks.

[7]We always added channels in bidirectional pairs, disallowed self loops, and discarded disconnected topologies.

3.4 Implementation

A fundamental assumption in our discussion of routing algorithm design up to this point has been that the routing variables (x_{cij}) can be represented as real numbers. When using the optimization approach as a guide for algorithm design, as with the IVAL algorithm discussed in Section 3.3.1, this assumption has no impact. However, if the results of the optimization are used to directly define the algorithm, a finite representation will need to be used.

In this section, we first examine the properties of several routing algorithms designed using the techniques described in this chapter and the cost of moving these algorithms to a finite representation (Section 3.4.1). A randomized rounding approach is developed and it is experimentally shown that approximating the routing probabilities of an oblivious routing algorithm as an integer multiple of a value ϵ decreases the worst-case throughput by a factor of $1 - O(\epsilon N)$.

Two different hardware approaches for storing the representation of a routing algorithm as a source route are presented in Section 3.4.2. A direct SRAM lookup approach is simple, but is expensive for storing a small number of paths precisely. An alternative ternary-CAM (TCAM) approach that adds an extra stage to route lookup that covers the drawback of the SRAM approach is also presented along with an area comparison of both approaches.

3.4.1 Routing algorithm approximation

When designing routing algorithms for hardware implementation using optimization techniques, the ideal approach would be to add integral constraints to the optimization problem so that the resulting routing variables were always an integer multiple of a granularity ϵ. Unfortunately, affixing such constraints creates an integer program and integer programs are generally NP-hard to solve exactly. There has been a significant amount of work in approximating the solution to these optimization problems. Specifically, the randomized rounding technique of Raghavan and Thompson [55] and later improvements by Srinivasan [64] for packing problems. Both of these techniques work on a class of optimization problems, the most relevant of which is the integer packing problem. The WCORDP can

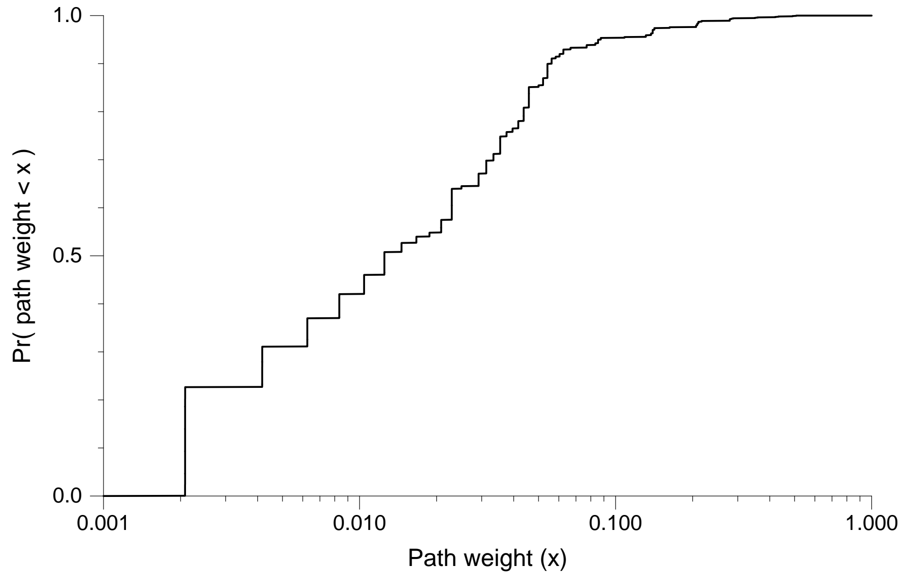


Figure 3.8: Cumulative distribution of path weights for a routing algorithm that minimizes path length while maintaining optimal worst-case throughput in a 8-ary 2-cube network.

be cast as a packing problem by using the formulation of (3.8), but this is extremely inefficient — the problem size is exponential in the size of the network. Rather, we adopt a pragmatic approach which starts with solutions produced by either of the methods from Section 3.2 and then applies a randomized rounding step. Before describing the details of this approach and its performance, it is informative to see an example of the results produced by the optimization approach prior to rounding.

Figure 3.8 shows the cumulative distribution of path weights (the p_{kij} variables used in Section 3.2.2) for a worst-case optimal routing algorithm for the 8-ary 2-cube network ($N = 64$). For this particular algorithm, each source-destination pair uses roughly 33.5 paths on average. In terms of approximation, it is important to note the substantial number of paths that have low weights — more than 20% of the paths have a weight less than 0.0021, meaning they are used less than 0.2% of the time. Also, very few paths are used more than 10% of the time. Although this distribution is only an example, it is representative of the routing algorithms generally produced when optimizing for the worst case — load is spread finely over a large number of paths.

To approximate a particular algorithm so that the paths weights are all integer multiples of a granularity ϵ, we adopt a randomized rounding [55] approach. First, the routing algorithm is expressed using path weights (p_{kij}). Edge weights (x_{cij}) can be converted to paths weights in polynomial time as described in Section 3.5 of Ahuja et al. [2]. Then, approximated path weights are formed by rounding each p_{kij} down to the nearest multiple of ϵ. The difference in the rounded and original path weights is stored as a remainder r_{kij}. Additional weight is distributed to the paths in proportion to their remainder until the sum of approximated weights is one for each source-destination pair. (See Algorithm 1.)

Algorithm 1 Routing algorithm approximation

1. *Extract paths.* Either use the given p_{kij} values, or use Dijkstra's algorithm to extract the path weights (largest weight paths first) from the x_{cij} values.
2. *Loop over s-d pairs.* For each $i, j \in \mathcal{N}$,
   (a) *Round paths down.* For all paths k from i to j,
   $$\tilde{p}_{kij} \leftarrow \epsilon \left\lfloor \frac{p_{kij}}{\epsilon} \right\rfloor,$$
   $$r_{kij} \leftarrow p_{kij} - \tilde{p}_{kij}.$$
   (b) *Randomized rounding.* While $\sum_k \tilde{p}_{kij} < 1$ randomly pick a path l with probability $r_{lij}/\sum_k r_{kij}$,
   $$\tilde{p}_{lij} \leftarrow \tilde{p}_{lij} + \epsilon.$$

As is the case with the randomized rounding approach, the expected value of an approximated path weight is equal to its corresponding original path weight, $E[\tilde{p}_{kij}] = p_{kij}$, and, by linearity of expectation, the same holds for the edge weights. Of course, the important aspect of this algorithm is how it affects the worst case. While exact analysis is difficult, qualitatively, worst-case channel load tends to increase additively by $O(\epsilon N)$. So, to have an approximation error that is constant with the number of nodes ϵ should be $O(1/N)$.

Intuitively, the $O(\epsilon N)$ error is because the standard deviation in edge weight in the approximated algorithm is proportional to ϵ and many of the edge weights in the corresponding maximum-cost flow will increase by at least ϵ. Since a worst-case optimal routing algorithm tends to balance load, there are generally many traffic patterns that cause

worst-case or near worst-case load and, with high probability, one of these patterns will largely use the edges that have been increased by ϵ in the maximum-cost flow graph. The same argument holds for each channel and the variation from channel-to-channel tends to be small. So, the number of channels has negligible impact on the increase in the worst case. It follows that the variation between particular instances of the algorithm (one set of random values) is small and little benefit is seen from running the algorithm multiple times and taking the approximation with the least error.

An example of the performance of the randomized routing algorithm is shown in Figure 3.9. For small ϵ, our previous observation about the increase in channel load due to approximation implies the throughput should be reduced by a multiplicative factor of $(1 - O(\epsilon N))$. For $1/\epsilon$ between 64 and 1024 this relationship holds at most packet distances. For the two algorithms in the upper-right corner of the feasible space, the output of the optimization produced edge weights that were all multiples of $1/64$, so the approximation had no effect until $\epsilon \geq 1/32$. The fact that the optimization produced such regular values is attributed to the symmetry of this particular network. For all $\epsilon \geq 1/32$, the approximations performed poorly.

While this randomized rounding approach appears to require a path granularity of $\epsilon = O(N)$ to maintain constant error, it may be possible to reduce the granularity in many networks. To see how this is possible, consider an optimal oblivious routing algorithm X^*. One way to view a constant error approximation of X^* is as a sum of deterministic routing algorithms D each weighted by ϵ. If $f(x)$ is the worst-case channel load of a routing algorithm X, then, by convexity

$$f(X^*) + C = f(\epsilon \sum_i D_i) \geq \epsilon f(D^*),$$

where D^* is an optimal deterministic routing algorithm and $C \geq 0$ is the constant error. Then,

$$\epsilon \leq \frac{f(X^*) + C}{f(D^*)}.$$

So, the maximum epsilon is determined by the ratio of the optimal oblivious algorithm's

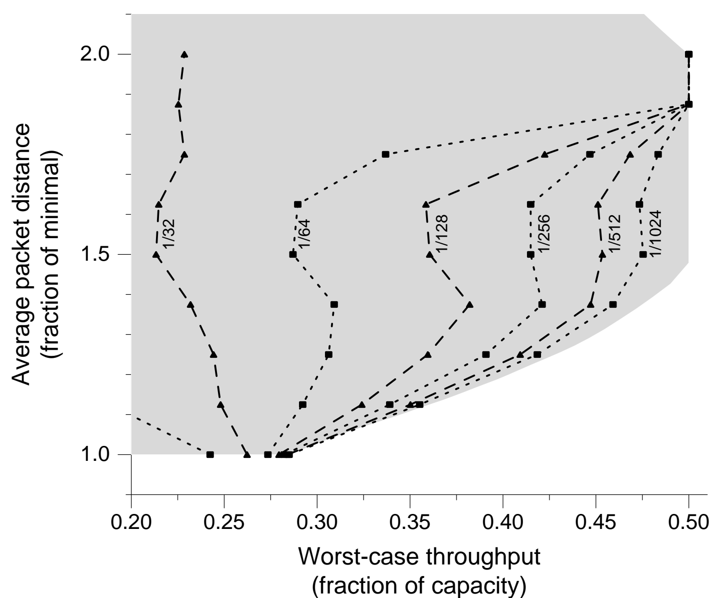


Figure 3.9: Performance of approximated routing algorithms for a 8-ary 2-cube network. Each point is an approximation of the routing algorithm with the same average path length and optimal worst-case (all approximations lie on the same horizontal line as their original) and every sequence of points joined by lines represents a particular value of ϵ (whose value is indicated by a label). The gray region represents the set of feasible oblivious routing algorithms.

channel load to optimal deterministic algorithm's channel load. For any network, the channel load of a deterministic algorithm is at least $f(D^*) = \Omega(\sqrt{N})$ by the result of Borodin and Hopcroft [12] and this bound can be met in many common networks, such as the torus, mesh, and butterfly. By a later result by Borodin et al. [13], the lower bound on the channel load of an oblivious routing algorithm is $f(X^*) = \Omega(\log N)$. Combining these results, it could be possible for an ϵ as large as $O(\log N/\sqrt{N})$ to give a constant approximation error for a general network. For some networks, such as the torus and mesh, the best oblivious algorithms have $f(X^*) = \Omega(\sqrt{N})$, which allows the possibility of a constant number of paths giving a constant approximation error regardless of the size of the network. It is left as an open problem as to whether these bounds can be achieved or countered.

Another interesting approach to the problem of reducing the granularity of oblivious routing algorithms is presented by Krizanc et al. [41] for the closely related problem of batch routing. In this work, the authors take advantage of the fact that a very small set of deterministic routing algorithms can be selected so that, with high probability, a particular traffic pattern routed with one of these algorithms will have a low channel load. This essentially allows the route granularity to be reduced to a constant. Some of the broadcast aspects of the algorithm may be difficult to adapt efficiently to our problem of continuous (as opposed to batch) routing, but the general approach is certainly noteworthy.

3.4.2 Hardware organization

Once we have a finite precision representation of our routing algorithm, we are still faced with the task of randomly selecting a route based on the given probabilities. One approach for computing routes is *source routing*, where a packet's route is completely determined at injection. This information is then attached to the packet and used at the intermediate hops to direct the packet to its destination. The mechanics of source routing within the fabric are well understood (see Dally and Towles [26], for example), so we focus on the initial route look up in this section. An advantage of this approach is that it allows routing algorithms to be loaded into memory at run time, so retuning of the algorithm, due to a fault in the network, for example, is possible.

Two different hardware organizations for route lookup are shown in Figure 3.10. For

both approaches, route lookup begins by creating a destination-random value (DRV) pair. The random value can be created by a pseudo-random number generator and, for a path weight granularity of ϵ (Section 3.4.1), contains $\log_2 \epsilon^{-1}$ bits. This pair is enough to access all the possible paths from a particular source and the same route lookup hardware is replicated at each source.

In the first approach (Figure 3.10[a]), the DRV pair is used to directly access an SRAM that stores path indices. Then, for example, if path p to destination 10 is used with probability $1/8$ and the granularity is $\epsilon = 1/64$, then 8 entries are placed in the SRAM ($(1/8)/(1/64) = 8$) all of which store the index of path p. So, the total number of entries in the SRAM is equal to the number of destinations over the path granularity or N/ϵ. Each entry stores a path index and, if there are an average of P paths per source-destination pair, each path index is approximately $\log_2 NP$ bits wide. Storing indices in the table reduces replication for paths that are used more frequently than the minimum granularity — only the path indices need to be replicated instead of the entire path description. Moreover, storing indices allows paths that are subsequences of other paths to be overlapped in path storage as illustrated in Figure 3.10(c). Dally and Towles [26], Section 11.1.1, also covers this overlapped storage technique.

Although storing path indices reduces the problem of replication in the SRAM-based approach, it still seems wasteful — especially in the case where a relatively small number of paths need to be stored with a high precision (low granularity). To address this, a hardware organization based around a ternary content addressable memory (TCAM) is introduced (Figure 3.10[b]). Unlike an SRAM, a TCAM is an associative structure, which is accessed by providing a tag that is matched against tags stored in each of entry of the TCAM. The output of a lookup is the address (entry number) of the matching tag(s). In the case of a ternary CAM, each bit of the stored tags can take three values: zero (0), one (1), or don't care (X).

