Louisiana State University
LSU Digital Commons

LSU Doctoral Dissertations | Graduate School

2009

Scheduling and reconfiguration of interconnection network switches

Krishnendu Roy
Louisiana State University and Agricultural and Mechanical College, kroy1@ece.lsu.edu

Follow this and additional works at: https://digitalcommons.lsu.edu/gradschool_dissertations

Part of the Electrical and Computer Engineering Commons

Recommended Citation
Roy, Krishnendu, "Scheduling and reconfiguration of interconnection network switches" (2009). *LSU Doctoral Dissertations*. 2842.
https://digitalcommons.lsu.edu/gradschool_dissertations/2842

This Dissertation is brought to you for free and open access by the Graduate School at LSU Digital Commons. It has been accepted for inclusion in LSU Doctoral Dissertations by an authorized graduate school editor of LSU Digital Commons. For more information, please contactgradetd@lsu.edu.

SCHEDULING AND RECONFIGURATION OF INTERCONNECTION NETWORK SWITCHES

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The Department of Electrical and Computer Engineering

by
Krishnendu Roy
B.Sc., University of Calcutta, India, May 2000
B.Tech., University of Calcutta, India, May 2003
M.S., Louisiana State University, USA, December 2005
August 2009

Acknowledgments

I would like to thank all the members of my final examination committee – Dr. Jagannathan Ramanujam, Dr. David Koppelman, Dr. Brygg Ullmer, and Dr. Ambar Sengupta, and the members of my general examination committee – Dr. Suresh Rai and Dr. Rajgopal Kannan who could not be in my final examination committee. I would also like to thank John Bordelon for all his help related to executing the simulations.

I consider myself very fortunate to be able to make so many great friends during my doctoral study at LSU, and I want to thank all of them as well. Their friendship and support has meant a lot to me. I want to recognize the support and understanding that my family has shown to me. Thank you for your unwavering confidence in me.

Finally, I want to express my sincere gratitude to my advisors Dr. Ramachandran Vaidyanathan and Dr. Jerry L. Trahan. This dissertation is just a fraction of what I have learnt from interacting with them. Without their support, time, guidance, and patience this work would have been impossible. Thanks for everything.

Table of Contents

Acknowledgments ii

List of Tables v

List of Figures vi

Abstract viii

Chapter 1: Introduction 1
- 1.1 Crossbar-Based Input-Queued Switch 3
- 1.2 Circuit-Switched Tree 6
- 1.3 Fat-Tree Switch 7

Chapter 2: Crossbar-Based Switches: Conditions for Logarithmic Delay 9
- 2.1 Introduction and Background 9
- 2.2 Slots, Rounds and Frames 14
- 2.3 Stability and Delay 17
- 2.4 Necessary Conditions for Logarithmic Delay 19
  - 2.4.1 Uniform Random Traffic 20
  - 2.4.2 Bursty Traffic 22
- 2.5 Simulation Results 23
  - 2.5.1 Uniform Random Traffic 25
  - 2.5.2 Bursty Traffic 31
- 2.6 Summary 39

Chapter 3: Fast Scheduling Algorithm on Mesh-of-Trees 41
- 3.1 Introduction 41
- 3.2 Kelsen's $O(\log^3 n)$ Bipartite Matching Algorithm on the PRAM 44
- 3.3 Reconfigurable Mesh Preliminaries 46
- 3.4 R-Mesh Bipartite Matching Algorithm 50
- 3.5 Time Complexity 55
  - 3.5.1 Other Considerations 56
- 3.6 Summary 57

Chapter 4: Scheduling and Configuration of the Circuit-Switched Tree 58
- 4.1 CST Background 58
  - 4.1.1 Structure of the CST 60
  - 4.1.2 Communications on a CST 61
  - 4.1.3 CST Configuration 63
- 4.2 The Configuration Algorithm for Width-w Communication Sets 64
- 4.3 Width-1 Communication Sets 71
  - 4.3.1 Modified CST Configuration Algorithms for Width-1 Sets 72

4.3.2 Oriented, Well-nested, Width-1 Point-to-Point Communication Sets CST Configuration Algorithm . . . 74
4.3.3 Width-1 Multicast Sets . . . 77
4.4 Well-Nested, Width-w Communication Sets . . . 81
4.4.1 Algorithm Adaptation . . . 82
4.4.2 Correctness of Phases 1 and 2 . . . 83
4.4.3 Proof of Optimality . . . 85
4.5 Summary . . . 87

Chapter 5: Routing Algorithm for an R-Mesh Based Fat-Tree Switch . . . 88
5.1 Introduction . . . 88
5.2 Routing Algorithm for a Fat-Tree Switch Implemented as an R-Mesh . . . 90
5.2.1 Examples Illustrating the Algorithm . . . 94
5.3 Summary . . . 97

Chapter 6: Summary of Results and Open Problems . . . 98
6.1 Crossbar-Based Input-Queued Switches . . . 98
6.2 Fast Scheduling Algorithm on Mesh-of-Trees . . . 99
6.3 Circuit-Switched Tree Switches . . . 101
6.4 Fat-Tree Switch . . . 102
6.5 Other Directions . . . 103

References . . . 105

Vita . . . 111

List of Tables

2.1 VOQ occupancy as observed by an incoming packet for $pps = 1$. 26

2.2 VOQ occupancy as observed by an incoming packet for $pps = 2$. 28

2.3 VOQ occupancy as observed by an incoming packet for $pps = 3$. 28

2.4 VOQ occupancy as observed by an incoming packet for $pps = 4$. 29

2.5 VOQ occupancy as observed by an incoming packet for $pps = 5$. 29

2.6 VOQ occupancy as observed by an incoming packet for $pps = 1$ and $b = 3$. 34

2.7 VOQ occupancy as observed by an incoming packet for $pps = 2$ and $b = 3$. 34

2.8 VOQ occupancy as observed by an incoming packet for $pps = 3$ and $b = 3$. 34

2.9 VOQ occupancy as observed by an incoming packet for $pps = 4$ and $b = 3$. 34

2.10 VOQ occupancy as observed by an incoming packet for $pps = 5$ and $b = 3$. 35

2.11 VOQ occupancy as observed by an incoming packet for $pps = 1$ and $b = 6$. 35

2.12 VOQ occupancy as observed by an incoming packet for $pps = 2$ and $b = 6$. 35

2.13 VOQ occupancy as observed by an incoming packet for $pps = 3$ and $b = 6$. 35

2.14 VOQ occupancy as observed by an incoming packet for $pps = 4$ and $b = 6$. 37

2.15 VOQ occupancy as observed by an incoming packet for $pps = 5$ and $b = 6$. 37

2.16 VOQ occupancy as observed by an incoming packet for $pps = 8$ and $b = 6$. 37

2.17 VOQ occupancy as observed by an incoming packet for $pps = 10$ and $b = 6$. 37

3.1 tag of active PEs based on ps_{row} and ps_{col}. 51

3.2 Internal bus connections depending on tag. 51

4.1 The function f_s for well-nested, width-1 CST configuration algorithm. 74

4.2 The function f_c for well-nested, width-1 CST configuration algorithm. 75

4.3 The function f_s for width-1 multicast CST configuration algorithm. 79

4.4 The function f_c for width-1 multicast CST configuration algorithm. 80

5.1 PE configurations for creating buses to the right. 94

5.2 Position of sources and destinations on the R-Mesh. 95

List of Figures

1.1 Basic structure of a switch. 2

1.2 Basic architecture of an $n \times n$ crossbar-based input-queued packet switch. 4

1.3 Crosspoint configurations. 4

1.4 Communications on a CST. 6

1.5 A fat-tree with multiple edges between two nodes denoting higher bandwidth. . . . 8

2.1 Basic structure of a switch. 10

2.2 Structure of an $n \times n$ input-queued packet switch with a crossbar-based data fabric. 11

2.3 Scheduling on a 3×3 crossbar. 14

2.4 Slots, rounds, and frames. 15

2.5 Slots and frames in Neely *et al.* [59]. 19

2.6 Delay for various switch sizes for different *pps*. 25

2.7 VOQ occupancy as observed by an incoming packet. 27

2.8 Percentage packet loss for different *pps*. 30

2.9 Percentage packet loss for different VOQ sizes. 30

2.10 Delay for different frame sizes. 31

2.11 The on-off traffic model. 31

2.12 Average delay for bursty traffic for different *pps*. 32

2.13 VOQ occupancy as observed by an incoming packet for mean burst size 3. 33

2.14 VOQ occupancy as observed by an incoming packet for mean burst size 6. 36

2.15 Percentage packet loss for bursty traffic with for different *pps*. 38

2.16 Percentage packet loss for different VOQ sizes for traffic with mean burst size 3. . 39

2.17 Percentage packet loss for different VOQ sizes for traffic with mean burst size 6. . 39

2.18 Average delay for different frame sizes for bursty traffic with mean burst size 3. . . 40

2.19 Average delay for different frame sizes for bursty traffic with mean burst size 6. . . 40

3.1 Example of equivalence between crossbar scheduling and bipartite matching. . . . 42

3.2 Maximal size bipartite matching by Kelsen. 45

3.3 An example of Kelsen's algorithm. 46

3.4 A 3×5 DR-Mesh. 47

3.5 Prefix sums computation on an R-Mesh. 49

3.6 Neighbor localization on an R-Mesh. 49

3.7 Procedure *halve*. 53

4.1 The Self-Reconfigurable Gate Array. 59

4.2 Communications on a CST. 60

4.3 Some arbitrary configurations of a CST switch. 61

4.4 A multicast set. 62

4.5 A well-nested communication set. 63

4.6 The internal structure of a CST switch. 64

4.7 Some switch configurations. 66

4.8 Pseudocode for Phase 3 of the algorithm. 68

4.9 An example of the general CST configuration algorithm. 69

4.10 Two different multicast sets with same source-destination pattern. 78

4.11 Computation of IDs for a well-nested communication set. 82

4.12 Part of a CST showing a maximum source incompatible. 86

5.1 An 8-leaf fat-tree; multiple edges between two switches denote higher bandwidth. . 88

5.2 A level $k+1$ switch. 89

5.3 The R-Mesh inside the fat-tree switch. 92

5.4 Examples of allowed and conflicting configurations. 94

5.5 Example illustrating Stage (ii). 95

5.6 Example illustrating Stage (iii). 96

5.7 Final bus configurations. 96

5.8 Bus configurations of example 2. 96

Abstract

Interconnection networks are important parts of modern computing systems, facilitating communication between a system's components. Switches connecting various nodes of an interconnection network serve to move data in the network. The switch's delay and throughput impact the overall performance of the network and thus the system. Scheduling efficient movement of data through a switch and configuring the switch to realize a schedule are the main themes of this research. We consider various interconnection network switches including (*i*) crossbar-based switches, (*ii*) circuit-switched tree switches, and (*iii*) fat-tree switches.

For crossbar-based input-queued switches, a recent result established that logarithmic packet delay is possible. However, this result assumes that packet transmission time through the switch is no less than schedule-generation time. We prove that without this assumption (as is the case in practice) packet delay becomes linear. We also report results of simulations that bear out our result for practical switch sizes and indicate that a fast scheduling algorithm reduces not only packet delay but also buffer size. We also propose a fast mesh-of-trees based distributed switch scheduling (maximal-matching based) algorithm that has polylog complexity.

A circuit-switched tree (CST) can serve as an interconnect structure for various computing architectures and models such as the self-reconfigurable gate array and the reconfigurable mesh. A CST is a tree structure with source and destination processing elements as leaves and switches as internal nodes. We design several scheduling and configuration algorithms that distributedly partition a given set of communications into non-conflicting subsets and then establish switch settings and paths on the CST corresponding to the communications.

A fat-tree is another widely used interconnection structure in many of today's high-performance clusters. We embed a reconfigurable mesh inside a fat-tree switch to generate efficient connections. We present an R-Mesh-based algorithm for a fat-tree switch that creates buses connecting input and output ports corresponding to various communications using that switch.

Chapter 1
Introduction

Most of today's computing systems, ranging from large parallel systems and high-performance clusters to a single chip System-on-Chip (SoC), that need communication among multiple constituents, use interconnection networks. Interconnection networks are one of the main factors in determining the performance of such architectures, affecting the message latency, bandwidth, routing complexity, switching structure, system scalability, fault-tolerance and overall cost. Additionally, advances in interconnect technology often lag the developments in other aspects of computing, especially processing speed. Hence, the performance of the interconnection network is often the critical limiting factor in many of today's computing systems [16].

The topology is one of the main design choices for any interconnection network that dictates various other considerations like routing and flow control. Researchers have proposed various topologies [16, 20, 46]. There are two main classes of interconnection network topologies – direct and indirect. A direct network is one where all the nodes act as sources and destinations of data and participate in the routing of the data as well. An indirect network, on the other hand, is one in which the source/destination nodes and routing nodes are distinct.

Indirect networks are often switch-based networks, where communication between source and destination nodes is realized through one or more *switch* or *router* nodes. Data paths exist between source and destination nodes through switches. The switches are usually connected to other switches as well. The arrangement of switches connecting the sources and destinations determines the topology of the entire network.

Most of today's fastest supercomputers use indirect networks. For example, the Earth Simulator, the fastest supercomputer between the years 2002 and 2004 (according to the top500.org website [37]), uses a crossbar-based switch; Cray's XT5m [38] employs 2-d torus-based switches. These

switches only forward or route data rather than produce or consume data. Many of the current Networks-on-Chip (NoCs) [9, 10, 54] also use indirect networks.

This dissertation deals primarily with network switches. Figure 1.1 depicts the basic structure of a network switch. A switch has several input and output ports through which data arrives and leaves the switch. Each switch has a data unit, also referred to as switch fabric or data plane, which is responsible for the physical transmission of data from the input ports to the output ports. There is no one universal structure for this data plane and manufacturers of switches use various architectures ranging from crossbars and multi-dimensional tori to multistage interconnection networks. Additionally, each switch has a control unit, also referred to as the control plane, arbiter or switch processor, that controls the way the data unit handles data transmission. The data and control planes of a switch are separate entities and their architecture and implementation need not be related. The control plane can operate at a separate rate compared to the data plane and can have a centralized or a distributed architecture.