To see the advantage of a TCAM in storing path weights, consider an example where the granularity is $\epsilon = 1/64$ and a path of weight 28ϵ needs to be stored. In the SRAM organization, that path's index would simply be replicated 28 times. However, only three entries are needed in the TCAM as shown in Table 3.2. By using don't care bits, a single entry can match against many DRV pairs. For the example, the probability of taking the

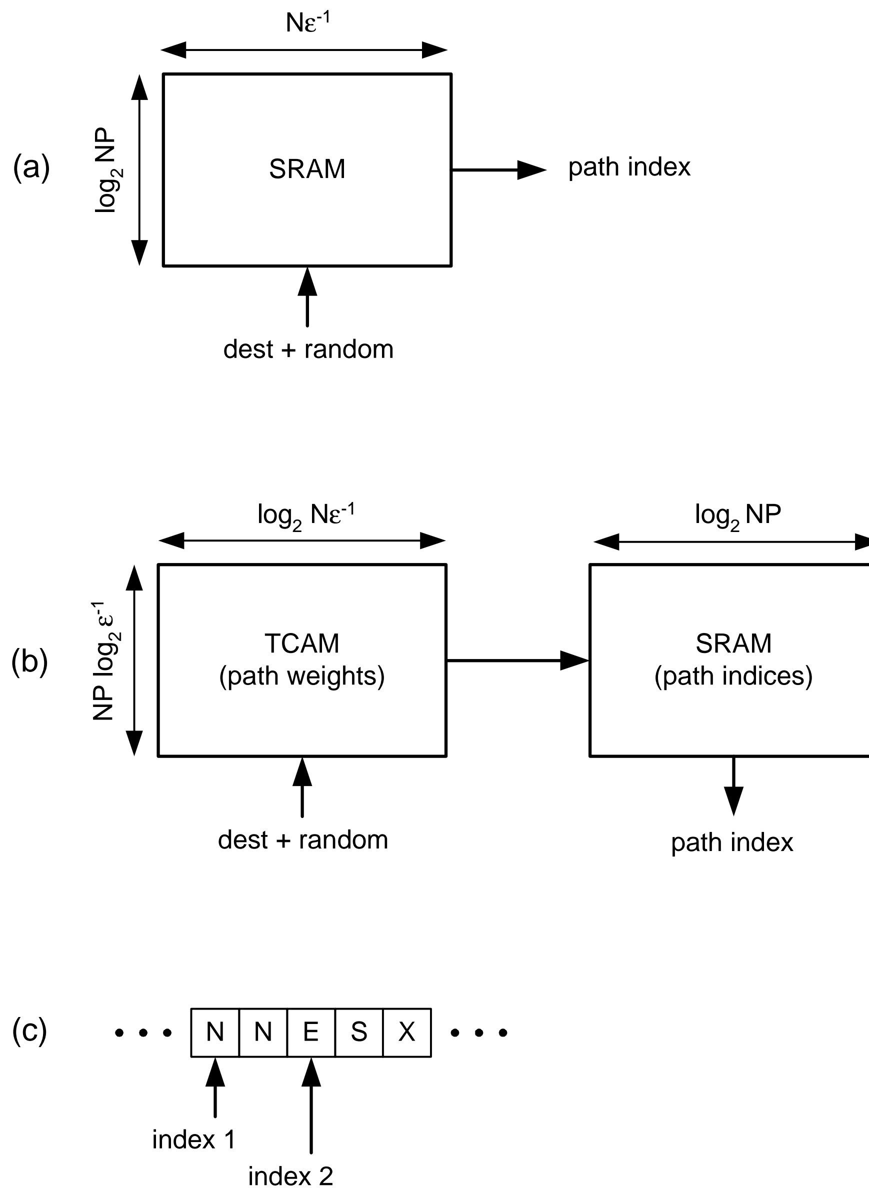


Figure 3.10: Hardware organizations for route lookup. P is the average number of paths per source-destination pair, N is the size of the network, and ϵ is the path weight granularity. (a) A direct, SRAM-based approach in which a destination-random value pair is used to access a table of path indices. (b) An indirect, TCAM-based approach that uses the TCAM to more efficiently store path weights. (c) Example of two path indices accessing a path table.

Table 3.2: TCAM encoding of a path of weight 28ϵ destined to node d.

Tag	Probability
d,00XXXX	1/4
d,010XXX	1/8
d,0110XX	1/16

path is $28/64$ and this is represented as $1/4 + 1/8 + 1/16$ in the TCAM. In general, any weight can be decomposed into at most $\log_2 \epsilon^{-1}$ entries — one entry per bit of the weight. So, the TCAM needs at most $NP \log_2 \epsilon^{-1}$ total entries. Once a tag has been matched in the TCAM, this address is used to index an SRAM that stores in the path indices. The number of entries in this SRAM is the same as the TCAM and, as in the SRAM only organization, the size of each entry is $\log_2 NP$ bits.

Qualitatively, the SRAM-based organization is more efficient when the number of paths is large and the TCAM-based organization is better at storing a smaller number of paths with high precision. To make a quantitative comparison of these two organizations, we derive simple estimates of the area of both approaches. For the SRAM-based approach, the total number of SRAM cells needed is

$$N\epsilon^{-1} \cdot \log_2 NP.$$

The cells of a TCAM are about 1.54 times larger than those of a SRAM[8] and using this conversion factor, the area of the TCAM-based organization is

$$1.54NP \log_2 \epsilon^{-1} \cdot \log_2 N\epsilon^{-1} + NP \log_2 \epsilon^{-1} \cdot \log_2 NP$$

SRAM cells. While these area models ignore other structures such as decoders, they provide a good first-order estimate of the total area required. Using these models, the minimum area organization is plotted as a function of the average number of paths per source-destination pair and the granularity in Figure 3.11.

[8]The 1.54 area factor is derived from the $52\lambda \times 24\lambda$ TCAM cell used by Shafai et al. [58] and a $41\lambda \times 28\lambda$ 6-transistor SRAM cell.

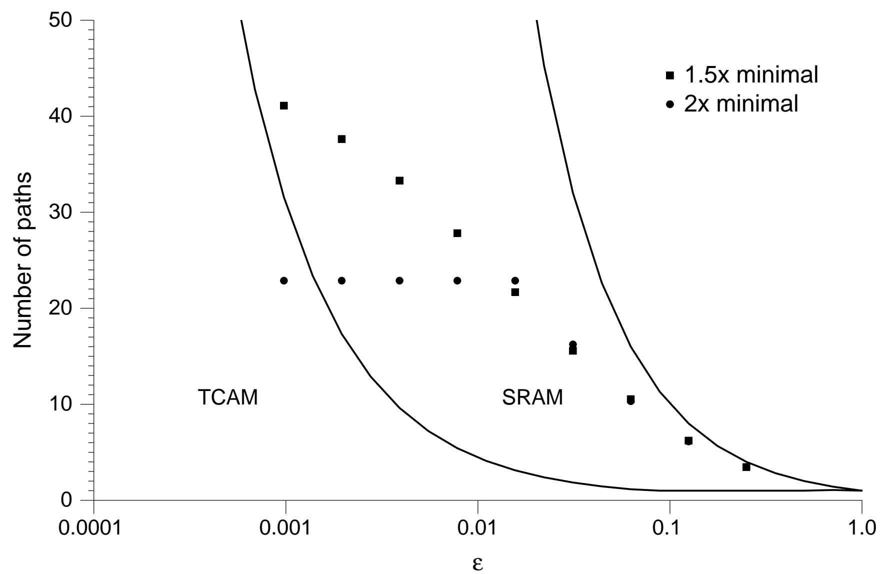


Figure 3.11: Minimum area hardware organization for route lookup as a function of the average number of paths per source-destination pair and the path granularity shown as a phase diagram. A 64-node network is used. The solid lines show the transitions between preferred implementations: a TCAM organization is preferred in the leftmost region, an SRAM implementation in the middle, and the rightmost region is infeasible. Also, the points shown indicate the number of paths and granularity for approximated worst-case optimal routing algorithms in a 8-ary 2-cube. Optimal algorithms with path lengths that are 1.5 times and 2 times minimal are shown.

As expected, the TCAM organization is preferred when the number of paths is small relative to the granularity. However, the area models reveal that for the two algorithms approximated in Figure 3.9, the SRAM implementation is preferred in all but one of the cases. That is, the number of paths generated by approximating at a particular granularity are more compactly implemented using the SRAM-based organization. Still, for networks where the number paths or granularity is lower, the TCAM implementation may be attractive.

3.5 Adaptive routing

In this chapter, we have assumed all of our routing algorithms are oblivious and therefore do not incorporate network state into their routing decisions. An important question is how much this costs in terms of the worst-case performance of a network. As previously mentioned, for many symmetric networks, specifically the torus [26, 61], it can be shown that adaptive routing offers no advantage in the worst case.

To explore this performance gap in non-symmetric networks, we compared the performance of optimal oblivious routing and adaptive routing for the faulty ring networks and random networks considered in Section 3.3.2. We consider an ideal adaptive routing algorithm that has perfect knowledge of the current traffic and minimizes channel load by solving a maximum concurrent flow problem (MCFP) as described in Section 3.1.1. If maximum channel load of this adaptive routing is thought of as a function of the traffic pattern, it can be shown that this function is convex. Then, finding the worst traffic pattern for an ideal adaptive algorithm is equivalent to maximizing this convex function. However, maximizing convex functions is difficult in general. (Convex functions can be *minimized* efficiently.) So, we determine a lower bound on the performance for adaptive routing by simply selecting a large number of traffic patterns, solving the corresponding MCFPs, and taking the worst channel load across all the patterns as a lower bound.

The surprising result after running this experiment on all the topologies from Section 3.3.2 was that the gap between adaptive and oblivious routing was zero in all the cases. Further experiments, though, reveal a gap does exist in some networks. To find these examples, a systematic exploration of all simple, connected, directed graphs with a

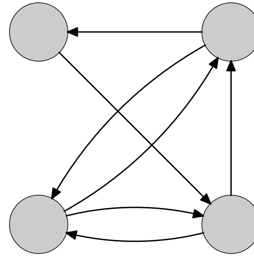


Figure 3.12: A network with a gap in the worst-case performance of an adaptive routing algorithm and an optimal oblivious routing algorithm. If each channel has a bandwidth of 1 bit/s, adaptive routing supports 1 bit/s of injection and ejection bandwidth from each node in the worst case while oblivious routing can only support approximately 0.875 bit/s per node.

given number of nodes was performed. Since the number of nodes was necessarily small, it was possible to find the exact worst case of adaptive routing by simply trying all possible permutation traffic patterns. For all graphs with two and three nodes, the gap was zero. Of the 83 unique graphs with four nodes, adaptive routing had a throughput approximately 14% higher than oblivious routing for the worst case in four cases. One of these topologies is shown in Figure 3.12 as an example. The gap was zero in the 79 other cases.

While the gap between adaptive and oblivious appears to be quite small, there are other aspects of adaptive routing that warrant its consideration in many cases. One of the drawbacks of oblivious algorithms is their potentially large routing tables. As shown in Section 3.4, the total size of routing tables tends to grow as $O(N^3)$ when trying to approximate an arbitrary oblivious routing algorithm with constant error. Adaptive algorithms do not generally need large routing tables such as these because their routing decisions are based on dynamic information. Also, recent work by Singh et al. [61, 62] introduces new approaches to designing adaptive routing algorithms. Specifically, the work indicates that approximate global information may be sufficient to design a practical adaptive routing that makes good global decisions and, hence, has high throughput in the worst case.

3.6 Related work

In parallel with the work on which this chapter is based, Räcke [54] developed an interesting theoretical result on the relative performance of oblivious and adaptive algorithms in the worst case. His work showed that in networks with undirected channels, oblivious routing is always $O(\log^2 N)$ competitive with adaptive routing in terms of channel load. That is, for any traffic pattern, an optimal oblivious routing algorithm loads a channel at most $O(\log^2 N)$ times as heavily as an ideal adaptive algorithm. This is a slightly different focus than our work as we are interested in the ratio of worst-case performance (as opposed to the worst-case ratio of performance) taken over of a set of admissible patterns. Still, his result adds more weight to the observation that oblivious routing algorithms can perform very well in a worst-case sense. Also, while Räcke's first results relied on an explicit, but exponential time construction, Azar et al. [5] have since shown that his routing algorithm design problem can also be formulated as a convex optimization problem.

Beyond the application of the ideas of this chapter in designing routing algorithms for IP router fabrics, they can certainly be used in any situation where robust routing is necessary. We explore this line of thinking in greater detail as part of Chapter 5, but it is worth mentioning that both Applegate et al. [4] and Kodiamlam et al. [40] have started exploring similar approaches to solve larger-scale routing problems in the Internet, such as routing between the many separate IP routers typically located in a central office.

3.7 Summary

This chapter has demonstrated that designing worst-case optimal oblivious routing algorithms is a tractable problem. The underlying reasons for this are that the worst-case channel load can be expressed as a convex function of the routing algorithm and that the set of oblivious routing algorithms is convex. These features enable the design problem to be cast as as a convex optimization problem. We showed two possible approaches to solving these problems, one based on a linear programming method and another based on a subgradient method. More detail on the subgradient method is contained in Appendix B.

Having a technique for finding optimal worst-case routing algorithms enabled several

interesting comparisons. Exploring the tradeoff between locality and worst-case throughput in the torus revealed that most existing algorithms lie far from the Pareto optimal set and several new algorithms along with the idea of interpolated algorithms were introduced to address this in the specific case of torus networks. Additional experiments showed that worst-case optimal oblivious routing algorithms also tend to offer the largest advantage over existing algorithms in highly irregular topologies. The details of taking the real-valued results produced by optimization methods and adapting them for hardware implementation were also explored. In general, worst-case optimal algorithms tend to spread their load finely over many paths. This, combined with the fact that the worst-case is quite sensitive to approximation error, necessitates large routing tables.

Chapter 4

Flow control

Broadly speaking, flow control refers to the coordination and scheduling of packets in time through a network. While our focus to this point has largely been on routing (scheduling in space), we now shift our attention to flow control issues specific to distributed router fabrics. One of the challenges of building an efficient fabric is handling the variation in IP packet sizes robustly. That is, for any packet size distribution, an IP router should continue to deliver data at full rate. As discussed in Section 4.1, an interconnection network typically subdivides a packet into smaller chunks, called *flits*. Flits, or flow-control digits, are then used as the unit of resource allocation throughout the fabric. Their small, fixed size simplifies the fabric router design and also decouples fabric buffering requirements from IP packet size.