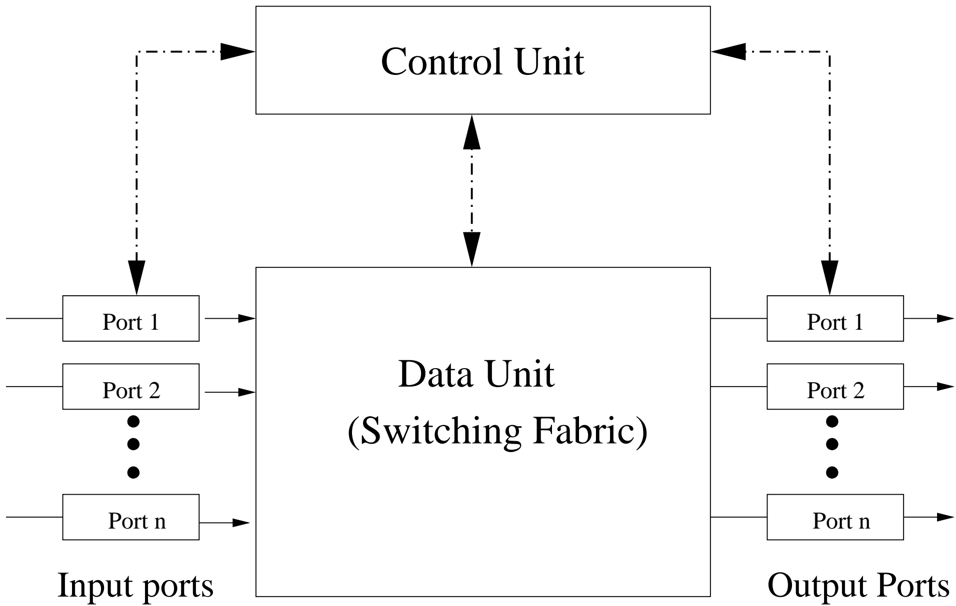


FIGURE 1.1. Basic structure of a switch.

Given a set of input-output connection requests, it is the job of the control unit to construct an efficient *schedule* of transmissions between input and output ports. Moreover, the control unit must also establish paths between inputs and outputs to perform these transmissions. This is done

by *configuring* the switch to connect the appropriate input and output ports. Typically, not all requested paths can be established simultaneously (due to topological and architectural constraints of the switch). The scheduling algorithm should be cognizant of these constraints in constructing a schedule and configure the switch accordingly. These two functions of the control unit, namely scheduling and configuration, are critical and determine the overall performance of the switch including delay, throughput, and complexity.

This dissertation examines scheduling and configuration of three classes of switches that find applicability in networks ranging from the Internet to Networks on (a) chip (NoCs):

1. crossbar-based input-queued switches,
2. circuit-switched trees, and
3. switches of fat-tree based interconnection networks.

The next few sections introduce our work on network switches. In each of these sections, we first present some background needed to frame our contribution, followed by an overview of our contributions.

1.1 Crossbar-Based Input-Queued Switch

A crossbar-based switch is one whose data fabric is a crossbar; an $n \times n$ crossbar switch (see Figure 1.2) has n input ports (conventionally to the left) and n output ports (at the bottom). An input-queued switch is one in which input ports have buffers that temporarily queue arriving packets before scheduling them for transmitting to the appropriate output port. The buffers at each input port is organized as n queues, one for each output port. For any $1 \leq i,\ j \leq n$, queue$(i,\ j)$ holds packets arriving at input i and destined to output j. This form of maintaining a separate queue for each output is called virtual output queuing (VOQ). Switches with VOQs are scalable, capable of very high throughput (100% in theory) and avoid the head of line blocking problem [56] that reduces the throughput to 58.6%.

This dissertation studies packet scheduling, packet delays, and buffer requirements for these switches. It is clear that fast algorithms generally increase the switch throughput and reduce packet

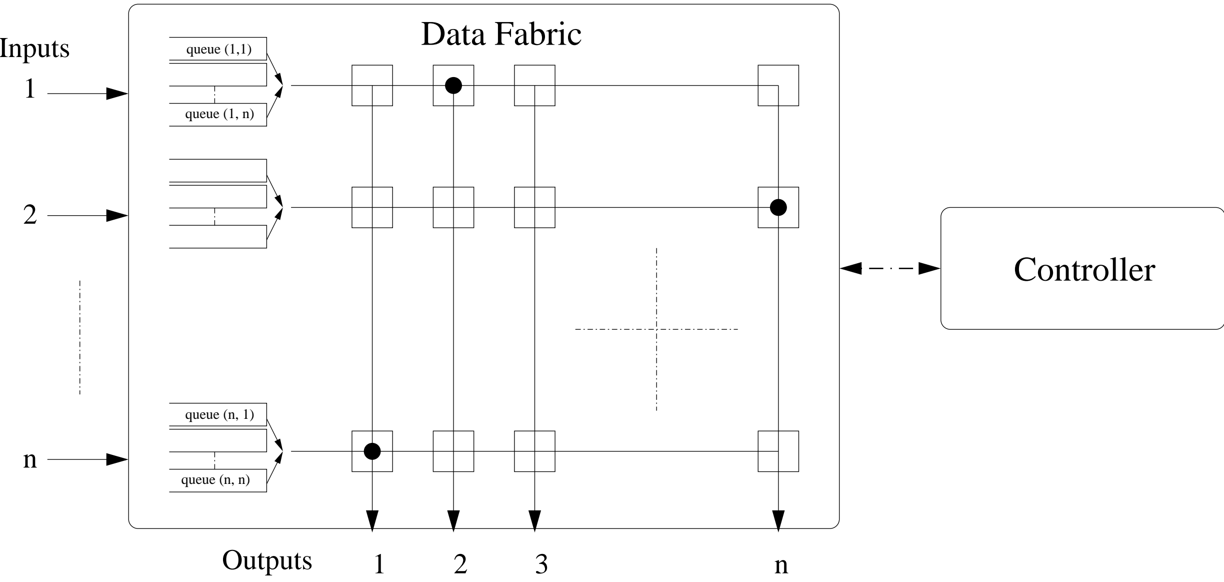


FIGURE 1.2. Basic architecture of an $n \times n$ crossbar-based input-queued packet switch.

delay (at least they do not degrade it). However, the price paid for speed is usually in hardware cost, complexity, and power consumption. In our work we show that there is a disproportionately high benefit to using a very fast scheduling algorithm for input-queued crossbar switches. While it drastically reduces delay, it pays (if at all) a much smaller price in hardware cost and power. We make our contributions more specific later in this section.

A crossbar usually transmits a packet from input port i to output port j by sending it right along row i to column j, and then turning down at column j to reach output j. For this, the crosspoint (internal switch of the crossbar) at row i and column j configures as in Figure 1.3(a), and all other crosspoints configure as in Figure 1.3(b). During transmission of this packet from i to j, no

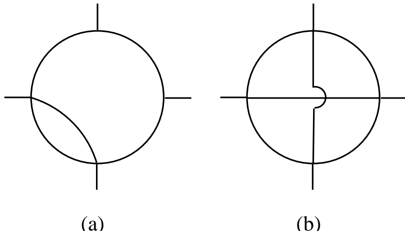


FIGURE 1.3. Crosspoint configurations.

other packet can be transmitted from i or to j. That is, the crossbar has at most one set crosspoint (configuration of Figure 1.3(a)) in a row and at most one set crosspoint in a column. This restriction,

called the *crossbar constraint* and affects the way packets are scheduled on the crossbar. Figure 1.2 shows a partial configuration of a crossbar that satisfies the crossbar constraint.

At any point in time, the crossbar has a set of packets to deliver at its input ports. It selects a subset of waiting packets (or a schedule) to transmit. For example, in Figure 1.2 the crossbar selects to send packets from inputs 1, 2 and n to outputs 2, n and 1, respectively, and configures the crossbar accordingly. Other packets (such as from input port 2 to output port 1, for example) must wait for another schedule. The time between a packet's arrival at an input port and the point when it leaves the input queue (for its destination output port) is called its *delay*.

Packet transmission time is the amount of time needed to transmit all bits in a packet. Call the time unit of one packet transmission as a *slot*. Until recently, packet scheduling algorithms for an $n \times n$ crossbar with uniform traffic attained an average packet delay of $\Omega(n)$ slots. Neely, Modiano, and Cheng [59] developed the Fair Frame scheduling algorithm that reduces delay to $O(\log n)$ slots. The result of Neely *et al.* assumes that schedule generation time is no more than the packet transmission time. Packet transmission time, however, depends on the medium bandwidth and the packet length and is independent of the schedule time. In practice packet transmission time is much smaller than schedule times for large crossbars. We investigate the effect on the delay of the Fair Frame algorithm of decoupling schedule generation time from packet transmission time.

Contribution of Our Work: Let t_{slot} (or a slot time) denote the packet transmission time, and let t_{round} (or a round time) denote schedule generation time. Define $pps = \frac{t_{round}}{t_{slot}}$ as the packets per schedule or the number of packets that a switch can transmit from an input port in the time needed to construct a schedule. With $pps = 1$, Neely *et al.* [59] proved that the average packet delay on an $n \times n$ crossbar is $O(\log n)$ slots.

In Chapter 2, we analytically show that when $pps > 1$ (as is usually the case in practice), then packet delay jumps to $\Omega(n)$. It is easy to show that packet delay is $O(\log n)$ [59]. Therefore our result establishes that for $pps > 1$ the packet delay jumps directly from $O(\log n)$ to $\Theta(n)$. Our result is also counter-intuitive as is establishes that there is no graceful degradation of packet delay as pps increases beyond 1.

Next, we examine the applicability of our result to practical crossbar sizes through extensive simulations. We show that, for $pps > 2$, packet delay degrades significantly. For $pps = 2$, the delay is reasonable for network sizes used in practice. We also show in our simulations that for higher *pps* the number of input buffers needed for a given level of performance (say, packet drop rate) is significantly higher. This points to the possibility that any savings in computational hardware and power consumption afforded by a large value of *pps* may be lost to a larger demand for buffer space. These results are presented in Chapter 2.

Having established the importance of a fast scheduling algorithm, we construct one in Chapter 3. This algorithm runs in polylog time ($O(\log^4 n \log\log n)$ time for an $n \times n$ crossbar) on a mesh-of-trees type structure that closely resembles the crossbar topology. In devising this algorithm we also construct a polylog time maximal matching algorithm for a reconfigurable mesh [77]; this may be of independent interest.

1.2 Circuit-Switched Tree

The circuit-switched tree (CST) is an interconnect structured as a binary tree with processing elements (PEs) at leaves and switches at internal nodes (see Figure 1.4). It facilitates communication between PEs at the leaves.

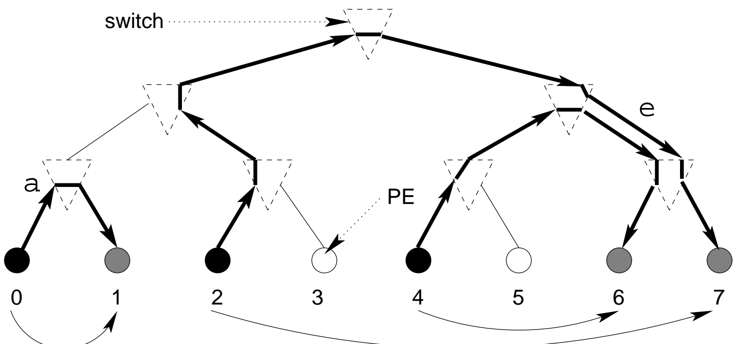


FIGURE 1.4. Communications on a CST; a dark (resp., shaded) leaf represents a source (resp., destination).

Each switch-switch or PE-switch connection is a pair of oppositely directed edges. Each directed edge of the CST can carry only one communication at a time. This necessitates an efficient scheduling for communication among PEs. For example, in Figure 1.4 communication between PE

pairs $(2, 7)$ and $(4, 6)$ cannot be simultaneous. To implement a schedule, the switches of a CST must configure to establish paths between communicating PEs.

A key descriptor of a set of communications on a CST is its width [21]. A set of communications with width w requires at least w rounds to complete. However, a width-w communication set may require more than w rounds.

Contribution of Our Work: We devise a distributed algorithm that schedules any width-w set of (point-to-point) communications. The algorithm terminates in r rounds, where $w \leq r < 2w$, and configures the CST for each round of scheduled communications. We adapt this algorithm for two special cases that are provably optimal. The first is a width-1, oriented communication set and the second a width-w, oriented, well-nested set; these are described in more detail in Chapter 4. The significant feature of the algorithm for these cases is that each PE starts off with just a local descriptor of whether it is a source or a destination. The source and destination PEs do not know each other's identities or position on the CST. That is, the algorithm builds on local information to construct an optimal global schedule and configuration for each round of that schedule.

We also present a multicasting algorithm for oriented, well-nested communications. In addition to local information (about whether a PE is a source or a destination), here a PE that is at the "end" of a multicast chain needs to be flagged as well. We prove that without this additional information, no two-pass algorithm (such as ours) can solve this problem.

1.3 Fat-Tree Switch

A fat-tree [46, 47] is a variation of the simple tree where the bandwidth between the switches increases exponentially with increasing levels of the tree, as shown in Figure 1.5. Fat trees are a popular choice for connecting processors and devices in many of today's high performance computing environments [37]. In a general tree, there is a unique path between a source-destination pair. However, in a fat-tree, there are multiple links between any two switches. Hence, while the set of switches traversed from a given source to a given destination is unique, with multiple links between these switches, the control unit needs to efficiently select a link connecting each pair of switches.

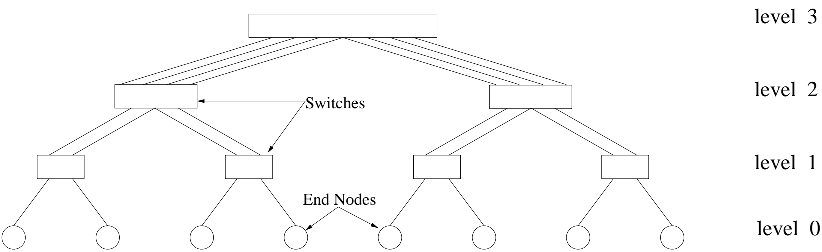


FIGURE 1.5. A fat-tree with multiple edges between two nodes denoting higher bandwidth.