However, in Section 4.2 we show that the standard approach of fixed-size flits leads to large control overheads. In a typical network, for example, fixed-size flits require approximately 0.6375 control bits to be sent for every data bit. To solve this problem, we introduce the idea of variable-size flits. As we show, variable-size flits reduce overhead in our example to just 0.2 control bits per data bit while only requiring minor changes to a typical router implementation.

Variable-size flits tackle the overheads that are most problematic in small packets, but, as discussed in Section 4.3, fabric designers must also carefully consider long-packet effects. Experimentally, long packets tend to drastically reduce the throughput of a fabric — sometimes by over 50%. This loss is attributed to the fact that long packets are spread

over many fabric routers, thus coupling channel resources and reducing throughput. The solution is to divide long IP packets into many smaller fabric packets. Combined with variable-size flits, this approach does not increase control overhead, but does avoid the loss in performance associated with longer packets.

Finally, IP routers are generally designed so that the flow of packets between a source-destination pair leaves the router in first-in first-out order. This requires the use of reorder buffers at the outputs. In Section 4.4, we present a window-based scheme for reordering and develop expressions for both the added latency due to reordering and the size of the reorder buffer. As these expressions show, reordering latency and reorder buffer size are related to the variation in delays through the network, the rate at which packets are injected into the fabric, and the burstiness of this rate.

4.1 Background

Before we delve into several of the flow control issues specific to distributed fabrics, we present a brief background on both IP packets and the flow control commonly used in interconnection networks.

4.1.1 IP packets

The most relevant feature of Internet protocol (IP) packets to the design of a switch fabric is their variable size. IP packets can be at most 64K bytes [53] and valid packets as small as 28 bytes can be constructed.[1] Of course, most traffic on the Internet is TCP/IP for which the minimum packet size is a 40 byte TCP acknowledgment (ACK). Also, much of the traffic is delivered across at least one segment of an Ethernet network whose maximum transfer unit is 1500 bytes [33].

Figure 4.1 shows a typical distribution of packet sizes measured at an Internet router. The most common packet sizes are 40 and 1500 bytes corresponding to TCP ACKs and the Ethernet maximum, respectively. In terms of the number of bytes of data delivered through routers, 1500 byte packets dominate. For example, in the distribution of Figure 4.1,

[1] An ICMP packet can be constructed with a 20 byte header and 8 bytes of data.

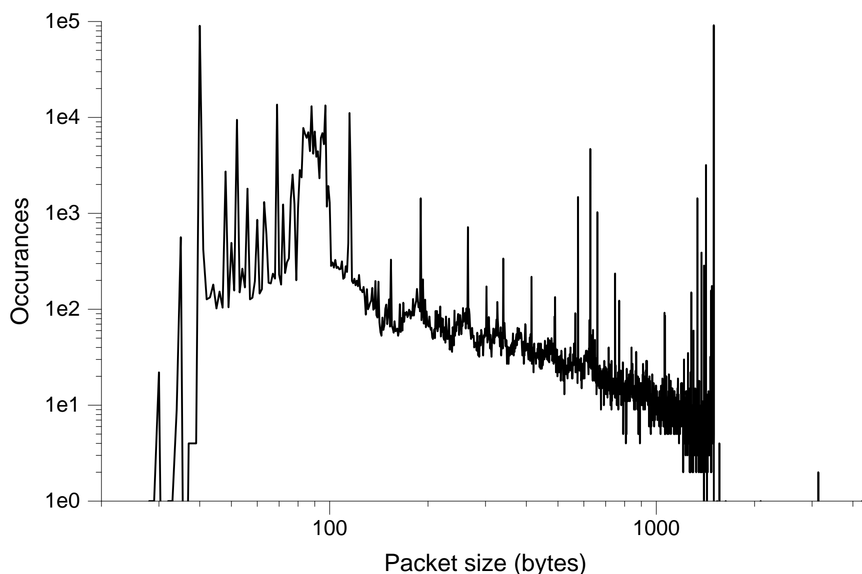


Figure 4.1: A distribution of Internet (IP) packet lengths captured at the University of Memphis on October 4$^{\text{th}}$, 2002.

almost 75% of the delivered bytes are contained in 1500 byte packets. If routers were designed for this typical distribution, packet formatting issues would be largely moot — as long as the flit size was chosen to be much smaller than 1500 bytes, any overhead due to fragmentation and control information would be amortized over the long packet. However, current practice is to design the router to be robust to changes in the packet distribution. Under this consideration, the extremes (minimum and maximum) of the distribution play a more important role as we will see.

4.1.2 Flow control in interconnection networks

Most of today's routers are built using flit-buffer flow control [25, 26]. The key idea behind this approach is to decouple the unit of routing through the fabric from the unit of resource allocation where resources are primarily channels and buffers. Figure 4.2 shows a typical packet formatting for this flow control method. As shown in the figure, a packet is the unit of routing and contains control information that is used by the fabric routers to determine its route. For delivery through the network, the packet is split into smaller flits or flow-control digits. These flits follow the same route through the network, so a packet is always

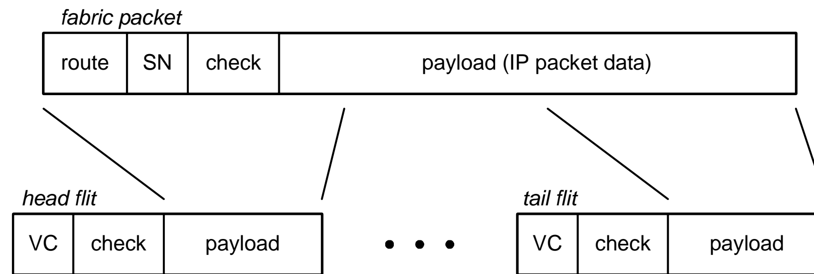


Figure 4.2: A typical packet format for flit-buffer flow control. Packets are the unit of routing and may contain route information and packet sequence numbers (SN), for example. These packets are broken into one or more flits for transmission across the network. Flits contain a virtual-channel identifier (VC) that ties them to the same packet and may also contain other control information.

delivered as an entire unit. However, each flit must individually vie for access to each of the resources (channels and buffers) needed for delivery. Then, the flow of flits is pipelined through the network — as the first (head) flit arbitrates for access to a channel, the second flit arbitrates for access to the channel just held by the first flit, for example.

As is common in many interconnection networks, distributed fabrics also take advantage of virtual channels [22]. That is, the buffer resources associated with a single physical channel are divided to support many parallel, virtual channels. Each virtual channel is multiplexed onto the single physical channel. This allows, for example, a packet traveling on one virtual channel to "pass" a blocked packet traveling on a different virtual channel (VC). To isolate traffic destined to different outputs of the fabric and in different service classes, a VC is typically associated with each service class-destination pair. This use of VCs to isolate traffic types is analogous to the idea of providing virtual-output queues [44, 67] at each of the inputs of an input-queued crossbar. The result is a need for 100s to 1000s of VCs. Flit-buffer flow control makes supporting this many virtual channels much more manageable as each fabric router only needs to able to store a single flit per VC instead of an entire packet.

4.2 Flit sizing

A typical interconnection network uses a fixed-size flit and, in this case, selecting a flit size that minimizes control overhead is a tradeoff between fragmentation and per-flit overheads — a small flit size results in more bandwidth dedicated to the per-flit control information, while a large flit wastes bandwidth for packets that do not fit into an integer number of flits. As we show in Section 4.2.1, even for an optimally chosen flit size, this results in high overheads. For our example network, 0.6375 control bits are required per data bit. To solve this problem, we introduce variable-size flits (Section 4.2.2), which greatly reduce overheads. For the same example network, only 0.2 control bits are required per data bit. Since the size of a variable-size flit is bounded to a small range, their hardware implementation remains simple and requires only small modifications to a typical router microarchitecture (Section 4.2.3).

4.2.1 Fixed-size flits

For simplicity of implementation, most interconnection networks are built using fixed-size flits. In picking a flit size, the designer must take router microarchitecture, fragmentation, and control overheads into consideration. The microarchitecture itself generally places an upper and lower bound on the size of an individual flit. The upper bound arises from the limited buffering on chip. Since resources are allocated in units of flits, there must always be enough buffering to store a flit before that flit can be sent. Moreover, the buffering of a fabric router is generally split between many parallel virtual channels, one for each output port of the fabric, so that maximum flit size is limited to be smaller than the maximum IP packet size. Small flits stress the resource allocation speed of a router. Since resources must be allocated every flit arrival time, smaller flits require faster, more deeply pipelined allocators. See Dally and Towles [26] for further details on how flit-size affects microarchitecture.

Overhead due to fragmentation is a direct result of our choice of fixed-size flits. For example, if the flit size is chosen so that the flit body can hold 40 bytes, sending a stream of 41 byte packets results in an overhead of almost 100% because each 41 byte packet must be split into two 40 byte flits. Obviously, picking a small flit size mitigates this effect, but,

at the same time, it increases the overhead due to control.

Control overhead is introduced for both packets and flits. At the packet level, routing information, a sequence number, and check field are added to each IP packet. The routing information is internal to the router fabric and is used to direct the packet to its output port. A sequence number is also attached to the packet because the fabric itself does not guaranteed in order delivery of packets and reordering may need to be performed at the output port (Section 4.4). The packet check field protects both the routing information and sequence number. A packet data check field is optional, but because the IP packet itself contains a checksum field, a packet-level check would only truly be necessary if the internal links of the fabric had a higher error rate than the long-haul optical fibers that connect IP routers.

At the flit level, a virtual channel field identifies the packet to which a particular flit belongs. To provide isolation between traffic destined to different outputs, a virtual channel is provided for each output port of the network. Once a flit leaves a particular router, flow control information must be sent back upstream to indicate the newly available buffer. Assuming credit-based flow control [26], this requires one credit to be sent per flit containing the virtual channel. Finally, both flits and credits are generally protected with a check field.

Let P_0 be the amount of per-packet control information (in bits) and let F_0 be the amount of per-flit control information. (We will discuss typical values for these overheads shortly.) Then, the total amount of data transmitted to sent a packet of P bits can be written as a function of both the packet size and the flit size F as

$$D(F,P) = (F_0 + F)\left\lceil \frac{P_0 + P}{F} \right\rceil . \tag{4.1}$$

The left term of (4.1) is the size of a single flit and the right term is simply the number flits in the packet — rounding up to the next multiple of the flit size accounts for the increased overhead due to fragmentation.

The next question is how to select a fixed value of F to minimize overhead while being robust to the packet size distribution. We first define α as worst-case number of total bits sent per true data bit,

$$\alpha(F) = \max_{P_{\text{min}} \leq P \leq P_{\text{max}}} D(F,P)/P,$$

where P_{min} and P_{max} are the minimum and maximum packet sizes, respectively. We assume $P_{\text{min}} = 40$ bytes and $P_{\text{max}} = 1500$ bytes for the rest of this chapter. Then, minimizing overhead is a matter of selecting the value of F that minimizes α.

This expression for α can be greatly simplified by observing the behavior of $D(F,P)/P$ for a specific value of F. As P is increased, $D(F,P)/P$ tends to decrease as control overheads are amortized over the larger packet size. However, periodic "jumps" occur at the packet sizes that correspond to the addition of another flit. This is shown in Figure 4.3. The peaks of later jumps (larger packet sizes) decrease because the packet overhead is being amortized over a longer packet. Based on this, the packet size that maximizes overhead is always the minimum packet size or the smallest packet that requires one more flit than the minimum-size packet. Thus, we can rewrite α as

$$\alpha(F) = \max\left\{\frac{D(F,P_{\text{min}})}{P_{\text{min}}}, \frac{D(F,P_{\text{jump}})}{P_{\text{jump}}}\right\}. \tag{4.2}$$

where

$$P_{\text{jump}} = 8\lceil(F\lceil(P_0 + P_{\text{min}})/F\rceil - P_0)/8\rceil \tag{4.3}$$

The outermost ceiling in (4.3) accounts for the fact that IP packets have integer byte lengths (multiples of 8 bits). Also, (4.2) holds as long as the maximum packet size occurs after the first jump ($P_{\text{max}} \geq P_{\text{jump}}$).

Following from (4.2), it is interesting that P_{max} plays such a minor role in determining the optimal flit size. While one might suspect that finding the worst-case overhead over larger and larger ranges of packet sizes (increasing P_{max}) would incur more overhead, this is not the case. For example, if $P_{\text{max}} \geq 2P_{\text{min}}$, overhead becomes independent of P_{max}. This follows from the simple observation that sending two packets always incurs more overhead than sending one packet of twice the length. Thus, as correctly reflected in the bounds, overhead is largely insensitive to maximum packet size.

To quantify the effects of overhead on the choice of flit size, consider a typical 256-port fabric implemented as a 2-dimensional torus network. Source routing is used and all routes can take at most two turns, which requires 18 bits to encode (2 bits for the initial direction and each turn and 4 bits for the length of each run). The number of sequence numbers required is determined by the largest number of packets in flight in the fabric

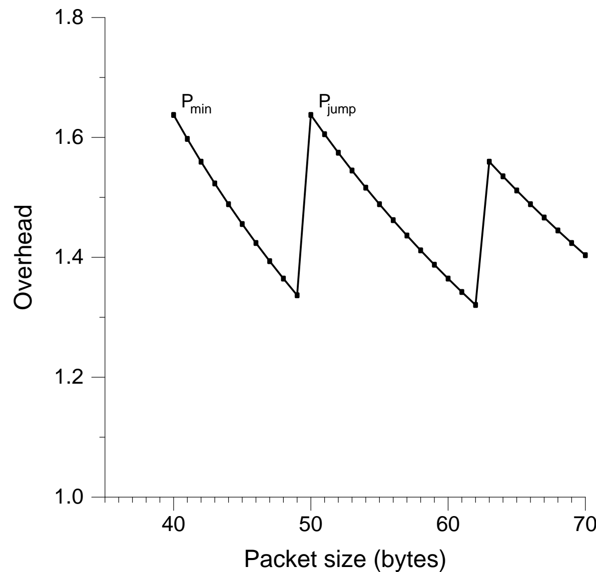


Figure 4.3: An example of overhead vs. packet length. As shown, overhead is maximized at either the minimum-size packet or the first "jump" corresponding to sending an additional flit. The peaks of later jumps (greater packet sizes) are always smaller than the first.

from a particular input. This is determined by the bandwidth-delay product of the input and, assuming a typical 10 Gb/s input link, a 12-bit sequence number field is sufficient to support a large delay of approximately 164μs for minimum size packets. Also, the 256 virtual channels, one per output, are encoded in 8 bits in both the flits and credits. These overheads are summarized in Table 4.1.