A significant amount of research has been carried out in the efficient design, implementation, routing, as well as performance analysis of fat-tree-based interconnection networks [4, 19, 34, 51, 61, 72, 82]. In most of these results, especially those for the InfiniBand architecture, some form of table lookup determines the intra-switch routing. A sequential computation of these routes does not scale well. To our knowledge no research has so far focused on a distributed approach to routing within the fat-tree switch. We use an R-Mesh [77], a versatile reconfigurable computational model, to implement the switch.

Contribution of Our Work: This part of the research is preliminary. We have identified an approach to the problem and have developed a basic strategy for further development. Specifically, for a given $\ell \times 2^{k+1}$ R-Mesh (representing a fat-tree switch with 2^{k+1} links to the parent and 2^k links to each child), we have devised an R-Mesh algorithm that efficiently matches requests from input ports to available output ports in the desired direction of the fat-tree. The algorithm generates a schedule (configuration of the R-Mesh) and configures the data plane to accommodate this schedule.

Chapter 2

Crossbar-Based Switches: Conditions for Logarithmic Delay

2.1 Introduction and Background

Interconnection networks used in many current systems are indirect (see Chapter 1). In an indirect network, switches act as intermediate routing nodes that forward packets from the source towards the destination. Depending on the type of switching, packets traversing the network could be temporarily stored at the switches as well. Figure 2.1 shows the basic structure of a network switch. The data unit or data plane represents the physical fabric for packet transfer. The control unit orchestrates the flow of packets in the data plane. In a typical setting, packets arrive at input ports with each packet destined to an output port. The control plane factors in topological and architectural constraints of the switch to construct a schedule for packet transfer. Then the data plane configures to deliver packets according to the schedule. In this chapter (and the next) we consider packet scheduling.

Various data fabric structures exist. We consider a crossbar-based data fabric, one of the most common structures used in interconnection network switches. Figure 2.2 depicts the structure of a crossbar. An $n \times n$ crossbar has n rows, n columns, and n^2 crosspoints. The usual convention is to connect each row to an input port and each column to an output port. The crossbar sets crosspoints (depicted as boxes with darkened circles in Figure 2.2; see also Figure 1.3(a) on Page 4) to connect a row to a column. By configuring other crosspoints in that row and column to as in Figure 1.3(b) on Page 4, the crossbar establishes a path from an input port to an output port through which data is transmitted. For example in Figure 2.2, the set crosspoint at the intersection of row 1 and column 2 connects input port 1 to output port 2. The crossbar is a non-blocking network. That is, an $n \times n$ crossbar can connect each input to a distinct output according to any of the $n!$ possible permutations. A given set of communications may not be a permutation, however. For example, an input may have packets destined for several outputs or an output may be the destination for

packets from several inputs. However, an input can transmit data to only one output at a time and an output can receive data from only one input at any time. (We are considering point-to-point communications through the crossbar here. For multicasts, a single input can send the same data to multiple outputs simultaneously.) This means that communications at any given point must be a restriction of a permutation. Put differently, at any point in time, the crossbar configuration can have at most one of the n crosspoints connected on each row or each column. We call this restriction the *crossbar constraint*. In constructing a schedule, the control plane must account for the crossbar constraint.

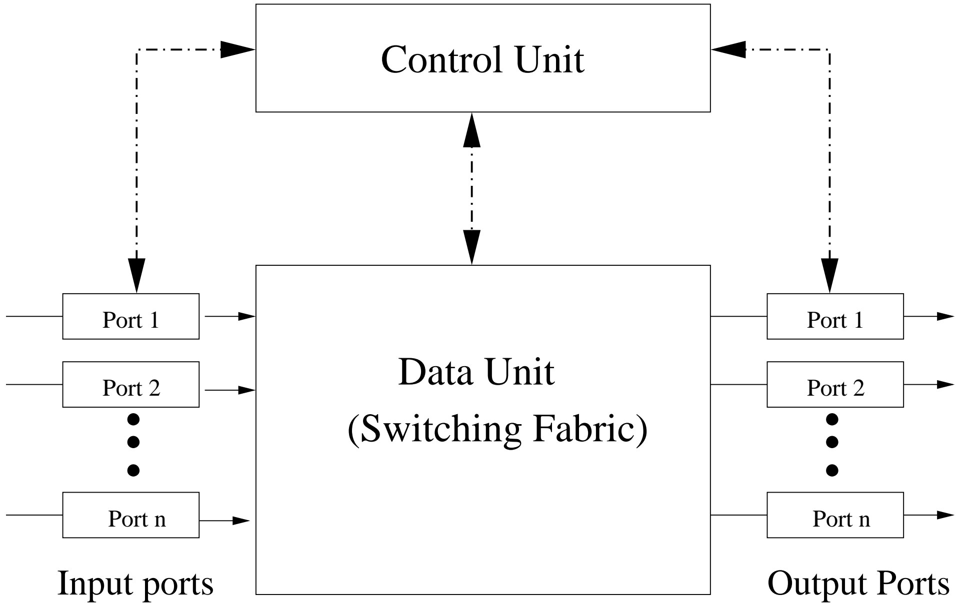


FIGURE 2.1. Basic structure of a switch.

There are many practical examples of routers and switches that employ a crossbar. These include the Intel Teraflop Router – Cavallino [11], Cray's T3E [68, 69], the Reliable Router [17], SGI's SPIDER [30], the Chaos Router [5, 6], the Arctic Router [64], HP's R2 router, the Ariadne router, IBM's SP2 switch, and the Inmos C104 switch [20]. The Earth Simulator, the fastest supercomputer from 2002 to 2004, uses a 640×640 single-stage crossbar. The interconnect families of Infiniband, Myrinet, and Quadrics together account for 31% of the current top 500 supercomputers [37]. Infiniband, Myrinet, and Quadrics implement crossbars of sizes 24×24, 32×32, and 8×8, respectively [36]. Examples of other routers and switches employing a crossbar include

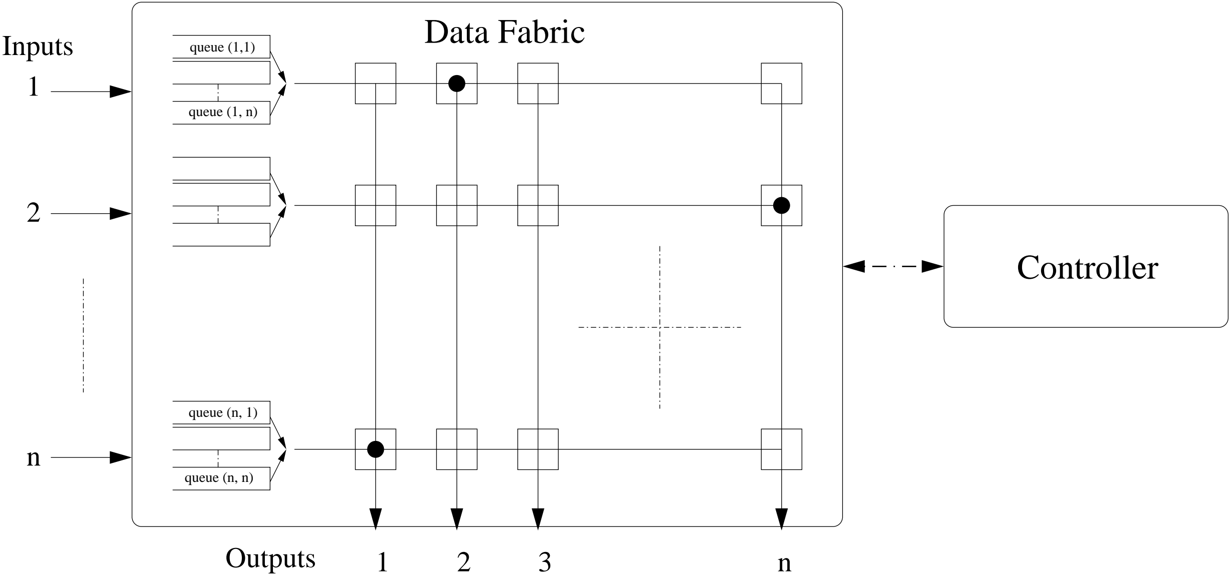


FIGURE 2.2. Structure of an $n \times n$ input-queued packet switch with a crossbar-based data fabric.

IBM's OSMOSIS project's optical switch that employs a 64×64 crossbar [83] and Huawei's Quidway S6500 series high-end multi-service switch [40] and Quidway S8500 series 10G core routing switch [41].

The crossbar switches we consider have buffers at input ports that temporarily store incoming packets before they are transmitted to the appropriate output port. On an $n \times n$ input-queued crossbar these input buffers are organized as n separate queues (queue$(i,\ j)$ for $1 \leq i,\ j \leq n$). Here queue$(i,\ j)$ holds packets arriving at input i and destined to output j. We call this method of organizing input buffers as Virtual Output Queuing (VOQ). VOQ has the benefit of avoiding the head of line (HOL) blocking problem [56] that can significantly impact the switch throughput. In practice, there are different ways of implementing VOQs. They range from having separate physical queues to having a single queue at an input with additional mechanisms to emulate separate VOQs on it. Figure 2.2 depicts an input-queued crossbar switch with VOQs.

Thus, in an input-queued crossbar switch with VOQ, packets arrive at input ports and are queued according to the destination ports. The distribution of the packets at the VOQs is represented as an $n \times n$ traffic matrix whose entry in row i and column j gives the number of packets in queue(i, j). For instance, Figure 2.3(b) shows the traffic matrix for the example in that Figure 2.3(a). Periodically, the control plane takes a snapshot of the traffic matrix and generates a schedule of parallel

communications on the crossbar. Packets at input queues wait for their turn in a schedule. The amount of time between the arrival of a packet at an input port and its exiting the input queue for transmission to the output port is the *delay* of the packet.

Crossbar switches with queues at only output ports are also possible. However, such a switch instantly transmits any arriving data packets through the crossbar to the output port. For the worst case, this requires the data fabric of an $n \times n$ crossbar to operate at a rate n times faster than the input rate[1]. This condition makes the implementation of output-queued crossbar switches impractical from a scalability point of view [13]. Combined input-output queued (CIOQ) crossbar switches employing both input and output queuing exist as well [13]. A relatively newer idea is to have queues at each crosspoint along with input queues [14]. We call these combined input-crosspoint queued (CICQ) switches. The common name for CICQ switches in the literature is buffered-crossbar switches. The common name for switches without any crosspoint buffers is unbuffered switches (in spite of the fact that they do have buffers at input and/or output ports). Various CICQ switches exist ranging from ones that employ a single buffer at each crosspoint [65] to ones that employ a constant number of buffers per output port [57]. Buffered crossbars often require less complicated scheduling algorithms compared to unbuffered ones that provide guaranteed performance in terms of throughput, rate, and delay [14].

Our research considers an input-queued crossbar switch with VOQ (see Figure 2.2). Of all the crossbar-based packet switch architectures, the input-queued packet switch with VOQs is one of the most widely used. As we noted earlier, it does not suffer from the HOL blocking phenomenon. These switches also offer high scalability and throughput [49, 55, 63]. Often, these switches operate on fixed length units of data called *cells*. The switch breaks variable length packets into cells at the input and reassembles them at the output. In this dissertation, we ignore the issue of packet sizes for the most part, assuming all packets to be one cell long. The main problem that we address is that of scheduling packets from input queues to output ports of the switch [12, 49, 59]. The control

[1]The ratio of operation rate of the data fabric of a switch to the packet arrival rate at its input is called the *speedup*. A constant speedup (around 2) is considered acceptable.

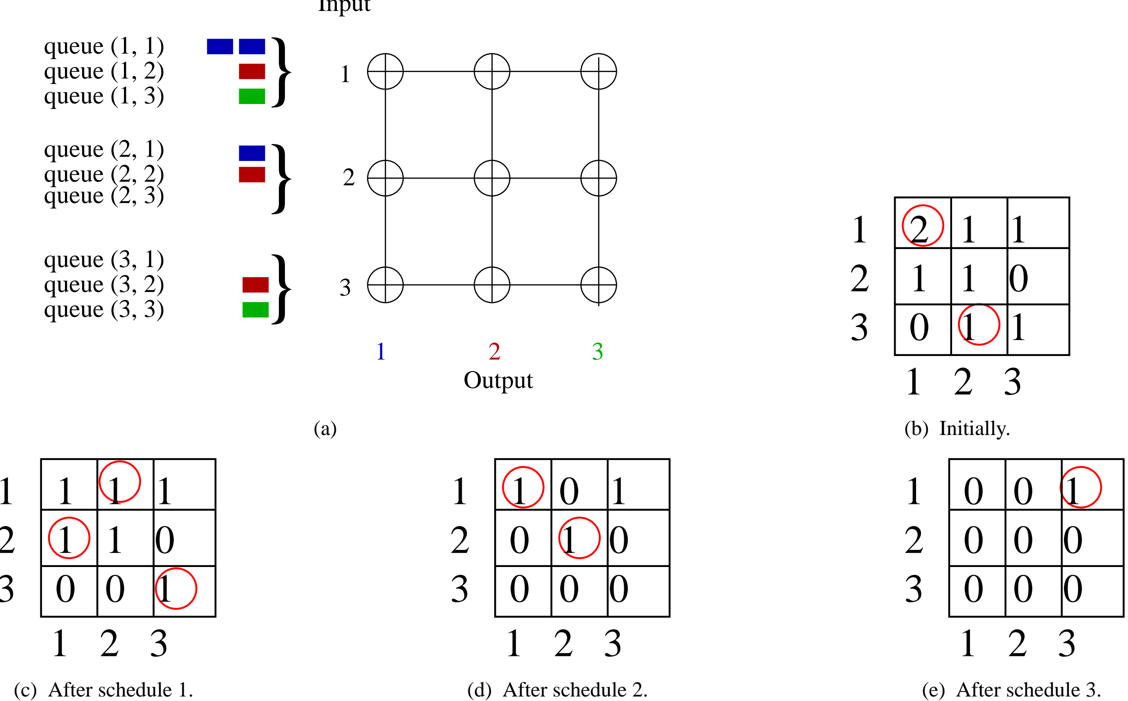
unit of an input-queued switch periodically generates schedules based on packets queued at the input ports, and the data plane then sets its crosspoints to reflect the schedule. Subsequently, for each connected input-output pair, the switch transmits packet(s) from the top of the corresponding VOQ. Many switches do this scheduling and packet transmission at fixed intervals of time.

It is customary to divide time in a switch into discrete *slots* where a slot is the time needed to transmit a packet [59]. However, most results also assume a slot to be long enough to generate a schedule as well. For the purpose of this work, let us call the condition that a slot is long enough to generate a schedule the "unit *pps* condition[2]". Practical systems often do not meet the unit *pps* condition. The time to transmit a packet (or a slot) is typically in the range of 50ns [12, 29, 83], but for large crossbars this is not sufficient time to construct a good schedule.

In this chapter we show that the unit *pps* condition is necessary for one recent significant result to hold. More specifically, Neely, Modiano and Cheng [59] recently showed that with the unit *pps* condition, packets can be scheduled on an $n \times n$ crossbar switch with $O(\log n)$ average delay. This result is significant as all previous results could only bound the delay to $O(n)$. The significance of our work is to prove that without the unit *pps* assumption, packet delay is $\Omega(n)$. That is, unit *pps* is necessary to achieve logarithmic packet delay. Our result also shows that there is no middle ground between logarithmic and linear packet delays. This underscores the importance of meeting the unit *pps* condition, perhaps by developing a fast schedule requiring no more than one slot.

An Example: Before we proceed to the formal description of the problem and its solution, we illustrate some of the ideas described so far through a small example. Consider a 3×3 crossbar with packets shown in Figure 2.3(a). The corresponding traffic matrix is in Figure 2.3(b). Suppose during the first slot the control plane schedules a packet from input port 1 to output port 1 and another from input port 3 to output port 2 (these are indicated by red circles in the traffic matrix of Figure 2.3(b)). After those packets are sent out, the new traffic matrix is as in Figure 2.3(c).

[2]The term *pps* stands for number of packets per schedule and is defined later in Section 2.2. Here "unit *pps*" simply means that the time to generate a schedule is no more than a slot (packet transmission time).



(a) (b) Initially. (c) After schedule 1. (d) After schedule 2. (e) After schedule 3.

FIGURE 2.3. Scheduling on a 3×3 crossbar.

It is customary to not consider new packet arrivals (if any) until the current matrix is exhausted. Figures 2.3(c), 2.3(d) and 2.3(e) show the subset of packets transmitted in the next three slots.

The entire schedule for this iteration spans four slots with 2, 3, 2, and 1 packets transmitted in these slots, respectively. For this span of four slots, the average packet delay for the eight packets is $(2 \times 1 + 3 \times 2 + 2 \times 3 + 1 \times 4)/8 = 18/8 = 2.25$; we have assumed that all eight packets arrived at the input just before the first slot in the example.

In the next two sections we formally define the quantities that characterize the problem addressed.

2.2 Slots, Rounds and Frames

As we noted earlier, time is discretized into slots on a switch. We now relate a slot to two other important time intervals, round and frame, for scheduling packets on a switch. Figure 2.4 shows these quantities.

Slot: A slot is the atomic unit in which time is divided for the crossbar. Denoted by t_{slot}, it equals the amount of time needed to transmit a packet from an input port to an output port. If the

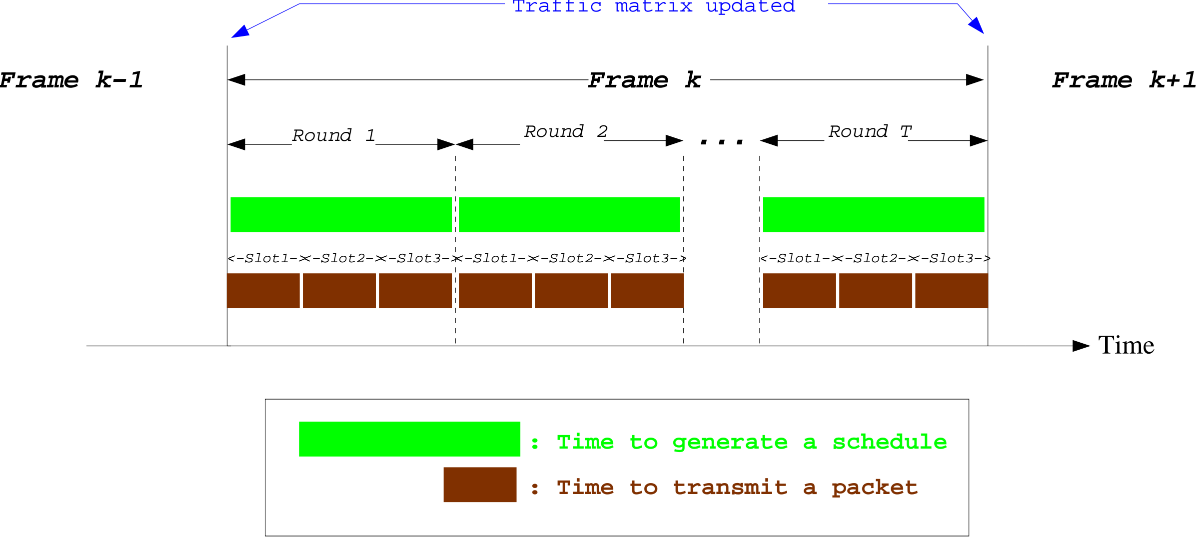


FIGURE 2.4. Slots, rounds, and frames.

data path from an input port to an output port can deliver b bits/second and a packet has a length of p bits, then $t_{slot} = \frac{p}{b}$.

<u>*Round:*</u> A round is the amount of time needed for the control plane to generate a schedule from a traffic matrix and for the data plane to configure for this schedule. This quantity, denoted by t_{round}, depends on the hardware available to (i) perform the computation to produce a schedule and (ii) to configure the data plane. This hardware complexity also depends on n, the size of the switch. Thus t_{round} is independent of t_{slot}, the slot time. In most practical systems, $t_{round} > t_{slot}$. Since scheduling and packet transmission are interleaved in the switch (see the "Fair Frame" algorithm below), there is no benefit to considering $t_{round} < t_{slot}$ as the scheduler cannot produce a new schedule until the current set of packets is sent. Thus, we assume that $t_{round} \geq t_{slot}$.

We define the quantity $\left\lfloor \frac{t_{round}}{t_{slots}} \right\rfloor$ as *pps* (packets per schedule), and it plays an important role in this chapter. For a given schedule and a switch configuration corresponding to it, it is possible to transmit as many as *pps* packets in a round. All these packets must use the same configuration, however.

For convenience and without loss of generality, we assume that t_{slot} divides t_{round}, so

$$pps = \frac{t_{round}}{t_{slot}}.$$

Frame: A frame is a sequence of rounds. Denote frame size (in rounds) by T. A frame is largely used as an analysis tool, though it also has some significance in the scheduling algorithm described below.

The Fair Frame Algorithm: As we noted earlier, our work in this chapter builds on results of Neely *et al.* [59]. They use an algorithm, called Fair Frame, that we suitably modify for our work (while retaining the same name).

Divide time into slots and let a sequence of *pps* slots constitute a round. A sequence of T rounds forms a frame (see Figure 2.4). At the start of Frame k, the algorithm takes a snapshot M_0^k of the traffic matrix. This matrix is the target of packet transmission during Frame k. Let the rounds of Frame k be $r_1^k,\ r_2^k, \cdots,\ r_T^k$. In Round r_1^k the scheduler takes the initial traffic matrix M_0^k and generates a subset of packets that can be scheduled simultaneously while respecting the crossbar constraint[3]. It also configures the crosspoints accordingly.

In Round r_2^k (the second round), the schedule and configuration generated in r_1^k (the first round) is used to send packets on the data plane. As many as *pps* packets are sent between each input-output pair connected in the schedule. In fact, if input i is connected to output j in the schedule and if queue$(i,\ j)$ has x packets in it, then $\min\{x,\ pps\}$ packets are sent from i to j in Round r_2^k. At the end of Round r_1^k, the control plane adjusts the initial traffic matrix M_0^k to account for the packets to be sent in Round r_2^k. Denote this new matrix by M_1^k.

While the data plane is transmitting packets during Round r_2^k, the control plane makes a new schedule based on M_1^k. In general, for Round r_z^k (for $1 \leq z \leq T$), the control plane generates a schedule on the basis of the traffic matrix M_{z-1}^k, while the data plane transmits packets according to the schedule and the configuration generated during Round r_{z-1}^k; here r_0^k is the last round of Frame $k-1$.

[3]The Fair Frame algorithm computes a maximum matching between the input output pairs with packets waiting to be transmitted between them. More details of this matching appear in Chapter 3.

The last schedule of the current frame (Frame k) and the adjusted traffic matrix are computed at the end of Round r_T^k, the last round. Since the algorithm aims to send all packet arriving in Frame $k-1$ by the end of Frame k, the traffic matrix M_T^k should ideally be empty. If M_T^k is not empty, the remaining packets (called *non-conforming packets*) are treated as if they arrived in Frame k.

At the start of Frame $k+1$, the control plane generates the initial traffic matrix M_0^{k+1} which includes all arrivals during Frame k and all non-conforming packets remaining in M_T^k. The scheduler then proceeds as in Frame k.

2.3 Stability and Delay

In any switch, the average overall packet arrival rate cannot exceed the average packet output rate (average throughput). A switch satisfying this condition is said to be *stable*. An unstable switch will require unbounded buffer space and packets will incur unbounded delay. In a stable switch the input and output rates (averaged over a large enough time) are equal. In the context of the Fair Frame algorithm, the length of the frame (T rounds) is this "large enough time". After T rounds of Frame k, the algorithm should have sent out (nearly) all packets that arrived before the start of Frame k. Note that other researchers in this area have used the term frame to define a set of slots but with different criteria as far as what groups those slots together. For example, Lou *et al.* [52], X. Li *et al.* [49], and Y. Li *et al.* [50] defined a frame as a set of slots for which a single schedule is generated. Rojas-Cessa *et al.* [66] defined a frame as a set of cells (fixed-size units into which variable-size packets are divided) that can be transmitted together.

Beyond this point we will take the length of a frame to mean not just the interval for updating the traffic matrix for new packet arrivals, but also as a large enough time to send out nearly all packets that arrived before the end of the previous frame. That is, a frame has, with high probability, no non-conforming packets.

If a frame size T has no non-conforming packets, then no packet waits longer than $2T$ rounds. Thus the frame size can be a useful tool to determine the packet delay.

The Result of Neely, Modiano and Cheng [59]: Neely *et al.* [59] used a special case of the Fair Frame algorithm of Section 2.2 to establish the following result.

Theorem 2.3.1. (Neely *et al.* [59]): *For uniform traffic, Poisson arrival*[4]*, and* $pps = 1$*, an* $n \times n$ *input-queued crossbar can schedule packets with* $O(\log n)$ *delay.* □

The above result is a significant improvement over previous $O(n)$ delay bounds. The basic idea of the analysis by Neely *et al.* [59] is to show that for $T = \Theta(\log n)$, the probability of a packet becoming non-conforming is very low ($O(\frac{1}{n^2})$), then use this fact to show that the average delay is $O(T) = O(\log n)$. A significant assumption in this result is that $pps = 1$, that is, the schedule time does not exceed the slot time (as in Figure 2.5).

Our Contribution: The unit *pps* assumption in Neely *et al.* [59] is not always true in practice. For example, the crossbar-based optical switch in IBM's OSMOSIS project [83] required only 52 ns to transmit a packet. However, constructing a good schedule entails finding a matching on a $2n$-node bipartite graph (see Chapter 3) and can be quite time consuming for large n. Our contribution is in examining the $pps > 1$ case.

In the next section, we prove that the $O(\log n)$ packet delay of Neely *et al.* [59] holds only for the $pps = 1$ case. Moreover, for the $pps \geq 2$ case, the packet delay jumps to $\Omega(n)$, with no middle ground between the logarithmic and linear delays.

In Section 2.5 we present results from extensive simulations to show that our analytical result hold practical importance. We also use these simulations to show that larger *pps* values require larger buffer sizes. Thus, lowering *pps* has the benefit of not just lowering delay, but potentially offsetting some of the hardware costs for doing so. Other papers [43, 49, 50, 66] related to frame-based scheduling algorithms present simulation results on overall delay experienced by packets for various switch sizes and frame sizes (different frame sizes have different meanings in these papers based on the definition of frame as mentioned above). However, none of these papers analytically determines packet delay (except for trivial bounds).

[4]Traffic refers to the distribution of packet destinations. Under uniform traffic, each packet is independently destined to any of the n possible destinations with probability $1/n$. The arrival on the other hand, refers to the temporal distribution of arriving packets, without considering their destinations.

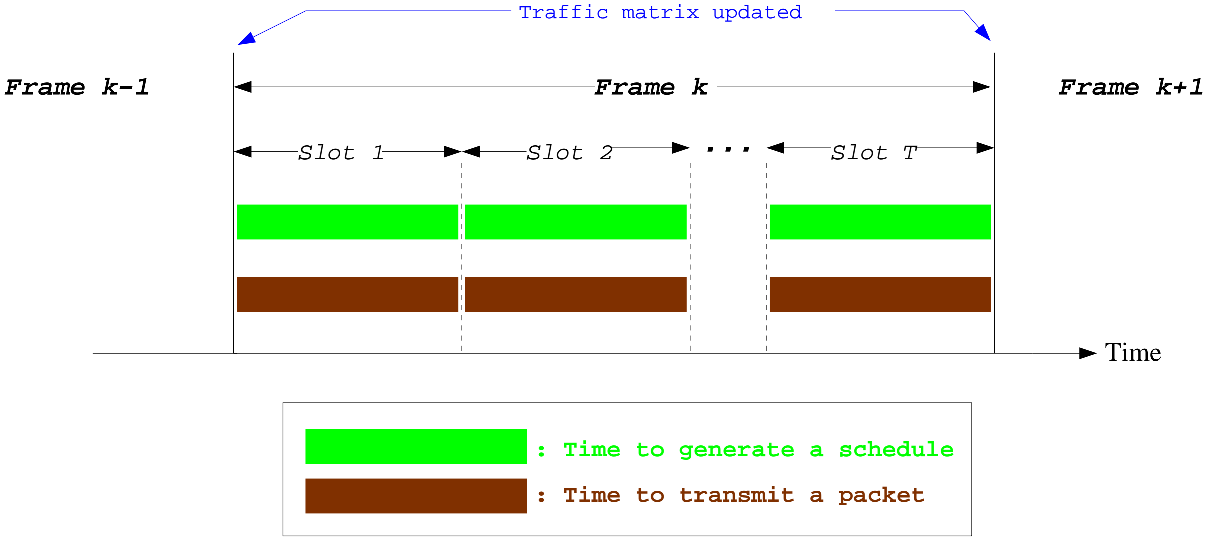


FIGURE 2.5. Slots and frames in Neely *et al.* [59].