The resulting overhead (α) as a function of the flit size is shown in Figure 4.4. For this example, the best flit size is 107 bits with an overhead of 1.6375. That is, for each data bit of an original IP packet, 1.6375 total bits must be sent over the fabric. This is a fairly substantial overhead with more than 1 of every 3 bits sent over the network used for control information.

Finally, selecting a flit size of 107 bits might give a hardware designer pause because most datapaths are powers of two or small multiples of a power of two. However, 107 just happens to be a prime number. This problem can be avoided, though, by providing a small amount of internal speedup to the routers. For example, a 107 bit flit could be aligned to a 16-bit data path by rounding 107 up to 128 inside the router. This wastes bandwidth due to fragmentation, but if the router is built with an internal bandwidth $128/107 \approx 1.2$

Type	Description	Size (bits)
packet	route information	18
	sequence number	12
	check	4
flit	virtual channel	8
	check	4
	credit VC	8
	credit check	4

Table 4.1: Summary of control overheads for a typical 256-node network with source routing ($P_0 = 34$ bits and $F_0 = 24$ bits).

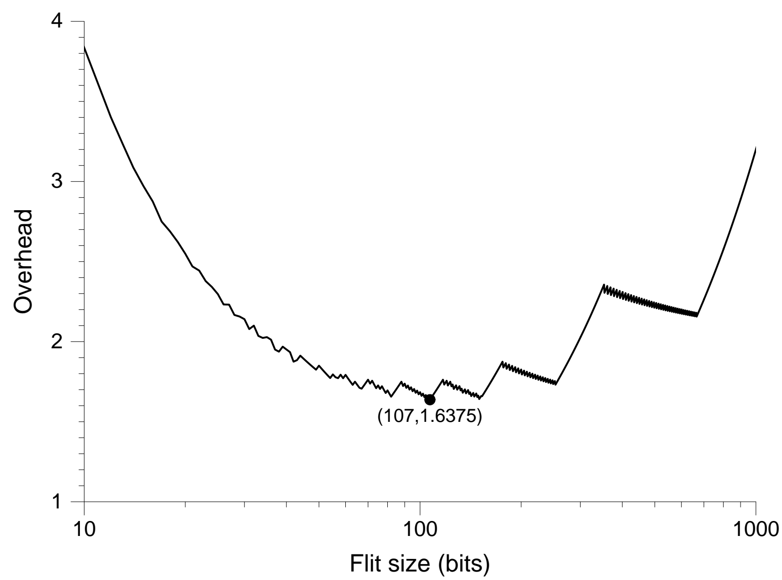


Figure 4.4: Control overhead as a function of the flit size for $P_0 = 34$ bits and $F_0 = 24$ bits.

times greater than its external (channel) bandwidth, no expensive channel bandwidth will be wasted. Such a small internal speedup is essentially free for the router chips because their area is generally pin limited.

4.2.2 Variable-size flits

Variable-size flits inherently tackle the overhead due to packet fragmentation. Instead of having to split a 41-byte packet into two 40-byte flits, for example, with variable-size flits that same packet could be split into a 40-byte flit and a 1-byte flit, eliminating fragmentation. Of course, that same 41-byte packet could be split into a 21-byte flit and 20-byte flit or could be contained in a single 41-byte flit. In fact, all packets could be sent in just a single flit — this would certainly eliminate fragmentation and would also minimize the total flit control overhead. However, this approach is equivalent to abandoning flits and their implementation advantages, such as small buffers and fine-grain resource allocation. Rather, we introduce an implementation of variable-size flits that eliminates fragmentation, but constrains flit sizes to maintain the advantages of distinguishing between flits and packets.

Independent of other constraints on the minimum flit size (F_{min}) and the maximum flit size (F_{max}), an important observation is that fragmentation can always be completely eliminated when

$$2F_{\text{min}} \leq F_{\text{max}}, \tag{4.4}$$

$$F_{\text{min}} \leq P_{\text{min}} + P_0. \tag{4.5}$$

To see this, consider the splitting of a packet into flits. Without loss of generality, assume the packet contains between F_{min} and $2F_{\text{max}} - 1$ bits. Maximum size flits can always be extracted from a longer packet until this condition is met and by (4.4) no packet shorter than F_{min} bits can be sent. Then, if the packet contains no more than F_{max} bits it is sent as a single flit. Otherwise, the X bits of the packet are split into a $\lfloor X/2 \rfloor$ bit and a $\lceil X/2 \rceil$ bit flit. Applying (4.4) and the constraints on X,

$$F_{\text{min}} \leq \lfloor X/2 \rfloor \leq \lceil X/2 \rceil \leq F_{\text{max}}.$$

By splitting the packet into two nearly equal flits, we are ensured both flits are at least minimum size. So, to eliminate fragmentation, the maximum-size flit should be at least twice the minimum-size flit.

The next consideration in constraining the size of a variable-size flit is the corresponding impact on control overhead. Intuitively, a larger maximum-size flit will reduce control overheads because the per-flit overhead (F_0) is amortized over a longer flit. While this is true for a fixed packet size, we are optimizing over a range of packet sizes. As with the fixed-flit size, the worst control overheads can only happen for two packet sizes: the minimum packet size $P_{\text{min}} + P_0$ or at one plus the next largest value of $P + P_0$ that is an integer multiple of F_{max}. This can be used to bound the maximum flit size by first assuming a minimum packet fits into a single flit ($F_{\text{max}} \geq P + P_0$). Then, as long as the overhead at the next multiple of F_{max} is less than the overhead for sending a minimum size packet in a single flit, further increasing the maximum flit size will not reduce overhead. Solving for that maximum flit size gives

$$F_{\text{max}} \geq 2P_{\text{min}} - \frac{(P_{\text{min}} - 1)(P_0 + 1)}{P_0 + F_0} - 2.$$

From this result, it is sufficient for $F_{\text{max}} = 2P_{\text{min}} + 2P_0$ to minimize control overhead and then selecting a minimum flit size of $F_{\text{min}} = P_{\text{min}} + P_0$ ensures no overhead from fragmentation. As long as the hardware has enough buffer space to store two minimum-length packets per virtual channel, fragmentation can be eliminated and overhead can be reduced to that of sending a minimum-length packet in a single flit

$$\frac{P_{\text{min}} + P_0 + F_0}{P_{\text{min}}}.$$

Moreover, as we will explore in Section 4.2.3, the 2-to-1 ratio of the maximum to minimum flit size allows a simple implementation.

To see the quantitative benefit of variable-size flits, we return to the example overheads from Table 4.1. In addition to comparing to fixed-size flits, we also compare to the approach adopted in the Avici TSR router fabric [24]. In the Avici TSR, two flit sizes are supported: a single flit and a double flit where two flits worth of data from the same packet are sent

back-to-back while sharing a single set of control information. From Section 4.2.1, the optimal flit size for the fixed-size flit approach is 107 bits, which yields an overhead of 1.6375 bits per original data bit. For the Avici TSR approach, the optimal single flit size is 58 bits, reducing overhead to about 1.464. Finally, using variable-size flits reduces the overhead to 1.2 when accounting for 6 additional bits per flit to specify its length — 26.7% lower than the fixed-size approach and 18.0% lower than the Avici TSR approach. Thus, variable-size flits can offer a significant reduction in control overhead.

4.2.3 Implementation

Variable-size flits can be implemented with a just few changes to a fabric's routers. For this section, we assume a typical input-queued microarchitecture: arriving packets are queued in buffers associated with input, queued packets arbitrate for access to a central crossbar switch, and, upon a successful arbitration, packets are switched from input to output. See Dally and Towles [26] for further details of input-queued router architecture.

Figure 4.5 shows how an input queue could be constructed to support variable-size flits. As illustrated, incoming flits are first deserialized in an input shift register. This shift register is used to align the flits to the input queues (typically implemented as an SRAM). To simplify the management of this input queue, it is allocated in units of flits and each queue entry is large enough to store a maximum-size flit. Since minimum-size flits are at least half the size of maximum-size flits, this ensures at least 50% utilization of the input buffer space. A virtual channel (VC) controller is responsible for managing this buffer as well as tracking the status of each of the VCs (e.g. idle, waiting for arbitration, waiting for credits, etc.).

A key aspect feature of the microarchitecture is that all control decisions are synchronized to *phit* or physical digit boundaries. A phit is simply a further subdivision of a flit and its size is chosen based on how frequently a router can make control decisions, such as switch arbitration. For example, if router decisions could be made at 500MHz and the channel rate was 10 Gb/s, a reasonable phit size might be $10^{10}/(5 \cdot 10^8) = 20$ bits or about 3 bytes. Then, assuming 3 bytes per phit, a 43 byte flit would be rounded up into fifteen 3-byte phits. This allows the entire router pipeline to work with fixed-size units of phits

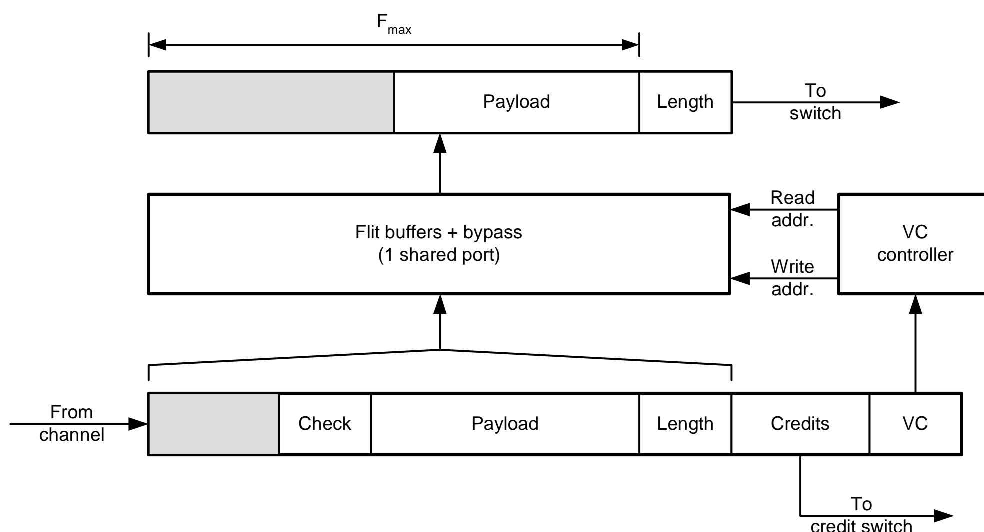


Figure 4.5: Input microarchitecture for a variable-size flit fabric router. Arriving packets are deserialized into entire flits with a shift register and then stored in a flit packet that is wide enough to store a maximum-size flit — the gray areas of the packets represent unused data in a less than maximum length flit. All control decisions are synchronized to phit boundaries.

and is much simpler than dealing with variable-size flits directly. A similar approach of making control decisions on phit boundaries is also used in the Alpha 21364 [47], but with the emphasis on reducing latency.

Another consideration in implementing a variable-size flit router is how flit and credit encoding is performed on the channels. The credit-based flow control method mentioned in previous sections often takes advantage of the fact that the average flit rate is equal to the average credit rate — each credit corresponds to a flit that had previously traveled across the channel in the opposite direction. Because of this relationship, returning credits can be piggybacked on the flits traveling in the opposite direction across a bidirectional channel pair. However, this does not work with variable-size flits because the rate at which flits are traveling across the opposite directions of a channel pair may differ because the size of the flits is different. For example, if 40 byte flits are traveling across a forward channel and 80 byte flits are traveling across the opposite channel, there need to be two credits per flit across the opposite channel. This effect is shown in Figure 4.6. A simple solution to this problem is to include space to piggyback enough credits in the worst-case mismatch

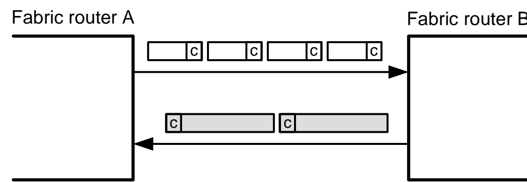


Figure 4.6: An example of flit and credit rate mismatch that can occur when credit-based flow control is used with variable size flits. Router A is sending 40 byte packets to router B and router B is sending 80 byte packets to router A. Since credits are returned to router A at only half of its flit rate, it will run out of credits and idle the channel.

of rates. Since we select a ratio of two between the maximum and minimum flit sizes, two credits are included with each flit.

Finally, care must be taken in the recovery of the channel in case of a bit error. When such an error is detected using the check bits of a flit, the length field of that flit must be considered as corrupted. This means the router no longer knows when the current flit ends and the next flit begins. Before the error is corrected, the routers must resynchronize. One approach would be to send a special flag to the upstream router using the opposite channel. This flag would first initiate a resynchronization procedure followed by a recovery of the corrupted flit, if necessary. Of course, this does not handle the case of when the opposite channel has also lost synchronization. Fortunately, such an event would be exceedingly rare and the efficiently of the recovery operation would not be important. The further details of robust synchronization recovery are left as future work.

4.3 Packet sizing

In the previous sections, we have not make a distinction between IP packets and fabric packets. For an interconnection network, packets are the unit of routing — all the flits in a *fabric packet* must follow the same path from source to destination. Also, as part of the IP protocol, each *IP packet* must leave the router as contiguous bits. However, this does not preclude breaking arriving IP packets into one or more fabric packets, transferring these fabric packets from source to destination, and then reassembling the original IP packet at the output before it leaves the IP router. Moreover, as we show in this section, not breaking

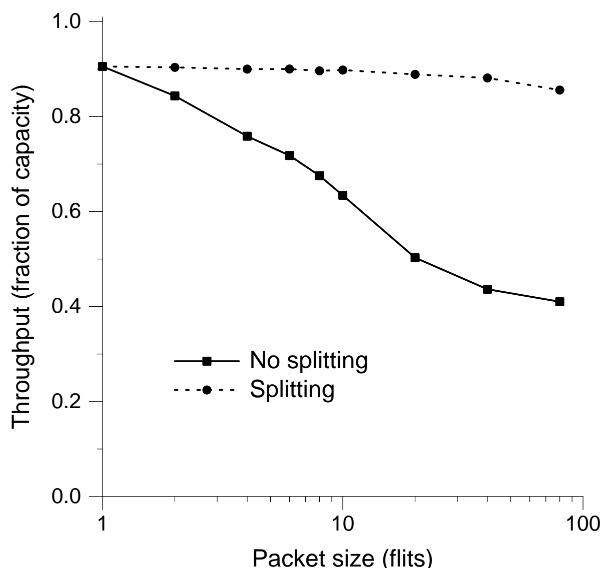


Figure 4.7: Saturation throughput of an 8-ary 2-cube (torus) network for various IP packet sizes under uniform traffic. Both the throughput of network in which IP packets are sent as a single fabric packet ("no splitting") and the throughput of a network in which IP packets are split into single flit fabric packets ("splitting") are shown.

long IP packets into smaller fabric packets can result in an over 50% loss of throughput. The solution is to divide these long packets into fabric packets that are approximately the size of minimum-size IP packets. This avoids additional overhead when used in conjunction with variable-size flits (Section 4.2.2) and eliminates the loss of throughput associated with long packets.

Figure 4.7 shows the effects of long packets on the saturation throughput of a 8-ary 2-cube network. For both simulations, the fabric routers are designed to model the event-driven routers used in Avici's distributed router fabric [24]. A separate virtual channel is used for each of the 64 destinations and each router has 2 flits of buffering per virtual channel. In addition, Valiant's routing algorithm is used. The size of incoming IP packets is varied along the horizontal axis. If these incoming IP packets are not split into smaller fabric packets, throughput degrades from approximately 90.6% of the network's capacity to roughly 41.0%. A second experiment splits each of the incoming IP packets into many single-flit fabric packets and only a modest reduction in throughput, to 85.6% of capacity, is observed.

The dramatic loss in throughput as packet length increases is due to the fact that the fabric routers have relatively shallow buffers. As mentioned in Section 4.1.2, an IP router needs a virtual channel (VC) for each service class-destination pair to prevent blocking and the result is a fabric router that needs to support 100s to 1000s of VCs. This, in turn, restricts the amount of buffering associated with any single VC due to area constraints. Since each VC must also have at least one flit of buffering, this also limits the maximum flit size.

When sending a fabric packet longer than the amount of per-VC buffering, it must be the case that this packet is spread over multiple fabric routers. The upside of this approach is that it allows the fabric packet and flit sizes to be chosen independently. However, the downside is that spreading packets over multiple routers creates a coupling between routers — for a packet to flow through the network, all of the routers that it is spread across must cooperate. If just one of the cooperating routers is busier than the others, then it limits the rate at which the packet can progress. This is the primary source of the throughput loss when sending long packets through the network. It also suggests that an important consideration is the ratio of a packet's length to the amount of per-router VC buffering. For example, a ratio of two implies that a packet is spread across at least two routers.

An upper bound on the saturation throughput for a certain packet length can be constructed by only modeling the router coupling that occurs at source nodes. We consider the case of one VC per fabric destination (no distinct traffic classes) and this implies that each destination serves a single packet at a time. The choice of the order in which a particular destination serves packets is distributed across the fabric routers: as packets for the same output merge at the different fabric routers, local arbitrations select which packets are forwarded. Since these local arbitrations are independent and fair, they can be approximated by randomly selecting the next packet served by a destination from the sources that have packets waiting for that destination.

These assumptions lead to a simple approach for modeling the throughput of packets while only considering coupling at sources:

1. Virtual-output queues are maintained at each source. IP packets arriving at a fabric source are queued according to their destination.

2. Each destination randomly selects a source from which to serve its next packet from among the sources with packets queued for that destination.

3. The maximum number of bits of a packet are assumed to be queued in the network when service starts. This maximum number of bits is equal to the length of a packet's path (in hops) times the number of bits of buffering per VC.

4. The packet is served at the destination's full rate for all the bits assumed to be in the network or the packet's entire length, whichever is shorter.

5. If the entire packet cannot be stored in the network, any extra bits are served at the rate of the source. The source rate is equal to its bandwidth divided by the number of packets it is currently serving.

The first two parts of the model are simply a restatement of the operation of a typical fabric network. Parts three and four form an ideal network — when a destination begins service on a packet, it is only limited by the destination's bandwidth. Part five incorporates the fact that even an ideal network only has a finite amount of queuing and packets that cannot fit within these queues must be served from the source. This introduces coupling — both the source and destination need to cooperate to serve long packets. Since the destinations independently choose the next source to serve, multiple destinations may simultaneously need access to the same source. In these cases, the source becomes a bottleneck and limits the service rate of all the destinations accessing it.

Figure 4.8 shows the throughput of this simple model for the network used in the first experiment of this section. While meant as an upper bound, the model shares the approximate features of the full network simulation. Specifically, the packet lengths with the steepest decline in throughput roughly coincide as do the points where the both curves begin to level out. The actual network does show an earlier decline in throughput and this is attributed to coupling between routers that begins to occur as soon as a single packet begins to spread across multiple routers. The model ignores this effect. Also, despite the very optimistic network assumed by the model, throughput is still reduced to approximately 55% of capacity for very long packets.

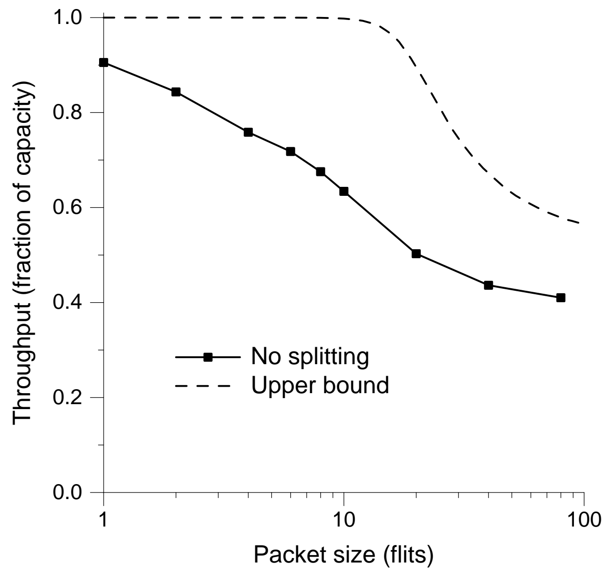


Figure 4.8: A simple model of the limits on saturation throughput due to packet length.

4.4 Reordering

In striving to maximize the throughput of a distributed router fabric, we have made design choices that can both reorder the sequence of IP packets traveling from a particular source to a particular destination and can also reorder the data within an IP packet. These effects are due to the fact that fabric packets can follow different paths with potentially different delays through the fabric and that long IP packets are broken in several smaller fabric packets, respectively. At the very least, if an IP packet is broken up for transmission across the fabric, it must be reassembled in its original reception order before being transmitted out of the IP router. Moreover, most IP routers are designed to avoid (or at least limit) the reordering of packets flowing between each source-destination pair. That is, the flow of data between each source-destination pair should appear to be in first-in first-out order when observing the external ports of the IP router.

Packet ordering is typically required because TCP, the congestion control mechanism use by most Internet traffic, performs poorly in the presence of misordered packet arrivals. See Blanton and Allman [10], for example, for specific details about TCP and reordering.

This is rarely a problem in centralized fabric architectures, which naturally maintain ordering, but other, more scalable fabric organizations often have to deal with this problem. For example, Iyer and McKeown's parallel packet switch [36] solves this problem by ensuring a limited amount of missequencing within their switching fabric and then providing a small coordination buffer at the output of the switch to reorder the packets. Keslassy [39] adopts a frame-based approach in his load-balanced router that also leads to a bounded amount of missequencing. However, these approaches inherently rely on the topology of their fabrics — both have Clos-like organizations. The Clos topologies combined with the synchronous operation of the routers leads a much more deterministic progression of packets through the fabric. These approaches do not easily extend to our distributed fabrics because of the fabrics' potentially irregular topologies and no assumptions about router synchronization.

To reorder packets in a distributed router fabric, we use a common sliding-window approach such as that Tanenbaum [68]. In this approach, each source attaches a unique sequence number to each fabric packet — this is the same sequence number we accounted for in the overhead calculations of Section 4.2. Each source maintains a sequence number counter for each destination. So, for example, if a source's counter associated with destination 1 has a value of 10, the next packet sent from that source to destination 1 is assigned the sequence number 10. Then, the counter is incremented and the next packet from that source to destination 1 receives a sequence number of 11 and so on. At each destination, a separate reorder buffer (ROB) is maintained for each of the sources. An arriving packet is then written into the buffer associated with its source using its sequence number to determine the address in the buffer to use. Continuing the example, the packet with sequence number 10 is written into entry 10 of the buffer associated with its input. Each destination also keeps counters for each source queue that remembers the sequence number of the next in-order packet. So, if the destination's counter for our example packet was 10, the packet could immediately leave the destination's port of the IP router and the counter would be incremented to 11. However, if the counter was 9, the previous packet has not yet been received, so packet 10 would be queued and wait until packet 9 arrived. We also assume that the ROB is organized as a circular buffer so that sequence numbers are eventually reused.

We will focus on the latency of reordering and the size requirements for ROBs for the rest of this section. The implementation details are beyond the scope of this thesis, but it is

worth noting that a practical implementation would use an $O(1)$ approach to maintaining and querying the ROBs. That is, only a constant number of buffer entries could be changed or examined each cycle. Also, it would be ideal if the ROBs could be kept on a single chip to eliminate expensive off-chip memory references.

Figure 4.9 shows the additional latency incurred by reordering packets under two different traffic patterns in the same router fabric. The left graph shows uniform traffic with single-flit IP packets and the right graph shows bit complement traffic[2] with 20 flits per IP packet. These long IP packets are broken into single-flit fabric packets for transmission and then reordered at the destination. As we will see shortly, the difference in reordering latency is explained by two factors. First, the bit complement pattern is a permutation — each source sends to a single destination. Since reordering occurs for source-destination pairs, permutation patterns offer the most chances for reordering as every packet from a particular source needs to stay in order with respect to all the other packets leaving that source. This is in contrast to the uniform pattern, where only one of every N packets from a source go to a particular destination. Second, the 20-flit IP packets of the bit complement case lead to a more bursty injection process — an arrival of a single IP packet creates a burst of 20 fabric packets, all of which must be ordered.

From these two experiments, we can also get a sense of the size requirements for a reorder buffer. For example, at an offered load of 40% of capacity in the bit complement case, the average reordering latency is approximately 113 flit cycles.[3] By Little's Law, each destination's reorder buffers contain a total of $113 \cdot 0.4 = 45.2$ flits on average, which is quite small. However, in terms of implementation, the range of ROB occupancies is just as important as the average. If a ROB becomes full, it must apply backpressure to the fabric to prevent lost packets.[4] This, in turn, reduces the throughput of the fabric. With the goal of ultimately developing an expression for the size of the ROB, we first focus on the delay incurred by reordering.

[2]In bit complement traffic, each source sends to the destination whose address is the bit-wise complement of the source's address.

[3]A flit cycle is the time for a minimize-size flit to be injected into the fabric from the source — the flit size divided by the source bandwidth.

[4]This backpressure must be applied directly to the source nodes to ensure the ROBs do not overflow. As an example, this could be accomplished with end-to-end packet . If instead the ROB simply stopped accepting packets when it became full, deadlock would occur.

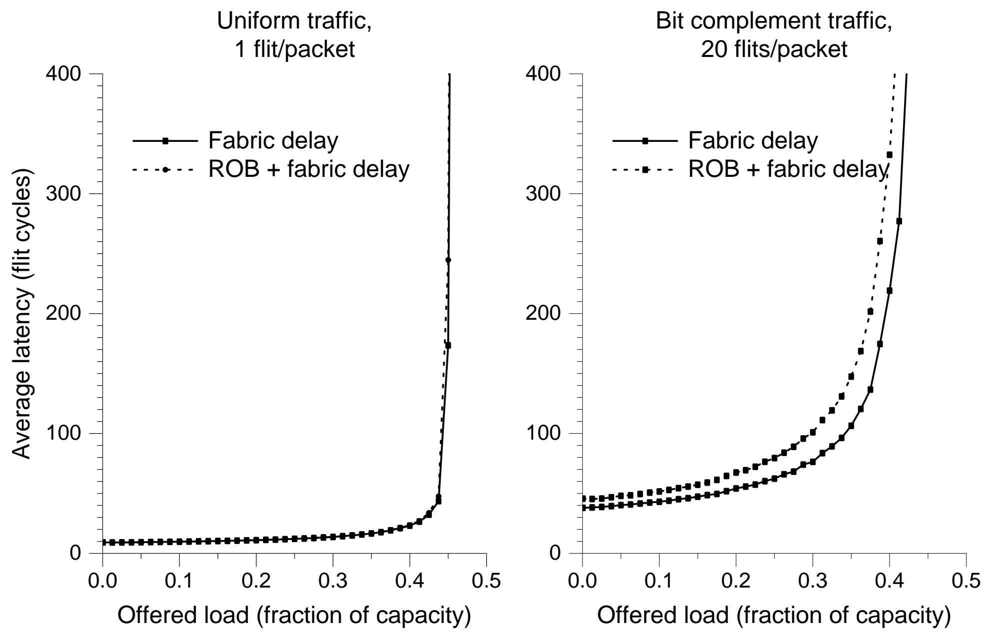


Figure 4.9: Two cases of the average fabric and total latency vs. offered load in an 8-ary 2-cube fabric. The left graph shows the performance of single flit IP packets and uniform traffic; the right graph shows the performance of 20 flit IP packets, which are broken into 20 single-flit fabric packets, under bit-complement traffic. Valiant's routing algorithm (Table 3.1) is used.

For a particular packet injected into the fabric at time t be to delayed in a ROB, an earlier packet sent from the same source to the same destination must still either be in the ROB when the later packet arrives or yet to arrive at the ROB. Denote the fabric delay of a packet sent at any time t as $D(t)$ and let $A(t)$ be 1 if a packet was sent at time t and zero otherwise. Figure 4.10 shows the condition when the packet sent at time t is delayed by an earlier packet sent at time $t-k$.

As illustrated by the figure, the later packet must first wait for the arrival of an earlier packet, which occurs at time $t-k+D(t-k)$. Then, all the packets sent from time $t-k+1$ until time $t-1$ must be drained from the ROB before the packet sent at time t can leave.[5] The resulting ROB delay of the packet sent at time t is denoted as Δ in the figure. In Figure 4.10, we have assumed that the packet sent at time $t-k$ is *critical*. That is, once it arrives in the ROB buffer, the buffer drains continuously until the packet sent at time t is removed from the ROB. In general, any packet sent before time t can be critical. Thus, the ROB delay of the packet $R(t)$ is the maximum over all earlier packets,

$$R(t) = \max_{k>0} \left(D(t-k) - D(t) + \textstyle\sum_{i=t-k}^{t-1} A(i) - k \right), \quad (4.6)$$

where $D(t-k)$ is defined as zero when no packet was sent on cycle $t-k$.