The results described so far is for uniform traffic. We also present results (both analytical and simulation-based) for bursty traffic in Sections 2.4.2 and 2.5.2.

2.4 Necessary Conditions for Logarithmic Delay

As noted earlier, a frame-based scheduling algorithm divides time into frames, rounds, and slots (see Figure 2.4). A slot has duration t_{slot}, time to transmit a packet across the switch. A round contains $pps = \frac{t_{round}}{t_{slot}}$ slots, and it represents time to execute the scheduling algorithm. At the end of Frame $k-1$, the algorithm takes a snapshot of the input queues; this snapshot represents a set PKT of packets that have arrived during Frame $k-1$. During each round of Frame k, the scheduling algorithm computes a configuration of the crossbar switch based on a bipartite matching of input and output ports and transfers a subset of packets in PKT to their output ports. Each round uses a single switch configuration. Therefore, if input port i is connected to output port j during a round, then any packet leaving i during that round must be destined to j. After each round, the scheduling algorithm updates PKT to account for transmitted packets. The scheduler accounts for newly arrived packets only at the beginning of each frame. The goal of the scheduling algorithm is to route all packets in PKT by the end of Frame k. This results in a delay that is linearly upper bounded in the frame size. Selecting the number T of rounds per frame to achieve this goal is the focus of this section.

Recall that $pps = \frac{t_{round}}{t_{slot}}$. For $pps = 1$ and $T = O(\log n)$, this scheduling method fits the Fair Frame algorithm of Neely *et al.* [59]. The Fair Frame algorithm schedules all the packets arriving in one frame during the next frame with high probability. In the case where it does not route all outstanding packets in the next frame, it will insert them in unutilized rounds in subsequent frames. This will not significantly contribute to the average packet delay [59].

We examine the natural extension of the Fair Frame algorithm to $pps > 1$ as described in Section 2.2. That is, when the scheduling algorithm schedules input port i with output port j during a round and i has k packets destined to j, then the algorithm sends $\min\{k, pps\}$ packets from i to j during that round. Since at most pps packets can be sent in a round, we can also have at most pps arrivals in a round (nominally); this corresponds to one packet per slot per input port on an average. Note that this extension does not allow changing the switch configuration during a round. Therefore if port i has packets destined to d different destinations, then the switch needs at least d rounds to move all packets in i.

We now derive a relationship between pps, the number of slots per round, and T, the number of rounds per frame.

2.4.1 Uniform Random Traffic

In this section we establish that for uniform random traffic with no correlation between two successive packet arrivals, T can be $O(\log n)$ only when $pps = 1$, where n is the number of input or output ports in the switch. The traffic is uniform if each arriving packet has an equal probability of being destined to any of the n output ports (independently of any other packet).

Lemma 2.4.1. *For large n, the average number of distinct output ports among* $\mathfrak{q}$ *randomly destined packets is* $n(1 - e^{-\frac{\mathfrak{q}}{n}})$*, where n is the number of output ports in the switch.*

Proof. Consider the problem of randomly tossing $\mathfrak{q}$ balls into n bins. It is well known that the average number of empty bins is $n(1 - \frac{1}{n})^{\mathfrak{q}} \cong ne^{-\frac{\mathfrak{q}}{n}}$ for large n [58, 67]. In our case, empty bins correspond to ports with no packets destined to them. Hence, the average number of distinct ports to which packets are destined is $n - ne^{-\frac{\mathfrak{q}}{n}} \cong n(1 - e^{-\frac{\mathfrak{q}}{n}})$. □

In a frame-based scheduling algorithm with T rounds per frame and pps slots per round, the upper bound on the average number of packets arriving during a frame at each input port of the switch is $T \cdot pps$. With $\mathfrak{q} = T \cdot pps$, we have the following corollary.

Corollary 2.4.2. *For uniform random traffic and $1 \leq T < n$, $pps \leq \frac{n}{T} \ln\left(\frac{n}{n-T}\right)$ on an average.*

Proof. From Lemma 2.4.1, the average number of distinct output ports among the packets arriving at an input port during a frame is $n(1 - e^{-\frac{T \cdot pps}{n}})$. Since each round has only one switch configuration, a scheduling algorithm needs at least $n(1 - e^{-\frac{T \cdot pps}{n}})$ rounds to fully deplete the input queue. For a frame-based scheduling algorithm, this quantity must be no more than T, the number of rounds in a frame. That is, $n(1 - e^{-\frac{T \cdot pps}{n}}) \leq T$. Simplifying this inequality completes the proof. □

Lemma 2.4.3. *For $1 \leq T < n$, the function $f(T) = \frac{n}{T} \ln\left(\frac{n}{n-T}\right)$ is an increasing function of T.*

Proof. Let $y = \frac{T}{n-T}$. Then, we can express $f(T)$ as $F(y)$, where

$$F(y) = \left(1 + \frac{1}{y}\right) \ln(1+y).$$

Since $1 \leq T < n$ and y increases with T, we have $\frac{1}{n-1} \leq y \leq n-1$ for this range of values of y. Now,

$$\frac{dF}{dy} = \frac{y - \ln(y+1)}{y^2}$$

For $y \geq 1$, $\ln(y+1) < y$. Hence, for $y \geq 1$, the right hand side of the above equation evaluates to greater than zero. For $y < 1$, using a power series expansion we get the following.

$$\begin{aligned} y - \ln(y+1) &= \frac{y^2}{2} - \frac{y^3}{3} + \frac{y^4}{4} - \cdots \\ &= \sum_{i=1}^{n} y^{2i} \left(\frac{1}{2i} - \frac{y}{2i+1}\right) \\ &= \sum_{i=1}^{n} y^{2i} \left(\frac{2i+1-2iy}{2i(2i+1)}\right) \end{aligned}$$

Now, $2i(1-y)+1 > 0$, for $y < 1$. This implies that $\frac{y - ln(1+y)}{y^2} > 0$ for $y < 1$ as well. This shows that $F(y)$ increases with y and, hence, $f(T)$ increases with T. □

This brings us to our main result, Theorem 2.4.4, where we prove that logarithmic delay is possible only when $pps = 1$.

Theorem 2.4.4. *For uniform traffic and $pps > 1$, an $n \times n$ input-queued crossbar has an average packet delay of $\Omega(n)$.*

Proof. Let $T = \frac{n}{c}$ for some constant c. Then,

$$f\left(\frac{n}{c}\right) = \frac{n}{\frac{n}{c}} \ln\left(\frac{n}{n - \frac{n}{c}}\right) = c \ln\left(\frac{c}{c-1}\right).$$

For $c \geq 1.26$, that is, $T \leq \frac{n}{1.26}$, we have the following by Corollary 2.4.2

$$pps \leq c \ln\left(\frac{c}{c-1}\right) < 2.$$

By Lemma 2.4.3, $f(T)$ is an increasing function of T, so if $pps \geq 2$, then T must be at least $\frac{n}{1.26}$. □

Theorem 2.4.4 proves that the $O(\log n)$ delay derived by Neely *et al.* [59] is possible only if $pps = 1$ and does not hold even when $pps = 2$.

2.4.2 Bursty Traffic

In this section we extend our results for bursty traffic[5]. For bursty traffic we consider an on-off arrival process modulated by a two-state Markov chain. In such a process if p is the probability of an on-period ending at a time slot, then $b = 1/p$ is the mean burst size [12]. Since the upper bound on the average packet arrivals over a frame is $T \cdot pps$, then for bursty traffic the upper bound on the average number of bursts per frame is $T \cdot pps/b$.

Corollary 2.4.5. *For bursty traffic and $1 \leq T < n$, $\frac{pps}{b} \leq \frac{n}{T} \ln\left(\frac{n}{n-T}\right)$ on an average.*

Proof. Replacing $T \cdot pps$ by $T \cdot pps/b$ in the proof of Corollary 2.4.2 and simplifying, we get the above expression. □

Theorem 2.4.6. *For traffic with mean burst size $b \geq 1$ and $pps \geq 2b$, an $n \times n$ input-queued crossbar has an average packet delay of $\Omega(n)$.*

[5] We describe bursty traffic in more detail in Section 2.5.2.

Proof. Clearly, $\frac{pps}{b}$ is equivalent to pps in the proof of Theorem 2.4.4. The main difference is that, unlike pps, $\frac{pps}{b}$ may not be an integer. Consequently, if $pps \geq 2b$ (or $\frac{pps}{b} \geq 2$), then the delay is $\Omega(n)$. □

Note: When $pps \leq b$, $T = O(\log n)$ is possible by a simple extension of the result of Neely *et al.* [59]. Additionally, there is a small range of values of $b < pps < 2b$ for which $T = O(\log n)$ remains possible. Simulation results in the next section support this observation. Simulation results of Neely *et al.* for a modified version of Fair Frame on bursty traffic showed a pattern consistent with an increase of delay that is logarithmic in n, though they did not analytically prove $O(\log n)$ delay for bursty traffic.

As noted earlier, Theorems 2.4.4 and 2.4.6 point to the fact that logarithmic delay is possible only for $pps = 1$ for non-bursty traffic and $pps < 2b$ for bursty traffic. This points to the important and hitherto unrecognized insight that reduction in t_{round}, and consequently pps, causes a huge improvement from linear delay to logarithmic delay, and this is the main contribution of this part of the dissertation.

2.5 Simulation Results

In the last section, we analytically established that while $O(\log n)$ average packet delay is achievable for $pps = 1$ (result of Neely *et al.* [59]), $pps > 1$ implies $\Omega(n)$ delay. The constants in the Ω notation could determine the true tradeoff between scheduler speed and packet delay for practical networks. In this section we study this through simulations. A delayed packet is stored in a buffer. The longer the average delay, the larger the buffer occupancy. Hence, we also examine the impact of VOQ sizes on packet delay and the relationship between input-buffer size and average packet delay. On the whole, our simulations show the expected separation between $pps = 1$ and large values of pps. For uniform traffic, $pps = 2$ seems to be reasonably close to $pps = 1$ for practical values of n. Similar results are obtained for bursty traffic.

We built a software framework using the OMNET++ simulation environment [42] to simulate the Fair Frame algorithm for various values of *pps* for both uniform random as well as bursty traffic. Our simulations considered switch sizes ranging from 16×16 to 100×100.

Each simulation ran for 15000 slots at each input by which time the system has reached a steady state for a large amount of time. The results from the simulations match our analytical results from the previous section and display the effects of constant factors hidden in the asymptotic notation there. For all the graphs (unless otherwise stated), we varied the packet arrivals for different switch sizes and different *pps* in such a way that the load at each input port is close to 0.9 (i.e., on average 0.9 packets arrive per slot)[6]. We also varied the size of each virtual output queue; we considered two scenarios, infinite and finite VOQs. For finite VOQs, we varied buffer size in the range $[1, \lceil \ln n \rceil]$. To support our analytical proofs we studied the delay as a function of *pps*, input load, and frame size. We also studied VOQ occupancy (for infinite VOQs) as observed by an incoming packet for different values of *pps*. Further, we looked at the percentage of packets lost for finite VOQs.

In our simulations we always kept t_{slot} fixed and expressed the overall delay in terms of slots. In order to achieve different values of *pps*, we varied t_{round}. We also defined the duration of a frame to be $t_{round} \lceil \ln n \rceil$. Hence, with varying t_{round}, the duration of the frame also varied. This implies that for $pps > 1$, even though the delay is $\Theta(n)$, the frame size was kept at $O(\log n)$. This allows more frequent updates to the traffic matrix in the $pps > 1$ cases than that allowed by Fair Frame. Thus our simulation results could potentially show more improvements if the right frame size was used. Further note that for $pps > 1$, we always transmitted *pps* packets per round (if there are packets to transmit).

The switch sizes for each simulation reported in this section were in the range 16×16 to 100×100. However, we ran some simulations for 500×500 and 1000×1000 switches to verify that the

[6]Stability conditions of a switch dictates that on average, at most a single packet can arrive per slot at each input port. This translates to a maximum load of 1 at each input port. However, to successfully schedule packets arriving with a load of 1, the scheduling algorithm must generate schedules that connect each input to some output at each slot. This is practically infeasible. Hence, for practical switch simulations, an input load close to but less than 1 is considered.

general trends shown in the results of switch sizes up to 100×100 translate to the bigger switch sizes.

2.5.1 Uniform Random Traffic

In this section we present results for uniform random traffic. Packets arrive at each input port following a Poisson distribution, and the arrival of each packet is completely independent of any other packet's arrival. Destinations of arriving packets are uniformly distributed among all output ports. Except when explicitly studying VOQ size, we used infinite queues for these simulations.

Delay: Our main result proved that logarithmic delay is possible only for $pps = 1$, and for $pps > 1$, the delay becomes linear. Neely *et al.*'s result of logarithmic delay is a significant improvement over any existing algorithm in terms of delay. So our result, which defines the limits of Neely *et al.*'s result, is also important. The delay graphs (Figure 2.6) support our results.

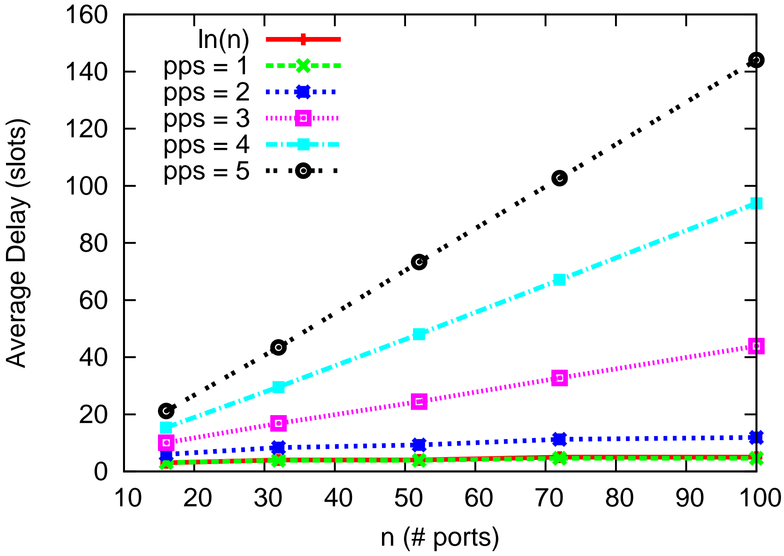


FIGURE 2.6. Delay for various switch sizes for different *pps*.

Figure 2.6 shows the average delay of packets for switches of various sizes for $1 \leq pps \leq 5$. We found when $pps = 1$, delay is nearly $\ln n$, as shown in the graph. Further, when $pps \geq 2$, the delay is visibly higher. For the case of $pps = 2$, experimental results show that the delay is slightly greater than $\ln n$, and the delay appears clearly linear for $pps \geq 3$. This corroborates our result that for uniform random traffic, $O(\log n)$ delay is only possible when $pps = 1$. While the $pps = 2$ case

is also linear (from our analytical results), the associated constants are sufficiently small to make the results seem close to $\ln n$ for practical values of n.

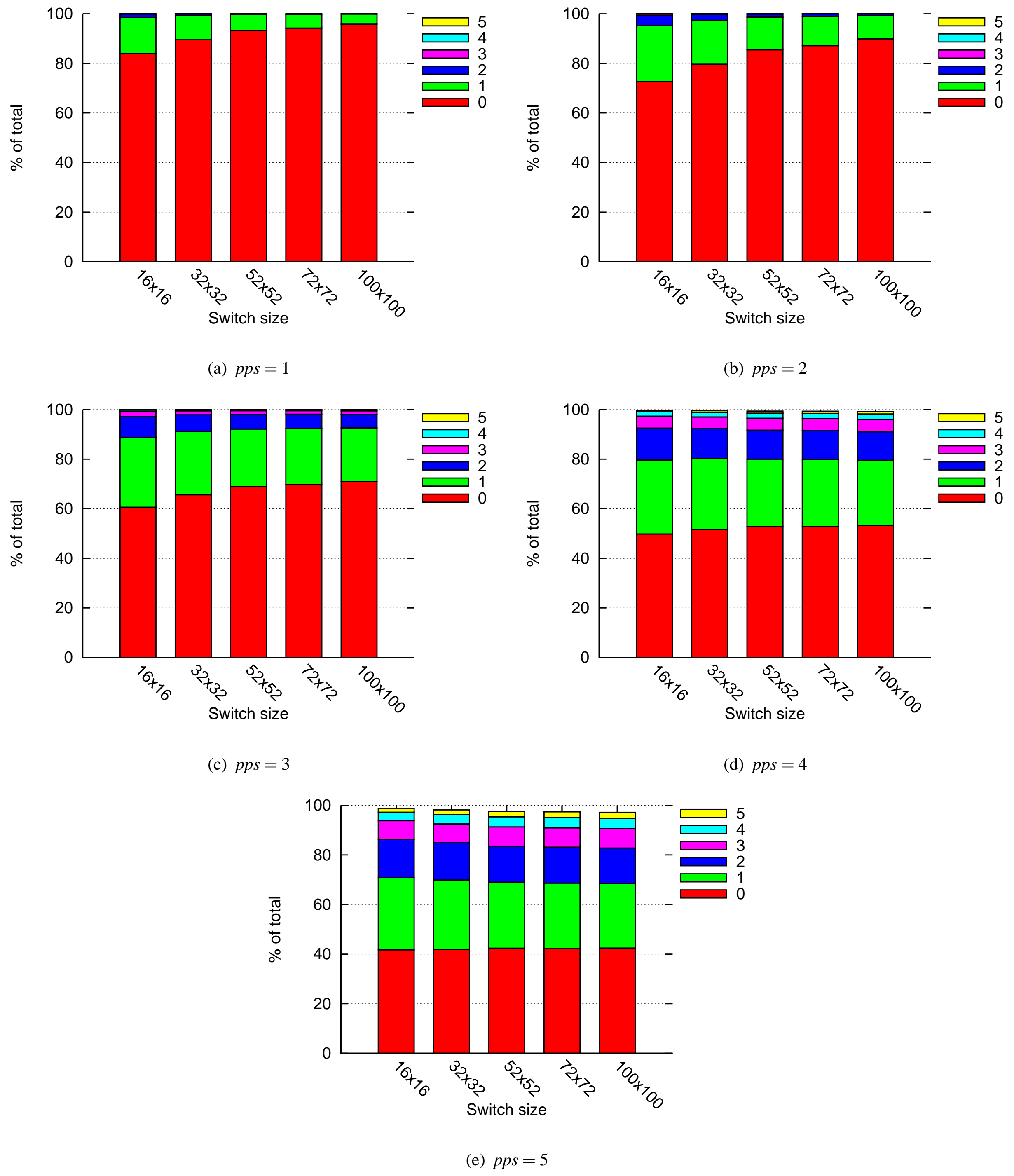
VOQ Occupancy: In an input-queued switch, the queues at the input ports temporarily store the incoming packets till they are transmitted through the switch. For the same input load, a switch with less delay will have to store the packets for a shorter amount of time. Hence, the delay affects the size of VOQs needed as well. Our results on VOQ occupancy (Figure 2.7) show that when $pps = 1$ the VOQ occupancy (buffer requirement) is less than $\ln n$ almost 100% of the time. This percentage decreases slightly with increasing *pps*.

We look at the distribution of the number of waiting packets in the corresponding VOQ, as observed by an incoming packet, for various *pps* (Figure 2.7). Figure 2.7 plots a stacked histogram where the percentage of packets encountering an empty queue on arrival, the percentage of packets that encounter a queue with one existing packet, and so on are plotted along a single bar in a cumulative way (Tables 2.1 - 2.5 shows the actual numbers). As shown in Figure 2.7(a), when $pps = 1$ almost 85% of the incoming packets for a 16×16 switch and almost 95% of the incoming packets for a 100×100 switch encounter an empty VOQ on arrival. (This suggests that with no input buffer, the drop rate for a 100×100 switch is only about 5%.) As *pps* increases, a higher percentage of incoming packets encounters a VOQ with one or more packets already waiting in it.

TABLE 2.1. VOQ occupancy as observed by an incoming packet for $pps = 1$.

	Percentage of arriving packets that encountered					
	0	1	2	3	4	5
Switch size	packet(s) in the queue					
16×16	84.01	14.53	1.36	0.10	0.01	0.00
32×32	89.51	9.86	0.60	0.03	0.00	0.00
52×52	93.36	6.40	0.24	0.01	0.00	0.00
72×72	94.26	5.56	0.18	0.00	0.00	0.00
100×100	95.80	4.10	0.10	0.00	0.00	0.00

This result points to the important fact that *pps* affects the VOQ size as well. For the Fair Frame algorithm, a logarithmic VOQ size will be sufficient (i.e., the probability of a packet loss due to



(a) $pps = 1$

(b) $pps = 2$

(c) $pps = 3$

(d) $pps = 4$

(e) $pps = 5$

FIGURE 2.7. VOQ occupancy as observed by an incoming packet.

VOQ overflow is pretty much zero) only when $pps = 1$. But as pps increases, more packets will be dropped.

Packet Loss for Different Switch Sizes and ***pps*****:** For large switch sizes, the amount of memory needed to store the packets at the input port is an important factor. Previously, our results showed

TABLE 2.2. VOQ occupancy as observed by an incoming packet for $pps = 2$.

	Percentage of arriving packets that encountered					
	0	1	2	3	4	5
Switch size	packet(s) in the queue					
16×16	72.60	22.58	4.17	0.56	0.09	0.01
32×32	79.68	17.73	2.33	0.24	0.02	0.00
52×52	85.48	13.17	1.25	0.09	0.01	0.00
72×72	87.14	11.83	0.96	0.06	0.00	0.00
100×100	89.89	9.46	0.61	0.03	0.00	0.00

TABLE 2.3. VOQ occupancy as observed by an incoming packet for $pps = 3$.

	Percentage of arriving packets that encountered					
	0	1	2	3	4	5
Switch size	packet(s) in the queue					
16×16	60.62	28.07	8.51	2.17	0.50	0.11
32×32	65.63	25.45	6.83	1.60	0.37	0.09
52×52	69.02	23.13	5.93	1.44	0.36	0.09
72×72	69.70	22.68	5.75	1.41	0.34	0.08
100×100	71.02	21.60	5.49	1.39	0.36	0.09

that different values of *pps* translate to different queue-length requirements to guarantee no packet loss. In these results we limit the size of VOQs and observe the effect of the limited sizes to the corresponding percentage packet losses. Many applications, especially ones without real-time constraints, can tolerate a certain amount of packet loss without any major problem. So, if a switch designer knows what the application requirements are in terms of allowable packet loss, then the designer could select an optimum VOQ size. The graphs in this section (packet loss for various switch sizes with fixed VOQ length, $\lceil \ln(n) \rceil$) and the next section (packet loss for two fixed switch sizes with variable VOQ sizes) attempt to shed some light on this VOQ-size – packet loss trade-off.

Figure 2.8 depicts the percentage of lost packets for different switch sizes for different *pps*. In Figure 2.8, we limit the size of each VOQ to $\lceil \ln(n) \rceil$, and like before, used an input load of 0.9. As *pps* increases beyond 3, the percentage of dropped packets starts to increase quite drastically.

Packet Loss for Different VOQ Sizes: We also selected two switch sizes, 50×50 and 100×100, varied the VOQ sizes for each switch in the range $[1, \lceil \ln(n) \rceil]$, and observed the packet loss for each VOQ size. Figure 2.9 presents the result. This result shows that if *pps* increases then for a

TABLE 2.4. VOQ occupancy as observed by an incoming packet for $pps = 4$.

	Percentage of arriving packets that encountered					
	0	1	2	3	4	5
Switch size	packet(s) in the queue					
16×16	49.82	29.81	12.90	4.84	1.75	0.59
32×32	51.70	28.54	12.08	4.71	1.82	0.70
52×52	52.82	27.21	11.65	4.83	2.04	0.85
72×72	52.85	26.99	11.64	4.92	2.07	0.88
100×100	53.28	26.28	11.48	5.00	2.20	0.98

TABLE 2.5. VOQ occupancy as observed by an incoming packet for $pps = 5$.

	Percentage of arriving packets that encountered					
	0	1	2	3	4	5
Switch size	packet(s) in the queue					
16×16	41.73	29.01	15.62	7.48	3.44	1.55
32×32	41.97	27.93	15.04	7.60	3.81	1.85
52×52	42.37	26.66	14.55	7.72	4.08	2.16
72×72	42.18	26.46	14.50	7.80	4.18	2.25
100×100	42.44	25.99	14.34	7.81	4.26	2.36

fixed VOQ size the number of dropped packets also rises, especially for the lower VOQ sizes. As shown in Figure 2.9, $pps = 1$ translates to very low VOQ size requirements for limiting packet loss to a given level. This, however, is not true for higher values of *pps* and indirectly reinforces the fact that ensuring $pps = 1$ leads to big gains in delay (VOQ size is related to delay). Hence, we can use the graphs of Figure 2.9 to determine the VOQ size required to ensure that the packet loss does not exceed a certain value, given a fixed *pps*. Conversely, given a fixed VOQ size and a maximum allowable loss percentage, using Figure 2.9 we can suggest allowable values of *pps*.

If one views reducing round time as an investment in the computing/hardware cost of the switch, then the added cost of a larger number of buffers needed for large t_{round} could make it worthwhile to reduce t_{round}.

Different Frame Sizes: The Fair Frame algorithm schedules all conforming packets that arrive in any frame during the next frame. Hence, the maximum delay that a conforming packet can experience is twice the frame size. We ran simulations with different frame sizes and *pps* mainly

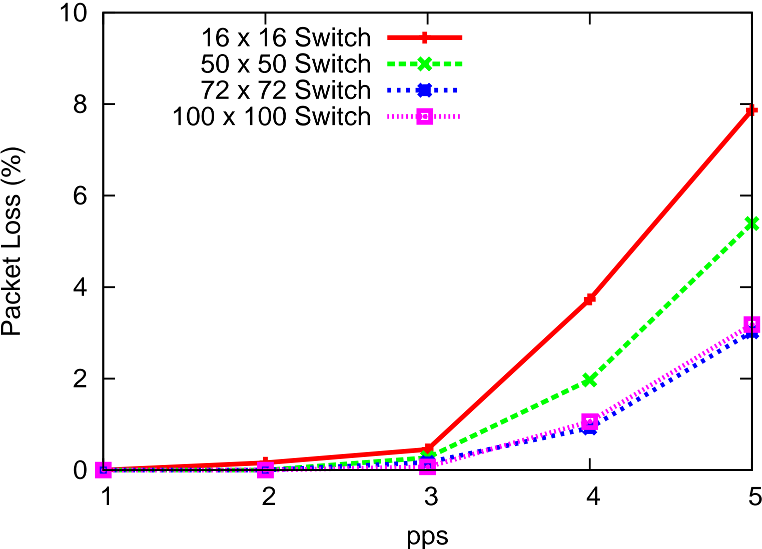


FIGURE 2.8. Percentage packet loss for different *pps*.