Although the queuing behavior of a ROB is complex, (4.6) illustrates that ROB delay is affected by both the difference in fabric delays, $D(t-k) - D(t)$, and, less obviously, the fabric injection process A. A fabric with a wide range of delays will create more opportunity for reordering and, at the other extreme, a fabric with constant delay eliminates all reordering. Moreover, design techniques that reduce the variation in network delay, such as the use of age-based arbitration [26], also reduce reordering costs. The connection between ROB delay and the injection process is because bursty injections also introduce more reordering — it takes less variation in network delay to cause reordering if packets are injected in closely-spaced bursts compared to more evenly-spaced injections.

The expression for ROB delay (4.6) can also be used to derive simple models or bounds. Chernoff bounds can be used to express ROB delay in terms of the variance of network delay and the injection process, but these bounds tend to be loose because delay distributions

[5] For clarity, this model assumes fixed-size flits and that the reorder buffer can drain at one flit per cycle, but it can be extended to more general situations.

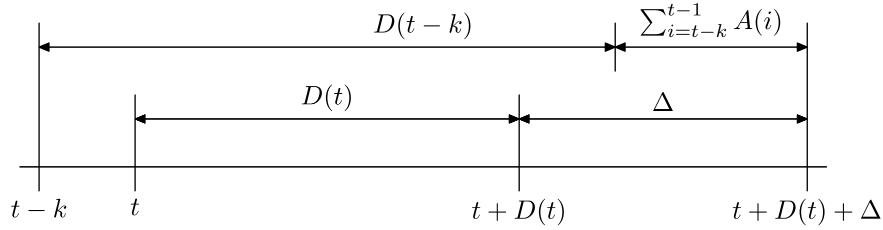


Figure 4.10: Reorder buffer timing for a packet sent on cycle t being delayed by a packet sent k cycles earlier. The delay Δ is incurred because the later packet must first wait for the earlier packet to arrive and then wait for packets sent between t and $t-k$ to drain from the reorder buffer.

typically fall off exponentially. For illustration, we use the simple bound

$$\mathbf{Pr}\left[\text{ROB delay} \geq x\right] \leq \sum_{k>0} \mathbf{Pr}\left[D(t-k) - D(t) + \textstyle\sum_{i=t-k}^{t-1} A(i) - k \geq x\right] \tag{4.7}$$

and further assume all delays and injections are independent. Then, by using a measured delay distribution from the uniform traffic case of Figure 4.9, a simple model of ROB delays can be derived. The resulting model, and its comparison to measured ROB delays, is shown as the left graph of Figure 4.11. For this case, the simple approximation worked extremely well, but is admittedly less accurate in general.

We can use the reordering delay to determine the number of packet entries required in the ROB. We begin with a ROB that can hold an infinite number of entries. At the instant a new packet sent at time t arrives at the ROB, the number of entries required is exactly the difference in the sequence number of the arriving packet and the sequence number of the next packet to leave the ROB. This next packet is the oldest outstanding (unordered) packet still in the fabric or ROB. Then, the probability the ROB size exceeds x packets is exactly the probability that the packet sent x packets before the newly arriving packet is still outstanding. Assuming this packet was sent k cycles earlier, it is still outstanding if the time it leaves the ROB ($t-k+D(t-k)+R(t-k)$) is greater than the arrival time of the new packet ($t+D(t)$). Accounting for the probability that the packet sent x packets

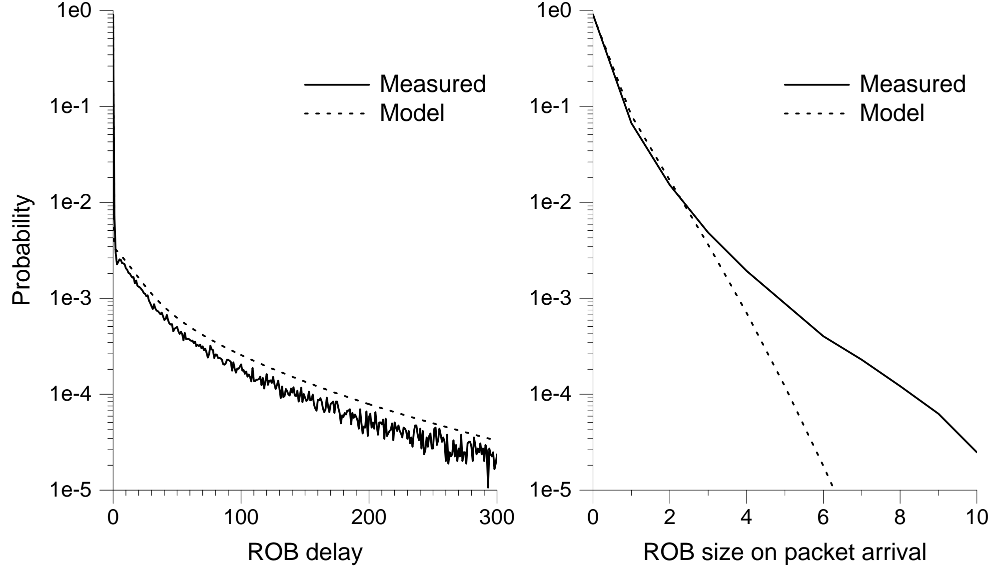


Figure 4.11: Distribution of reordering delay and reorder buffer size for an 8-ary 2-cube network with 1 flit per packet and uniform traffic. Both measured data and approximations based on (4.7) and (4.8) are shown.

ago was sent k cycles ago gives

$$\mathbf{Pr}\left[\text{ROB size} \geq x\right] = \mathbf{Pr}\left[\bigcup_{k>0} \left(\sum_{i=t-k}^{t-1} A(i) = x\right) \cap \left(D(t-k) + R(t-k) - k \geq D(t)\right)\right]. \quad (4.8)$$

As would be expected, the ROB size is directly tied to the ROB delay and any technique to reduce reordering delay will also reduce the ROB size. Figure 4.11 also shows an approximate distribution of the ROB size using the same set of assumptions as in the approximation of the reorder latency compared to measured values.

The key point emphasized by these analyses is that reducing reordering delay and reorder buffer size is about reducing both fabric delay variation and injection variation. A designer might have several options for reducing delay variation. As we previously mentioned, using age-based arbitration in the fabric (oldest packets first) tends to reduce delay variation. Fabric speedup would also reduce variation as over-provisioned fabric channels could handle more congestion before needing to delay packets. Delay variation is also

connected to routing — if a particular source-destination pair uses paths whose lengths (in terms of the number of hops) vary greatly, delay will also tend to vary greatly. So, for example, a designer could try to eliminate long paths as part of either the initial formulation of a path-based worst-case optimal routing problem or during the randomized rounding of the routing algorithm's path weights (Section 3.4.1). Another possible approach is to design a routing algorithm to minimize variance — for a particular average path length, the variance in packet length is convex in the routing algorithm's path weights (p_{kij}). By sweeping average path length and holding the worst-case throughput constant, for example, a low-path-length variance routing algorithm could be designed. The options in reducing injection variation are less involved, but a simple approach is to ensure each source injects packets into the fabric round-robin by destination. That is, packets to the same destination are interleaved evenly with packets to other destinations. Of course, for permutation patterns, this approach does not affect the injection because all packets at a source are all for a single destination.

4.5 Summary

As illustrated in this chapter, flow control plays a critical role in determining the ultimate efficiency of a distributed fabric. For small IP packets, control overheads tend to dominate. We introduced the idea of variable-size flits to address the problem and, for a representative design example, variable-size flits reduced control overhead by 26.7% over a common fixed-size flit approach and by 18.0% over the two-flit-size approach used in the Avici TSR router fabric. Since the size of the flits only needs to vary over a small range, their hardware implementation required only minor changes to a typical fixed-flit microarchitecture.

Long IP packets were also shown to be problematic. If flit-buffer flow control is used, these packets can be spread across many routers. This, in turn, creates a channel coupling — a packet has to gain access to several channels to make progress through the network. The more resources that are coupled together, the more difficult it is for a packet to progress. For example, we showed an example where throughput was reduced by over 50% due to long-packet effects. The solution is to split these long IP packets into many small fabric packets, each approximately the size of a minimum-size IP packet. When used with

variable-size flits, this does not introduce any additional overhead in the worst case.

Splitting long packets along with the typical ordering requirements of an IP router requires some reordering mechanism at the outputs. We described a simple window-based scheme and derived expressions for both the reordering latency and reorder buffer size. Based on these expressions, we suggested several possibilities for reducing the amount of reordering that occurs in a fabric.

The ideas of this chapter can be combined to create some simple design guidelines for efficient flow control in a distributed fabric. Split long IP packets into fabric packets whose length is between P_{min} and $2P_{\text{min}}$. When used with variable-size flits, this minimizes any long-packet effects without introducing extra control overhead. Let the range of variable-size flits also be from P_{min} to $2P_{\text{min}}$, if allowed by the fabric router area constraints. This minimizes control overhead. At the same time, it also eliminates the need for differentiating flits and packets — each fabric packet can fit into a single, variable-size flit. This shifts the fabric to packet-buffer flow control [26], also known as virtual cut-through flow control [38]. While modern networks with a limited range of packet sizes or a small number of virtual channels often use virtual cut-through flow control, such as the Alpha 21364 [46], our results are in contrast to the Avici TSR router fabric [24], which uses flit-buffer flow control. As we have shown, virtual cut-through flow control is a better design choice for distributed fabrics and other throughput-oriented interconnection networks.

Chapter 5

Conclusion

This thesis has shown that the line-rate guarantees of a distributed fabric are closely tied to the routing algorithm used for that packet. For a particular fabric, these guarantees can be found by solving a series of maximum-cost flow problems that find a worst-case traffic pattern. Building on this idea, we also showed that worst-case optimal oblivious routing algorithms can be found using convex programs. These results combined with the efficient flow control techniques developed in Chapter 4 give the IP router's designer a lot of flexibility in the organization of a router's fabric — this allows the fabric itself to be optimized for cost, packaging, or any other design variable.

As we stated in the introduction, line-rate service is just the vanilla flavor of router services and today's routers are typically asked to offer many more services. In Section 5.1, we briefly explore this avenue along with the behavior of a router under inadmissible traffic patterns. Current technology requires the complexity of the fabric routers to scale with the number of IP router ports times the number of traffic classes when supporting advanced services. However, this limits scalability and we present an alternative approach were packet delivery is separated from packet admission as an interesting area of future work.

We close by considering the ways in which a distributed router fabric could take advantage of the structure of Internet traffic. Our goal throughout this thesis research has been to guarantee a particular line-rate over all possible traffic patterns and this is the current practice in industry and almost all academic work. However, this worst-case design mentality is expensive. Rather, if we could take advantage of predictability, while still maintaining

robust operation in the face of some variation, router cost could be reduced. We explore this possibility in Section 5.2.

5.1 Traffic admission and quality of service

One of the important aspects of IP router design we did not focus on in this thesis is the situation where the incoming traffic is inadmissible. For example, all incoming traffic might be destined to output port 1 of the IP router due to a temporary IP routing table misconfiguration.[1] To satisfy our requirement for line-rate service, we only need to make sure the overloaded output port continues to deliver packets at its full output rate. This necessarily leaves a large fraction of the incoming packets unserved — there simply is not enough bandwidth at the single output to serve all the inputs simultaneously.

In general, an IP router has two responsibilities when an output becomes overloaded. First, the traffic destined to the overloaded output must be isolated from traffic destined to other outputs. This is analogous to avoiding head-of-line blocking in a traditional input-queued crossbar and is often called tree saturation [52] in the context of interconnection networks. In current distributed fabrics, specifically the Avici TSR [23], this isolation is achieved by using a non-interfering network. That is, each fabric output is assigned a separate virtual channel, which ensures isolation between outputs.

The second responsibility of an IP router is to select which packets get access to the overloaded output by applying a quality of service policy. For example, high priority traffic might be served before lower priority traffic or, in the case of best-effort traffic, each source vying for the overloaded output might get an equal share of its bandwidth. Currently, these polices are enacted using a combination of approaches: prioritized arbitrations within the fabric routers, per-class virtual channels, and end-to-end techniques.

The current approaches for providing traffic isolation and quality of service are workable and are in use, but, ultimately, have limited scalability. Perhaps the biggest obstacle is that supporting per-class and per-output virtual channels requires the complexity of the

[1] The IP routing tables define the routing between IP routers and should not be confused with the fabric routing tables discussed in Chapter 3.

fabric routers to grow with the number of traffic classes times the number of outputs. Ideally, the fabric routers would have a complexity that is largely independent of the size of overall IP router, thus maximizing scalability.

A powerful alternative to the current approach may be to separate the tasks of packet delivery and packet admission. Here, the fabric is solely focused on packet delivery — it makes no contingency for either traffic isolation or quality of service, it simply delivers the packets that are presented to it. This keeps the fabric routers extremely simple. Packet admission is then determined by end-to-end protocols (from fabric inputs to fabric outputs). To be scalable, these protocols must be distributed and can be thought of as a continuous negotiation process between the inputs and outputs to determine which packets can be admitted to the fabric such that no outputs become overloaded and the quality of service guarantees are met.

As an example, consider a batching approach for admission. For simplicity, we will assumed fixed size packets. At the beginning of each batch, a matrix B holds the packets that can be admitted during that batch: the integer entries b_{ij} indicate the number of packets that can transferred across the fabric from input i to output j during the batch. Each batch lasts T packet times and B can be thought of as the traffic matrix for the batch. By overlapping the forwarding of packets from one batch with computation of the next batch, the fabric is kept busy. The task of computing the next batch falls upon the end-to-end admission algorithm. So that each batch is admissible, the admission algorithm should ensure that each row and column sum of B is at most T.

There are many possible criteria, such as fairness or packet priority, an algorithm could use in selecting the packets for the next batch. Algorithm 2 shows one such example. For this algorithm, each input tracks the number of packets it has queued for each of the fabric outputs, which is stored in $q^{(0)}$. Then, the batch is built iteratively — the goal of the loop in Step 2 is to compute an admissible traffic matrix (row and column sums of one) based on the size of the queues. During this loop the batch variables q and t will not necessarily be integers, but the matrix is scaled and rounded to integer values in the final step of the algorithm. The first step of each loop is performed at the inputs. Each input computes the Euclidean projection, denoted $P(\cdot)$, of the vector of its queue sizes (the rows of the traffic matrix) onto the probability distribution with the same number of elements. The results of

this projection are transferred (across the fabric) to the outputs. The outputs then compute the projection of the columns of the traffic matrix and the process is repeated.

Algorithm 2 End-to-end admission

1. *Initialize.* q_{ij}^0 is set to the number of packets waiting at input i destined to output j. $\delta_{ij}^{(0)} \leftarrow 0$ and $\xi_{ij}^{(0)} \leftarrow 0$ for all $i, j \in \mathcal{N}$. $k \leftarrow 0$.
2. *Iterate projections.* While $k <$ MAXITER,
   (a) $k \leftarrow k + 1$.
   (b) *Project at inputs.*
   $$\left[t_{i1}^{(k)} \dots t_{iN}^{(k)}\right] \leftarrow P\left(\left[q_{i1}^{(k-1)} - \delta_{i1}^{(k-1)} \dots q_{iN}^{(k-1)} - \delta_{iN}^{(k-1)}\right]\right), \qquad \forall i \in \mathcal{N},$$
   $$\delta_{ij}^{(k)} \leftarrow t_{ij}^{(k)} - q_{ij}^{(k-1)}, \qquad \forall i, j \in \mathcal{N}.$$
   (c) *Distribute $t^{(k)}$ to outputs.* $t_{11}^{(k)}, t_{21}^{(k)}, \dots, t_{N1}^{(k)}$ go to output 1, etc.
   (d) *Project at outputs.*
   $$\left[q_{1j}^{(k)} \dots q_{Nj}^{(k)}\right] \leftarrow P\left(\left[t_{1j}^{(k)} - \xi_{Nj}^{(k-1)} \dots t_{1j}^{(k-1)} - \xi_{Nj}^{(k-1)}\right]\right), \qquad \forall j \in \mathcal{N},$$
   $$\xi_{ij}^{(k)} \leftarrow q_{ij}^{(k)} - t_{ij}^{(k)}, \qquad \forall i, j \in \mathcal{N}.$$
   (e) *Distribute $q^{(k)}$ to inputs.* $q_{11}^{(k)}, q_{12}^{(k)}, \dots, q_{1N}^{(k)}$ go to input 1, etc.
3. *Create batch.* $b_{ij} \leftarrow \left\lfloor T \cdot q_{ij}^{(k)} \right\rfloor, \quad \forall i, j \in \mathcal{N}.$

While this iteration may seem at best arbitrary, it is actually computing the Euclidean projection of the original queue sizes onto a matrix with row and column sums of one.[2] The Euclidean projection closely resembles the maximum weight-matching of the queues and, combined with the result of Shah and Kopikare [59], selection of batches using Algorithm 2 achieves a throughput arbitrarily close to the maximum as T and the number of loop

[2]There are many details that are not presented here, but the iteration is essentially Cheney and Goldstein's alternating projections algorithm [19]. With the addition of the corrective terms δ and ξ, as suggested by Boyle and Dystrka [16], we are ensured that the iteration converges to the Euclidean projection.

iterations are increased assuming the fabric can continuously deliver admissible batches.

We present this algorithm not to make any argument about its practical utility, but simply as an illustration of an end-to-end admission algorithm. Other approaches are certainly possible. The idea of core-stateless fair queuing developed by Stoica et al. [65] shares a similar goal and Charny and Ramakrishnan [18] present a distributed method for computing max-min fair admissions in a distributed way, although a direct implementation of their approach would likely result in too much complexity in the fabric routers. Finally, a more robust and responsive version of our admission algorithm could possibly be built around the ideas of online optimization or control theory.

Whatever the approach, there are certainly many interesting questions to be explored regarding both the advantages of separating packet admission from packet delivery and the design of admission algorithms.

5.2 Taking advantage of traffic structure

Throughout this thesis we focused on providing line-rate service in a distributed fabric and, thus, the set of admissible traffic patterns consisted of those whose total input and output rate (row and column sums) where at most the line rate. Today's router manufacturers and researchers have focused their efforts on the same criterion. However, a question that few router designers have yet considered is how we might take advantage of the predictable structure of Internet traffic. While there is certainly a significant stochastic component in this traffic, large-scale measurement studies, such as the one by Roughan et al. [56], reveal significant predictability. To see how the ideas of a distributed fabric, in particular, can take advantage of this, consider an example of routing traffic through one of an ISP's points of presence (POP).

Figure 5.1 shows an example POP topology with core IP routers interconnected by optical links. The links at the boundary of the POP are backbone links that are connected to other POPs. First, there is no reason that this entire collection of IP routers cannot be treated as a distributed fabric — the inputs and outputs of the fabric are the boundary links and the fabric routers are the core IP routers. Furthermore, assume that the predicted traffic pattern for this POP is given by $\tilde{\Lambda}$. Then, we could apply the same routing algorithm ideas

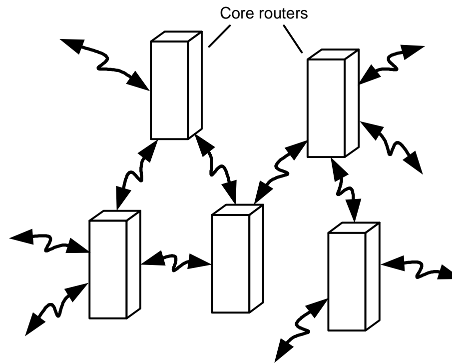


Figure 5.1: An example point of presence topology. (Not meant to be realistic.)

developed in Chapter 3, but adjust the set of admissible traffic patterns to be defined as

$$\mathcal{T} = \left\{ \Lambda \;\middle|\; ||\Lambda - \tilde{\Lambda}|| \leq \epsilon \right\},$$

for example. That is, we design a robust routing algorithm to maximize throughput over the set of traffic patterns whose distance from the predicted pattern is at most ϵ. This set of traffic patterns is convex for any norm and, therefore, the resulting routing algorithm design problem is a convex program and can be solved efficiently. Of course, many other distance or uncertainty measures could be developed and we expect that most reasonable measures would be convex.

Applegate and Cohen [4] have explored similar ideas for routing in POPs using a slightly different objective than our worst-case objective. Kodialam et al. [40] also investigate robust routing in POPs, but limit their study to the set of admissible traffic patterns with row and column sums equal to the line rate. However, an optimal routing algorithm just one possible advantage of operating what is traditionally treated as a collection of individual routers as a single distributed router fabric. It would also allow the elimination of many line card interfaces — the routers within a POP could communicate using the simple, low-overhead techniques of interconnection networks. Repeated packet processing, admission control, and statistics gathering could all be eliminated. The main challenge would be developing the distributed algorithms to control this large distributed fabric and provide the same service guarantees expected from today's IP routers. Still, the potential advantages of

such an approach may it an interesting line of future research.

Appendix A

Notation and definitions

All the topologies (graphs) in this thesis are assumed to be directed. In the context of networks, edges of the topologies are called *channels* and vertices are called *nodes*. A channel c can also be referred to using its endpoints $c = (i, j)$, where channel c is connected from node i to node j.

Multi-dimensional variables are referred to in capitals and their scalar elements as lower-case with subscripts. For example, the routing probability x_{kij} is an element of the multi-dimensional variable X.

The notation $[C]$ is used to indicate a function that is one if the condition C is true and zero otherwise. For example, $[x = y]$ is one if $x = y$ and is zero otherwise.

Symbol	Definition	Units
b_c	bandwidth of channel c	bits/s
b_{d_j}	ejection bandwidth of destination j	bits/s
b_{s_i}	injection bandwidth of source i	bits/s
$C(\mathcal{C})$	number (set) of channels in a fabric	
F	flit size	bits
F_0	per-flit control overhead	bits
$N(\mathcal{N})$	number (set) of fabric ports	
p_{kij}	probability of using path k when routing from source i to destination j	

P	packet size	bits
P_0	per-packet control overhead	bits
$\mathcal{T}$	a set of traffic patterns	
x_{cij}	probability of using channel c when routing from source i to destination j	
$\gamma_c(X, \Lambda)$	channel load — amount of data crossing channel c when routing traffic pattern Λ with routing algorithm X.	bits/s
Λ	traffic pattern — entry λ_{ij} is the average amount of traffic from source i to destination j.	bits/s
Θ	fraction of injected throughput	

Appendix B

A subgradient method for the WCORDP

In Section 3.2.2, a basic subgradient method for the worst-case optimal routing design problem (WCORDP) was introduced. This appendix details some of our experiences in developing a more robust and flexible subgradient method. While our custom code underperformed the CPLEX [34] commercial linear programming package, we believe there are some interesting aspects to our approach. Beyond the work presented here, perhaps the most promising direction for improvements in the solution time of the WCORDP is along the lines of the primal-dual methods for the maximum concurrent flow problem [28, 30, 60].

One of the important considerations of any iterative optimization method is the stopping criteria — both the number of iterations to be run and the guarantees on the quality of the solution once these iterations have been run. For subgradient methods, these issues are closely related to step size selection. Several approaches to selecting the step size give theoretical convergence to an optimal solutions. For example, letting the step size

$$\alpha^{(k)} = \frac{1}{k||g^{(k)}||_2}, \tag{B.1}$$

where k is the iteration number and g is the subgradient used to update the current solution, ensures convergence to an optimal solution. In practice, this convergence can be painfully slow. Moreover, any constant multiple of (B.1) correctly converges, so an additional problem of selecting this constant factor is introduced.

Of all the methods for selecting step size, including (B.1), we found that using

$$\alpha^{(k)} = \frac{f^{(k)} - f^*}{||g^{(k)}||_2^2}, \tag{B.2}$$

produced consistently good convergence. Here, $f^{(k)}$ is the value of the objective at the current iteration and f^* is the optimal value of the objective. While this step size eliminates the problem of selecting constant factors, it introduces another issue. Specifically, we do not know the optimal value f^* until the optimization is complete. To avoid this problem, we reformulate our optimization to solve both the primal and dual simultaneously.[1]

Our formulation of the WCORDP combines both (3.5) and (3.10) to yield

$$\begin{array}{lll}
\text{minimize} & \max_{\Lambda \in \mathcal{T}} \max_{c \in C} \sum_{i,j \in \mathcal{N}} \lambda_{ij} x_{cij}/b_c - \sum_{i,j \in \mathcal{N}} \min_{\substack{\text{paths } k \\ \text{from } i \text{ to } j}} \sum_{\substack{\text{channels } c \\ \text{in path } k}} \lambda_{cij}/b_c & \\
& \sum_{\{l | (k,l) \in \mathcal{C}\}} x_{(k,l),i,j} - \sum_{\{l | (l,k) \in \mathcal{C}\}} x_{(l,k),i,j} = [k = i] - [k = j] & \forall i,j,k \in \mathcal{N} \\
& x_{cij} \geq 0, & \forall i,j \in \mathcal{N},\ c \in \mathcal{C} \\
& \sum_{j \in \mathcal{N}} \lambda_{cij} \leq \phi_c b_{s_i}, & \forall i \in \mathcal{N},\ c \in \mathcal{C}, \\
& \sum_{i \in \mathcal{N}} \lambda_{cij} \leq \phi_c b_{d_j}, & \forall j \in \mathcal{N},\ c \in \mathcal{C}, \\
& \lambda_{cij} \geq 0, & \forall i,j \in \mathcal{N},\ c \in \mathcal{C}, \\
& \sum_{c \in \mathcal{C}} \phi_c = 1. &
\end{array} \tag{B.3}$$

The objective is the worst-case channel load minus the lower bound from the dual formulation of (3.10), so the optimal value of the objective is always zero and (B.2) can be used to determine the step size ($f^* = 0$). In addition, each step of the optimization produces both an upper bound and lower bound from the primal and dual solutions, respectively.

Solving (B.3) is almost the same as solving the primal and dual problems independently — the only coupling comes in the objective function. The total subgradient is simply the sum of the subgradients due to the primal and dual components of the problem. This only leaves the computation of the projection to complete our subgradient method.

The projection step could be performed by solving a quadratic optimization problem,

[1]Although we are solving the primal and dual simultaneously, our approach is not what is referred to as a primal-dual algorithm.

but the complexity of this problem is on the order of the complexity of our WCORDP. Instead, we use Boyle and Dykstra's algorithm [16] for computing the projection. Before we describe the algorithm, without loss of generality we can assume that all the constraints of the WCORDP are in the form

$$a_i^T y \leq b_i, \quad i = 1, \ldots, M,$$

where M is the total number of constraints, $||a_i|| = 1$ for all i, and y is a vector of all the optimization variables (X and Λ). The equality constraints can be replaced by two inequalities or relaxed to an inequality. For example, it can be shown that the value of the optimization is not changed if $\sum_{c \in C} \phi_c = 1$ is replaced by $\sum_{c \in C} \phi_c \leq 1$.

Boyle and Dykstra's algorithm for these linear constraints is shown as Algorithm 3. The algorithm cyclically applies projections onto the halfspaces where

$$P_i(y) = \begin{cases} y & \text{if } a_i^T y \leq b_i \\ y - (a_i^T y - b_i) a_i & \text{otherwise} \end{cases}$$

is the projection onto the ith subspace [1, 14]. Without the corrective terms δ, the algorithm would simply be an alternating projection [19], but the addition of the corrective terms ensure that the iterates converge to a Euclidean projection of y onto the intersection of halfspaces defined by the constraints. Furthermore, in the WCORDP each of the constraints involves only a few variables, so the projections are very simple to compute. The overall algorithm converges quickly (see Perkins [51] for specific results) and, in practice, 10–20 iterations of the outer loop are sufficient.

The performance of the resulting subgradient method is shown in Figures B.1 and Figure B.2 for a simple 4-ary 2-cube (16-node torus) network. Figure B.1 shows the optimality gap and the expected (sublinear) convergence of the subgradient method. This slow convergence is the ultimate limiter of the subgradient method in our experience — to reduce the optimality gap by a factor of 1000, for example, 10^6 iterations are required. Still, the subgradient method may still have some practical utility in large networks where "tuning" of the routing algorithm, instead of optimization from an arbitrary starting point, is required.