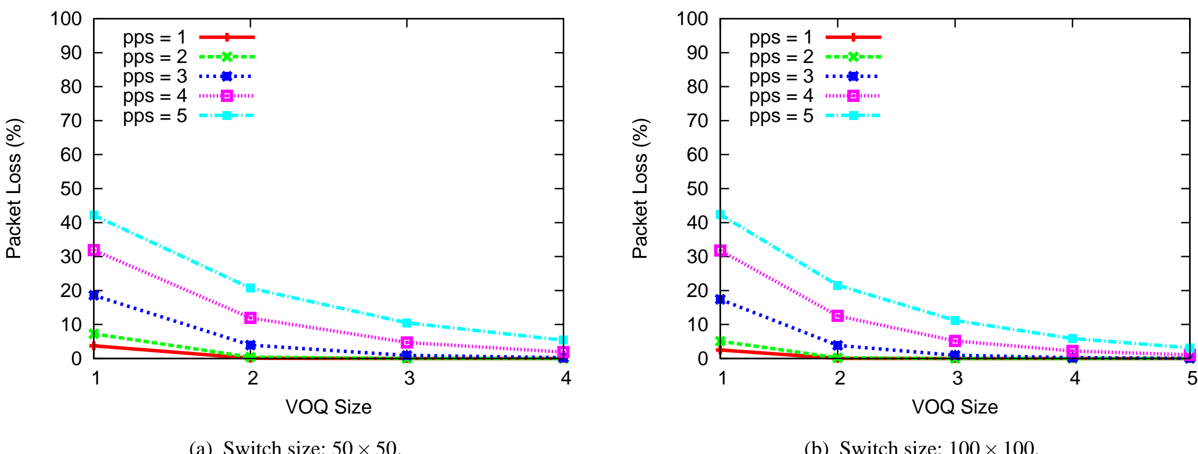
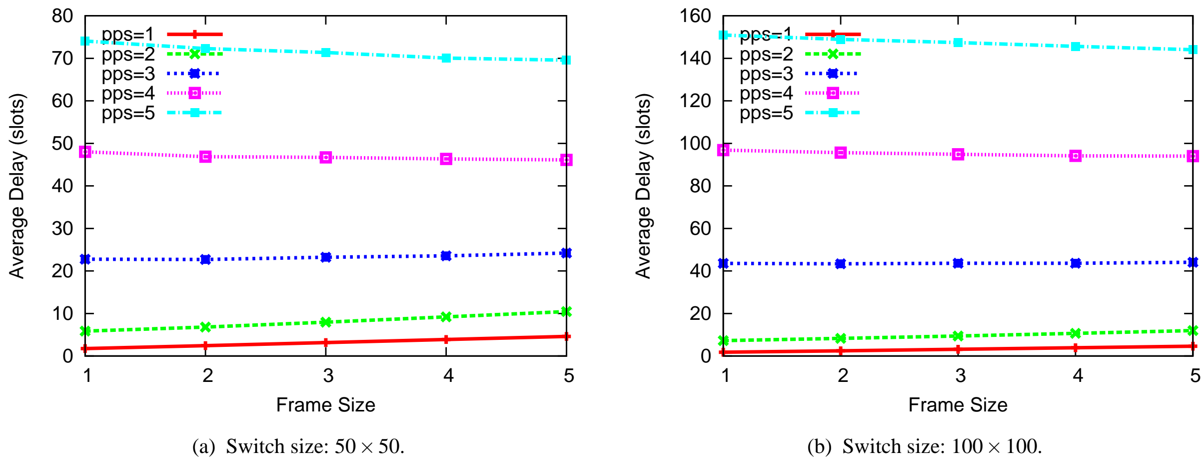


FIGURE 2.9. Percentage packet loss for different VOQ sizes.

to see whether this relationship between the frame size and delay holds for $pps \geq 2$. The results showed that for a fixed *pps*, different frame sizes do not make a big difference in the delay.

In our simulations we considered frame size $1 \leq T \leq 5$ and observed the average delay for a 50×50 and a 100×100 switch for $1 \leq pps \leq 5$. For each of these switches, we wanted to observe the effects of different frame sizes from 1 up to $\lceil \ln n \rceil$, hence the interval $[1,5]$. The results are in Figure 2.10.

Another interesting trend in Figure 2.10 is that for lower values of *pps*, the delay increases with increasing frame size, while with higher values of *pps*, delay decreases with increasing frame size. This happens because a higher value of *pps* means a scheduled input-output pair must receive



(a) Switch size: 50×50.

(b) Switch size: 100×100.

FIGURE 2.10. Delay for different frame sizes.

more packets in order to have something to transmit in each of the *pps* slots. Since higher frame size means a lower update frequency of queue information, the probability of having more packets in any particular VOQ is higher. Hence, with higher frame size, whenever an input-output pair is scheduled, the number of packets transmitted will be closer to *pps* relative to lower frame size, thus reducing the delay. On the other hand, with lower values of *pps*, arrival of *pps* packets at an input port destined to the same output port is relatively faster. Hence, with higher frame size, those packets have to wait longer in the VOQ before the queue information is updated, resulting in increased delay.

2.5.2 Bursty Traffic

Most Internet traffic is bursty in nature, so we ran our simulations for bursty traffic as well. For bursty traffic we use the on-off traffic model modulated by a two-state Markov chain [12]. Figure 2.11 shows the arrival process at each input port. During the on period, an input port con-

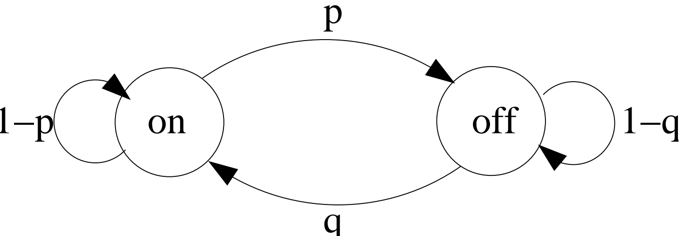
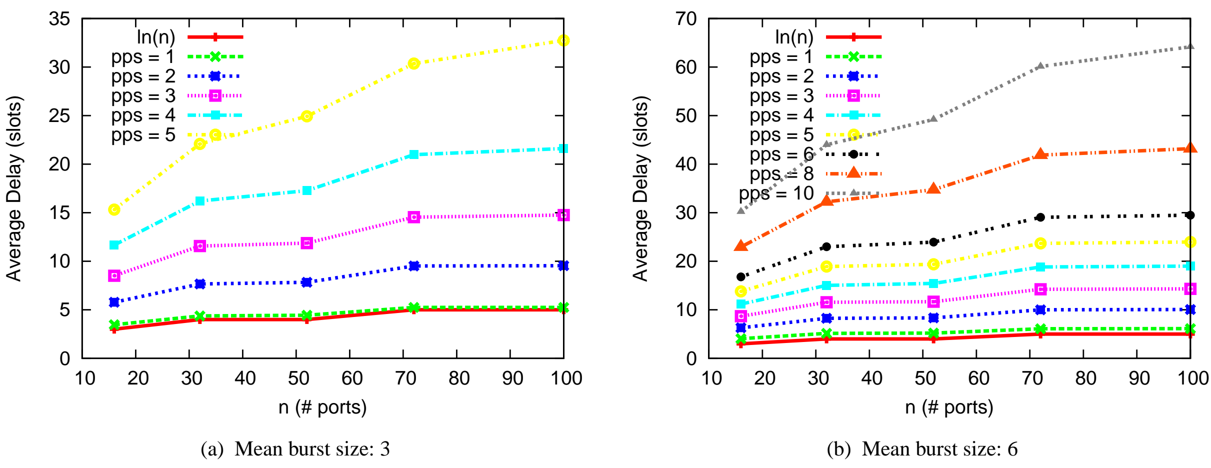


FIGURE 2.11. The on-off traffic model.

tinues to receive packets destined to the same output port. If p is the probability of starting an off

period and q is the probability of starting an on period, then the mean on period length b (burst size) is $1/p$, the mean off period length is $(1-q)/q$, and the offered load at each input port is $\frac{q}{q+p-pq}$. We use these expressions to model bursty traffic for two different mean burst sizes, 3 and 6. We ran all the simulations as in the uniform random traffic case for each of the burst sizes as well.

Delay: Figure 2.12 shows the average delay for bursty traffic for different values of *pps*. In Theorem 2.4.6 we showed that for $pps \geq 2b$, the delay is $\Omega(n)$. We also pointed out that for $pps \leq b$, delay can be logarithmic, but as *pps* starts to exceed b, the mean burst size, the delay quickly becomes $\Omega(n)$. In this study we primarily focus on $b \leq pps < 2b$ as the $pps \geq 2b$ case is very similar to the non-bursty case. In Figure 2.12(a), while $pps \leq b$ the average delay is indeed logarithmic. (Note that $\lceil \ln n \rceil$ ranges from 3 to 5 as n ranges from 16 to 100.)



(a) Mean burst size: 3

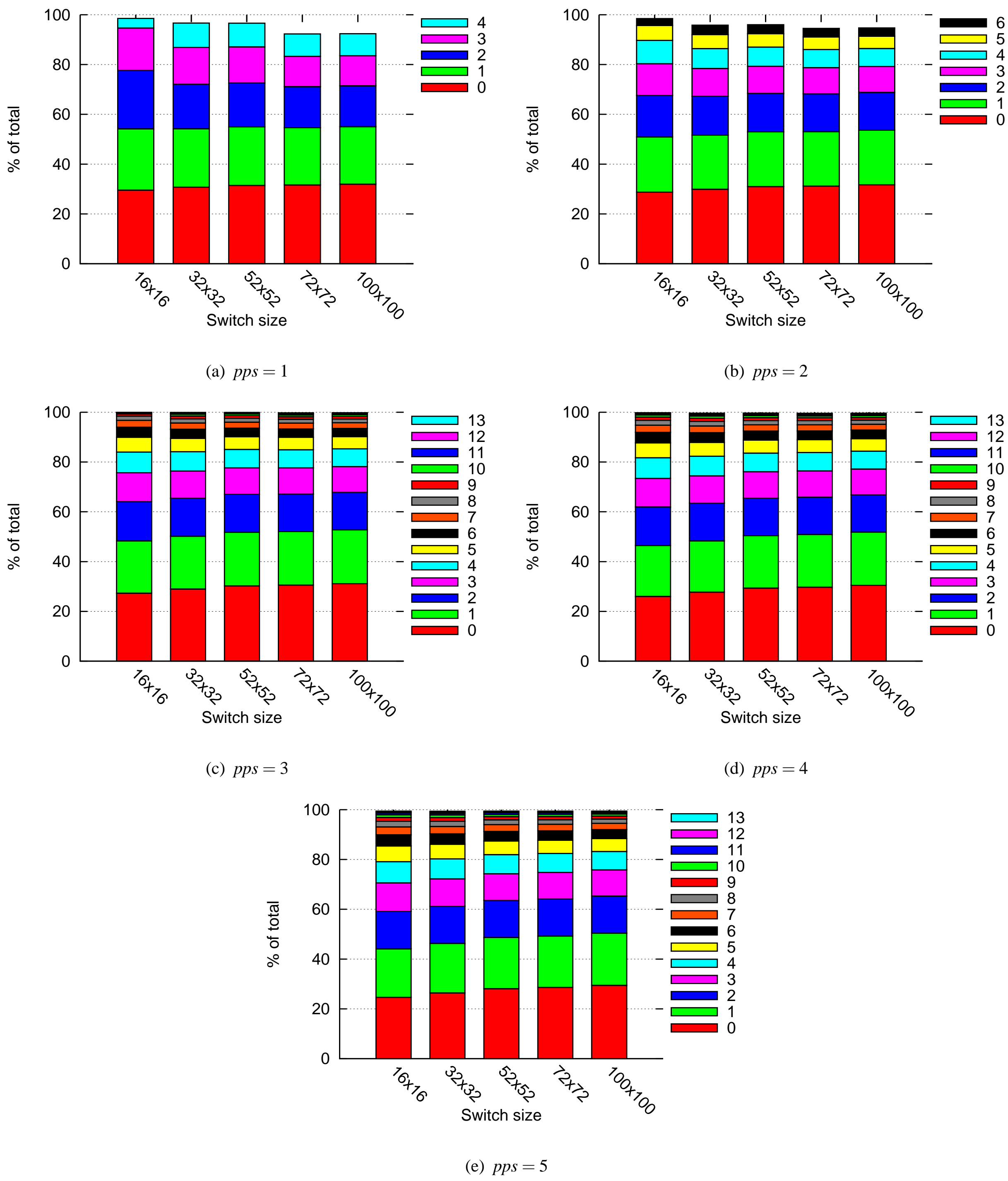
(b) Mean burst size: 6

FIGURE 2.12. Average delay for bursty traffic for different *pps*.

VOQ Occupancy: For bursty traffic we also analyzed the distribution of the number of waiting packets in the corresponding VOQ, as observed by an incoming packet, for various *pps* (Figures 2.13 and 2.14). For bursty traffic with $b = 3$, we expected that the number of arriving packets that will encounter between 0 and 2 waiting packets will dominate the percentage. In Figure 2.13 this is indeed the case. (Tables 2.6 - 2.17 shows the actual numbers).

As before, for $pps = 1$, a very high percentage of packets sees between 0 and b waiting packets at the VOQ. However, as *pps* exceeds b, a higher percentage of packets encounters more than b

waiting packets on arrival. This observation points to the relationship between VOQ lengths and value of *pps* relative to b for bursty traffic. For $b = 6$ similar trends are visible in Figure 2.14.



(a) $pps = 1$

(b) $pps = 2$

(c) $pps = 3$

(d) $pps = 4$

(e) $pps = 5$

FIGURE 2.13. VOQ occupancy as observed by an incoming packet for mean burst size 3.

TABLE 2.6. VOQ occupancy as observed by an incoming packet for $pps = 1$ and $b = 3$.

	Percentage of arriving packets that encountered					
	0	1	2	3	4	5
Switch size	packet(s) in the queue					
16×16	29.56	24.62	23.47	17.02	3.83	0.94
32×32	30.75	23.49	17.85	14.80	9.78	2.34
52×32	31.45	23.50	17.67	14.48	9.52	2.33
72×72	31.63	23.00	16.50	12.18	9.00	5.55
100×100	31.95	23.12	16.43	12.05	8.88	5.48

TABLE 2.7. VOQ occupancy as observed by an incoming packet for $pps = 2$ and $b = 3$.

	Percentage of arriving packets that encountered							
	0	1	2	3	4	5	6	7
Switch size	packet(s) in the queue							
16×16	28.75	22.18	16.64	12.77	9.37	5.96	2.79	0.92
32×32	29.94	21.75	15.57	11.21	7.96	5.60	3.82	2.35
52×32	31.00	21.98	15.45	10.88	7.69	5.38	3.65	2.26
72×72	31.17	21.87	15.21	10.50	7.29	5.06	3.43	2.29
100×100	31.66	22.00	15.15	10.42	7.20	4.94	3.35	2.23

TABLE 2.8. VOQ occupancy as observed by an incoming packet for $pps = 3$ and $b = 3$.

	Percentage of arriving packets that encountered										
	0	1	2	3	4	5	6	7	8	9	10
Switch size	packet(s) in the queue										
16×16	27.34	20.99	15.78	11.60	8.30	5.91	4.09	2.74	1.68	0.84	0.37
32×32	29.00	21.17	15.27	10.94	7.75	5.36	3.68	2.53	1.67	1.12	0.71
52×32	30.25	21.54	15.21	10.65	7.42	5.10	3.48	2.35	1.56	1.01	0.65
72×72	30.58	21.50	15.06	10.51	7.28	4.99	3.41	2.31	1.54	1.01	0.68
100×100	31.14	21.66	15.01	10.36	7.14	4.88	3.32	2.25	1.51	1.00	0.65

TABLE 2.9. VOQ occupancy as observed by an incoming packet for $pps = 4$ and $b = 3$.