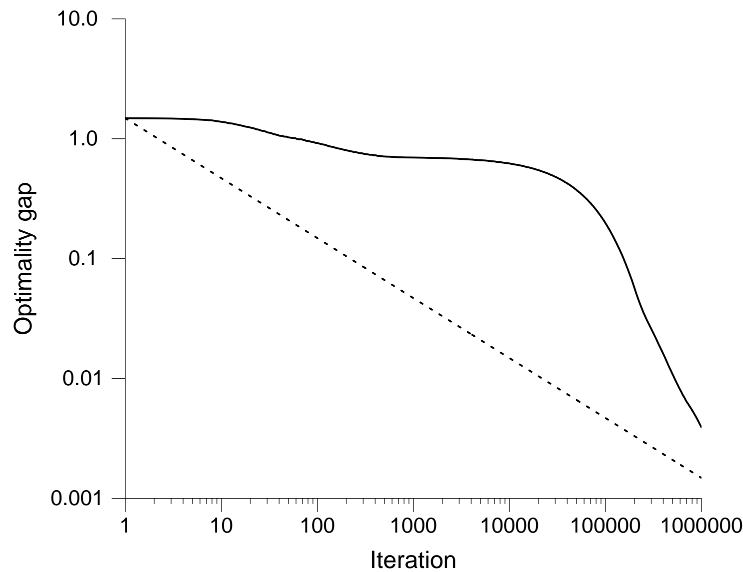


Figure B.1: Optimality gap vs. iteration for the subgradient method in a 4-ary 2-cube. The expected asymptotic convergence of $k^{-1/2}$, where k is the iteration number, is shown as a dotted line. (See Bertsekas [7] for an analysis of the convergence.)

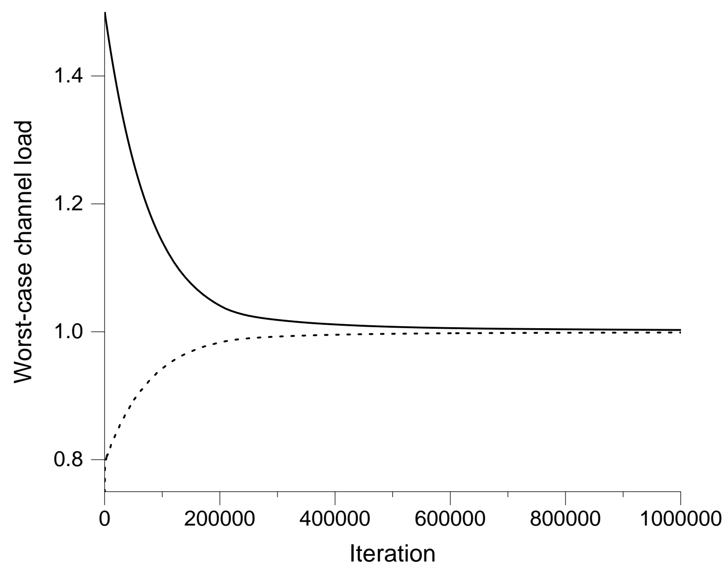


Figure B.2: Primal (solid line) and dual (dotted line) solutions vs. iteration for the subgradient method in a 4-ary 2-cube.

Algorithm 3 Projection onto an intersection of halfspaces

1. *Initialize.* $y^{(1,0)}$ is initialized to the value of y before projection, $\delta^{(0,i)} \leftarrow 0$ for $i = 1, \ldots, M$, and $k \leftarrow 1$.

2. *Loop until converged.* While $\max_{i=1,\ldots,M}(a_i^T y^{(k,0)} - b_i) > \epsilon$,

   (a) *Loop over projections.* For $i = 1$ to M,

   i. *Project onto the ith halfspace.*

   $$y^{(k,i)} \leftarrow P_i \left(y^{(k,i-1)} - \delta^{(k-1,i)} \right).$$

   ii. $\delta^{(k,i)} \leftarrow y^{(k,i)} - y^{(k,i-1)}$.

   (b) $k \leftarrow k + 1$ and $y^{(k,0)} \leftarrow y^{(k-1,M)}$.

Bibliography

[1] S. Agmon. The relaxation method for linear inequalities. *Canadian Journal of Mathematics*, 6:382–392, 1954.

[2] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network flows: Theory, algorithms, and applications*. Prentice-Hall, Inc., Upper Saddle River, NJ, 1993.

[3] W. A. Aiello, F. T. Leighton, B. M. Maggs, and M. Newman. Fast algorithms for bit-serial routing on a hypercube. *Mathematical Systems Theory*, 24(4):253–271, 1991.

[4] D. Applegate and E. Cohen. Making intra-domain routing robust to changing and uncertain traffic demands: understanding fundamental tradeoffs. In *Proc. of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 313–324, 2003.

[5] Y. Azar, E. Cohen, A. Fiat, H. Kaplan, and H. Räcke. Optimal oblivious routing in polynomial time. In *Proc. of the ACM Symposium on the Theory of Computing*, pages 383–388, 2003.

[6] D. Bertsekas and R. Gallager. *Data Networks*. Prentice-Hall, Inc., Upper Saddle River, NJ, second edition, 1992.

[7] D. P. Bertsekas. *Nonlinear Programming*. Athena Scientific, Belmont, MA, second edition, 1999.

[8] D. P. Bertsekas, A. Nedic, and A. E. Ozdalgar. *Convex analysis and optimization*. Athena Scientific, Belmot, MA, 2004.

[9] G. Birkhoff. Tres observaciones sobre el algebra lineal. *Univ. Nac. Tucumán Rev. Ser. A*, 5:147–151, 1946.

[10] E. Blanton and M. Allman. On making TCP more robust to packet reordering. *ACM Computer Communication Review*, 32(1):20–30, January 2002.

[11] K. Bolding, M. Fulgham, and L. Snyder. The case for chaotic adaptive routing. *IEEE Trans. on Computers*, 46(12):1281–1292, December 1997.

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

[13] A. Borodin, P. Raghavan, B. Schieber, and E. Upfal. How much can hardware help routing? *Journal of the ACM*, 44(5):726–741, 1997.

[14] S. Boyd and J. Dattorro. Alternating projections. `http://www.stanford.edu/class/ee392o/alt_proj.pdf`, 2003.

[15] S. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.

[16] J. P. Boyle and R. L. Dykstra. A method for finding projections onto the intersection of convex sets in Hilbert spaces. In *Advances in Order Restricted Inference 37*, Lecture notes in statistics, pages 28–47. Springer, 1985.

[17] D.-S. Lee C.-S. Chang and Y.-S. Jou. Load balanced Birkhoff-von Neumann switches, part i: one-stage buffering. *Computer Communications*, 25(6):611–622, April 2002.

[18] A. Charny and K. K. Ramakrishnan. Time scale analysis of explicit rate allocation in ATM networks. In *Proc. of IEEE INFOCOM*, pages 1182–1189, March 1996.

[19] W. Cheney and A. A. Goldstein. Proximity maps for convex sets. *Proc. of the AMS*, 10(3):448–450, 1959.

[20] Cisco Systems Inc. Cisco carrier routing system, May 2004.

[21] C. Clos. A study of non-blocking switching networks. *Bell System Technical Journal*, 32:406–424, 1953.

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

[23] W. J. Dally, P. P. Carvey, and L. R. Dennison. The Avici terabit switch/router. In *Conference Record of Hot Interconnects 6*, pages 41–50, August 1998.

[24] W. J. Dally, P. P. Carvey, L. R. Dennison, and A. P. King. Internet switch router. United States Patent 6,370,145, April 2002.

[25] W. J. Dally and C. L. Seitz. Deadlock free message routing in multiprocessor interconnection networks. *IEEE Trans. on Computers*, 36(5):547–553, May 1987.

[26] W. J. Dally and B. Towles. *Principles and practices of interconnection networks*. Morgan Kaufmann, San Francisco, CA, 2004.

[27] R. Durstenfeld. Algorithm 235: Random permutation. *Communications of the ACM*, 7(7):420, 1964.

[28] L. K. Fleischer. Approximating fractional multicommodity flow independent of the number of commodities. *SIAM Journal on Discrete Mathematics*, 13(4):505–520, 2000.

[29] L. Fratta, M. Gerla, and L. Kleinrock. The flow deviation method — an approach to store-and-forward communication network design. *Networks*, 3:97–133, 1973.

[30] N. Garg and J. Könemann. Faster and simpler algorithms for multicommodity flow and other fractional packing problems. In *Proc. of IEEE Symposium on Foundations of Computer Science*, pages 300–309, Palo Alto, CA, November 1998.

[31] R. Graham, D. E. Knuth, and O. Patashnik. *Concrete mathematics*. Addison-Wesley, 2nd edition, 1994.

[32] A. K. Gupta, W. J. Dally, A. Singh, and B. Towles. Scalable opto-electronic network (SOEnet). In *Proc. of Hot Interconnects*, pages 71–76, Stanford, CA, August 2002.

[33] IEEE std. 802.3 — Part 3: carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications, 2002.

[34] ILOG, Mountain View, CA. *CPLEX 7.5 User's Manual*. `http://www.ilog.com`.

[35] K. E. Iverson. *A programming language*. Wiley, 1962.

[36] S. Iyer and N. W. McKeown. Analysis of the parallel packet switch architecture. *IEEE/ACM Trans. on Networking*, 11(2):314–324, 2003.

[37] C. Kaklamanis, D. Krizanc, and A. Tsantilas. Tight bounds for oblivious routing in the hypercube. In *Proc. of the Symposium on Parallel Algorithms and Architectures*, pages 31–36, 1990.

[38] P. Kermani and L. Kleinrock. Virtual-cut through: a new computer communications switching technique. *Computer Networks*, 3(4):267–286, 1979.

[39] I. Keslassy. *The load-balanced router*. PhD thesis, Stanford University, 2004.

[40] M. Kodialam, T. V. Lakshman, and S. Sengupta. Efficient, robust routing in highly dynamic environments. Presented at the *Stanford Workshop on Load-balancing*, May 2004.

[41] D. Krizanc, D. Peleg, and E. Upfal. A time-randomness tradeoff for oblivious routing. In *Proc. of the ACM Symposium on the Theory of Computing*, pages 93–102, Chicago, IL, 1988.

[42] H. Kuhn. The Hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2:83–97, 1955.

[43] N. McKeown. Personal communication, December 2003.

[44] N. McKeown, V. Anatharam, and J. Warland. Achieving 100% throughput in an input-queued switch. In *Proc. of IEEE INFOCOM*, pages 296–302, San Francisco, CA, 1996.

[45] M. Mitzenmacher. A brief history of generative models for power law and lognormal distributions. *Internet Mathematics*, 1(2):226–251, 2003.

[46] S. S. Mukherjee, P. Bannno, S. Lang, A. Spink, and D. Webb. The Alpha 21364 network architecture. In *Proc. of Hot Interconnects*, pages 113–117, Stanford, CA, August 2001.

[47] S. S. Mukherjee, F. Silla, P. Bannon, J. Emer, S. Lang, and D. Webb. A comparative study of arbitration algorithms for the Alpha 21364 pipelined router. In *Proc of. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 223–234, San Jose, CA, October 2002.

[48] 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.

[49] A. M. Odlyzko. Internet traffic growth: Sources and implications. In B. B. Dingel, W. Weiershausen, A. K. Dutta, and K.-I. Sato, editors, *Optical Transmission Systems and Equipment for WDM Networking II*, volume 5247 of *Proc. of SPIE*, pages 1–15. SPIE, August 2003.

[50] F. Ros Peran. *Routing to minimize the maximum congestion in a communication network*. PhD thesis, MIT, 1978.

[51] C. Perkins. A convergence analysis of Dykstra's algorithm for polyhedral sets. *SIAM Journal on Numerical Analysis*, 40(2):792–804, 2002.

[52] G. F. Pfister and V. A. Norton. Hot spot contention and combining in multistage interconnection networks. *IEEE Trans. on Computers*, 34(10):943–948, Oct. 1985.

[53] J. Postel. RFC 791: Internet Protocol, September 1981.

[54] H. Räcke. Minimizing congestion in general networks. In *Proc. of IEEE Symposium on Foundations of Computer Science*, pages 43–52, Vancouver, Canada, 2002.

[55] P. Raghavan and C. Thompson. Randomized rounding. *Combinatorica*, 7:365–374, 1987.

[56] M. Roughan, A. Greenberg, C.Kalmanek, M. Rumsewicz, J. Yates, and Y. Zhang. Experience in measuring backbone traffic variability: Models, metrics, measurements and meaning. In *Internet Measurement Workshop*, November 2002.

[57] 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, Aug. 1996.

[58] F. Shafai, K. J. Schultz, G. F. R. Gibson, A. G. Bluschke, and D. E. Somppi. Fully parallel 30-MHz, 2.5-Mb CAM. *IEEE Journal of Solid-State Circuits*, 33(11):1690–1696, November 1998.

[59] D. Shah and M. Kopikare. Delay bounds for approximate maximum weight matching algorithms for input queued switches. In *Proc. of IEEE INFOCOM*, pages 1024–1031, New York, NY, June 2002.

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

[61] A. Singh, W. J. Dally, A. K. Gupta, and B. 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.

[62] A. Singh, W. J. Dally, A. K. Gupta, and B. Towles. Adaptive channel queue routing on k-ary n-cubes. In *Proc. of the Symposium on Parallel Algorithms and Architectures*, Barcelona, Spain, June 2004.

[63] A. Singh, W. J. Dally, B. Towles, and A. 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.

[64] A. Srinivasan. Improved approximations of packing and covering problems. In *Proc. of the ACM Symposium on the Theory of Computing*, pages 268–276, 1995.

[65] I. Stoica, S. Shenker, and H. Zhang. Core-stateless fair queueing: Achieving approximately fair bandwidth allocations in high speed networks. In *Proc. of ACM SIGCOMM*, pages 118–130, September 1998.

[66] 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.

[67] Y. Tamir and G. L. Frazier. High performance multi-queue buffers for VLSI communication switches. In *Proc. of the International Symposium on Computer Architecture*, pages 343–354, June 1988.

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

[69] B. Towles and W. 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.

[70] B. Towles, W. J. Dally, and S. P. Boyd. Throughput-centric routing algorithm design. In *Proc. of the Symposium on Parallel Algorithms and Architectures*, pages 200–209, San Diego, CA, June 2003.

[71] 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.

[72] J. von Neumann. A certain zero-sum two-person game equivalent to the optimal assignment problem. *Contributions to the Theory of Games*, 2:5–12, 1953.