	Percentage of arriving packets that encountered										
	0	1	2	3	4	5	6	7	8	9	10
Switch size	packet(s) in the queue										
16×16	26.02	20.43	15.51	11.47	8.30	5.97	4.22	2.89	1.95	1.31	0.82
32×32	27.76	20.59	15.11	10.97	7.89	5.58	3.90	2.71	1.86	1.25	0.87
52×32	29.35	21.08	15.00	10.66	7.50	5.22	3.64	2.51	1.71	1.16	0.77
72×72	29.75	21.13	14.98	10.56	7.39	5.14	3.55	2.45	1.68	1.14	0.76
100×100	30.49	21.35	14.93	10.41	7.21	5.00	3.44	2.35	1.60	1.08	0.72

TABLE 2.10. VOQ occupancy as observed by an incoming packet for $pps = 5$ and $b = 3$.

	Percentage of arriving packets that encountered										
	0	1	2	3	4	5	6	7	8	9	10
Switch size	packet(s) in the queue										
16×16	24.64	19.45	15.07	11.43	8.56	6.30	4.49	3.16	2.25	1.56	1.03
32×32	26.38	19.93	14.87	11.02	8.06	5.85	4.22	2.97	2.08	1.47	1.02
52×32	28.10	20.55	14.90	10.73	7.69	5.47	3.86	2.71	1.90	1.31	0.91
72×72	28.61	20.66	14.89	10.66	7.57	5.36	3.75	2.62	1.84	1.28	0.88
100×100	29.45	20.94	14.89	10.53	7.40	5.18	3.61	2.49	1.74	1.19	0.82

TABLE 2.11. VOQ occupancy as observed by an incoming packet for $pps = 1$ and $b = 6$.

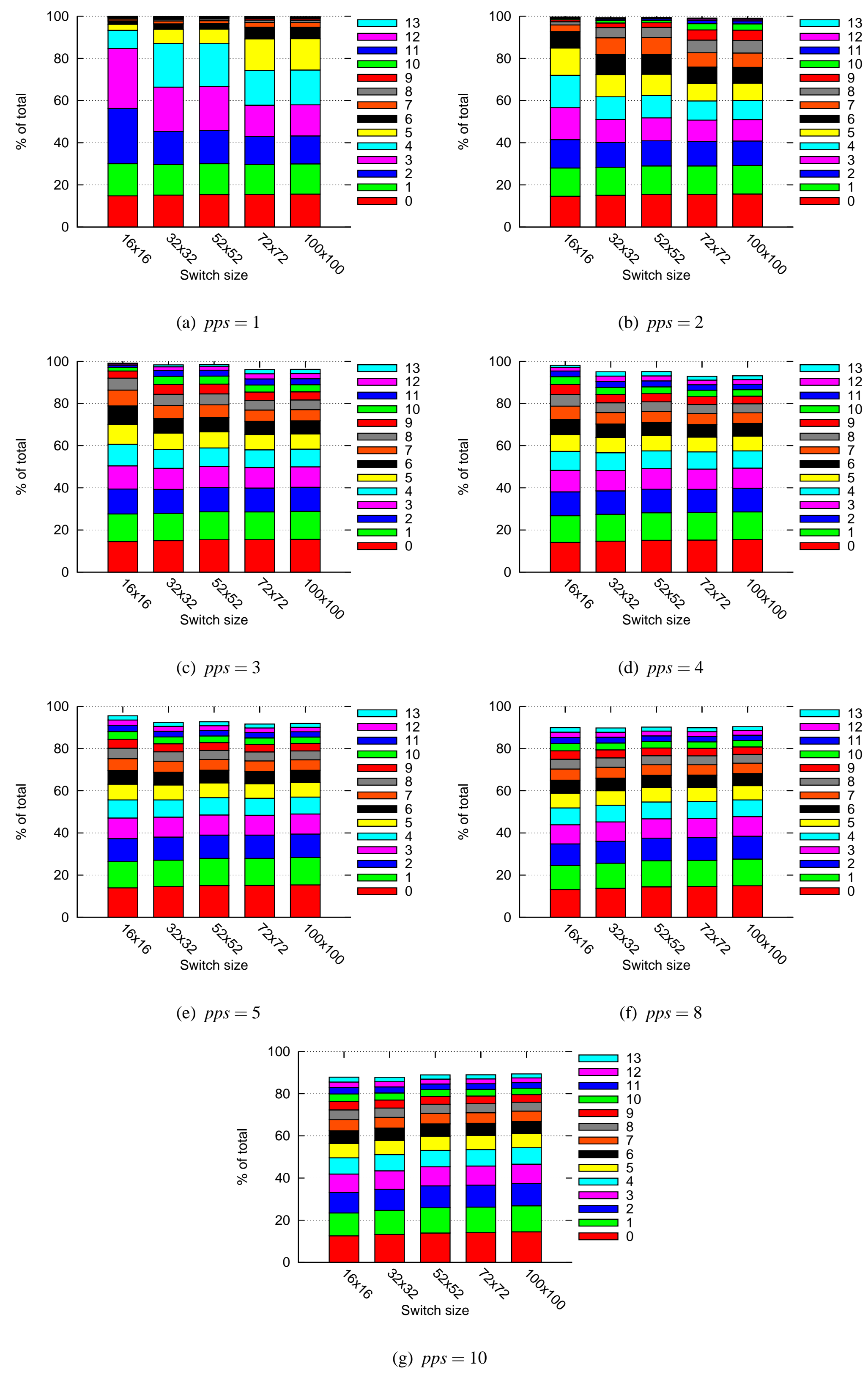
	Percentage of arriving packets that encountered										
	0	1	2	3	4	5	6	7	8	9	10
Switch size	packet(s) in the queue										
16×16	14.79	15.26	26.27	28.42	8.57	2.88	1.56	0.89	0.51	0.30	0.19
32×32	15.17	14.53	15.69	20.97	20.81	6.67	2.55	1.37	0.80	0.49	0.32
52×32	15.42	14.65	15.68	20.87	20.58	6.72	2.55	1.39	0.83	0.50	0.30
72×72	15.49	14.23	13.23	14.82	16.52	14.95	5.57	2.21	1.16	0.69	0.40
100×100	15.63	14.32	13.25	14.79	16.47	14.80	5.47	2.22	1.15	0.70	0.42

TABLE 2.12. VOQ occupancy as observed by an incoming packet for $pps = 2$ and $b = 6$.

	Percentage of arriving packets that encountered										
	0	1	2	3	4	5	6	7	8	9	10
Switch size	packet(s) in the queue										
16×16	14.60	13.44	13.43	15.18	15.33	13.01	7.68	3.19	1.52	0.89	0.56
32×32	15.06	13.33	11.78	10.92	10.70	10.42	9.63	7.98	4.82	2.16	1.14
52×32	15.45	13.56	11.88	10.91	10.57	10.13	9.45	7.92	4.84	2.20	1.15
72×72	15.52	13.47	11.63	10.11	9.08	8.40	7.64	6.82	6.02	4.82	2.95
100×100	15.67	13.54	11.62	10.09	9.03	8.33	7.52	6.77	6.02	4.83	2.97

TABLE 2.13. VOQ occupancy as observed by an incoming packet for $pps = 3$ and $b = 6$.

	Percentage of arriving packets that encountered										
	0	1	2	3	4	5	6	7	8	9	10
Switch size	packet(s) in the queue										
16×16	14.50	13.07	11.85	11.01	10.22	9.50	8.70	7.54	5.66	3.36	1.72
32×32	14.91	13.02	11.36	9.95	8.91	7.84	6.90	6.11	5.36	4.65	3.88
52×32	15.33	13.27	11.51	9.97	8.81	7.71	6.82	5.95	5.22	4.61	3.82
72×72	15.38	13.21	11.31	9.71	8.40	7.27	6.23	5.36	4.62	3.92	3.35
100×100	15.55	13.32	11.37	9.73	8.40	7.23	6.18	5.32	4.56	3.89	3.35



(a) $pps = 1$ (b) $pps = 2$

(c) $pps = 3$ (d) $pps = 4$

(e) $pps = 5$ (f) $pps = 8$

(g) $pps = 10$

FIGURE 2.14. VOQ occupancy as observed by an incoming packet for mean burst size 6.

TABLE 2.14. VOQ occupancy as observed by an incoming packet for $pps = 4$ and $b = 6$.

	Percentage of arriving packets that encountered										
	0	1	2	3	4	5	6	7	8	9	10
Switch size	packet(s) in the queue										
16×16	14.12	12.66	11.29	10.21	9.01	8.02	7.14	6.31	5.51	4.72	3.72
32×32	14.72	12.76	11.06	9.67	8.42	7.30	6.33	5.45	4.65	3.96	3.34
52×32	15.14	13.02	11.23	9.73	8.39	7.23	6.21	5.31	4.52	3.86	3.28
72×72	15.21	13.00	11.13	9.55	8.15	7.02	6.00	5.13	4.34	3.68	3.10
100×100	15.43	13.16	11.21	9.56	8.17	7.00	5.97	5.08	4.29	3.64	3.07

TABLE 2.15. VOQ occupancy as observed by an incoming packet for $pps = 5$ and $b = 6$.

	Percentage of arriving packets that encountered										
	0	1	2	3	4	5	6	7	8	9	10
Switch size	packet(s) in the queue										
16×16	13.95	12.40	10.94	9.79	8.58	7.46	6.47	5.65	4.95	4.25	3.61
32×32	14.51	12.57	10.93	9.47	8.18	7.08	6.08	5.20	4.46	3.78	3.24
52×32	15.04	12.89	11.06	9.53	8.21	7.02	6.00	5.08	4.35	3.68	3.13
72×72	15.05	12.87	11.03	9.44	8.05	6.87	5.87	4.99	4.25	3.59	3.05
100×100	15.33	13.04	11.12	9.47	8.05	6.86	5.85	4.96	4.21	3.56	3.01

TABLE 2.16. VOQ occupancy as observed by an incoming packet for $pps = 8$ and $b = 6$.

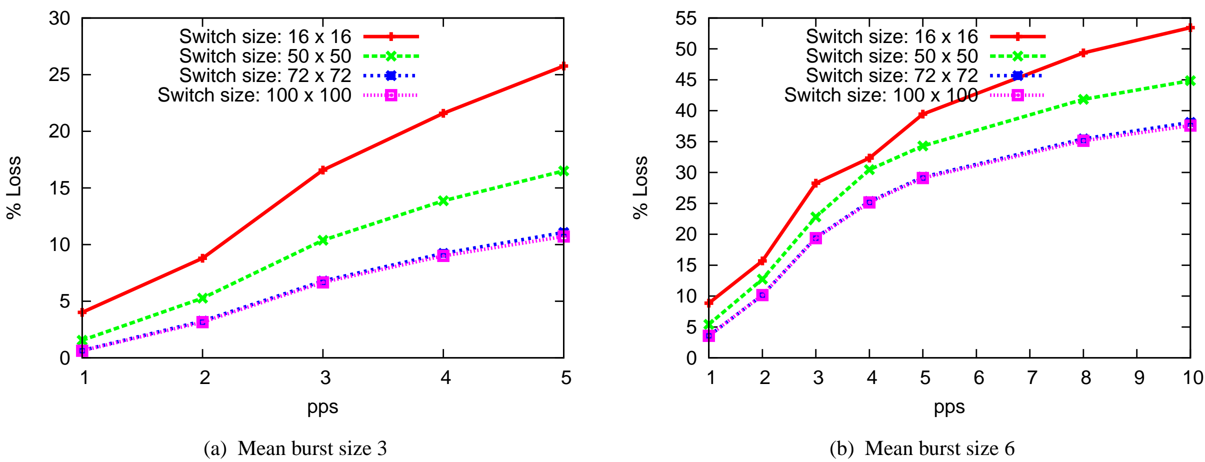
	Percentage of arriving packets that encountered										
	0	1	2	3	4	5	6	7	8	9	10
Switch size	packet(s) in the queue										
16×16	13.06	11.48	10.25	9.09	7.97	7.01	6.15	5.32	4.61	4.00	3.42
32×32	13.75	11.89	10.45	9.13	7.95	6.87	5.97	5.17	4.43	3.81	3.24
52×32	14.43	12.38	10.69	9.21	7.95	6.82	5.85	5.00	4.29	3.65	3.11
72×72	14.57	12.45	10.72	9.22	7.91	6.78	5.78	4.94	4.21	3.59	3.05
100×100	14.92	12.69	10.85	9.27	7.92	6.77	5.76	4.90	4.17	3.55	3.02

TABLE 2.17. VOQ occupancy as observed by an incoming packet for $pps = 10$ and $b = 6$.

	Percentage of arriving packets that encountered										
	0	1	2	3	4	5	6	7	8	9	10
Switch size	packet(s) in the queue										
16×16	12.55	10.88	9.73	8.71	7.71	6.81	6.02	5.26	4.61	4.04	3.52
32×32	13.25	11.38	10.02	8.77	7.69	6.72	5.86	5.09	4.40	3.84	3.31
52×32	13.93	11.96	10.40	9.01	7.81	6.74	5.82	5.02	4.31	3.71	3.17
72×72	14.11	12.08	10.44	9.02	7.80	6.73	5.78	4.98	4.27	3.67	3.14
100×100	14.49	12.34	10.62	9.12	7.82	6.71	5.75	4.92	4.21	3.59	3.07

Figure 2.15 presents the percentage of packets that are lost with $\lceil \ln n \rceil$ size VOQs for bursty traffic. The packet loss becomes appreciably higher even for $pps = 2$.

Packet Loss for Different VOQ Sizes: Just like with uniform random traffic, we also selected two switch sizes, 50×50 and 100×100, varied the VOQ sizes for each switch in the range $[1, \lceil \ln(n) \rceil]$, and observed the packet loss for each VOQ size. Figures 2.16 and 2.17 present the results. These results, like their uniform traffic counterpart, show that if the schedule generation time of a switch increases, thereby forcing an increase in *pps* to ensure stability for an identical packet arrival pattern, a fixed VOQ size leads to increasing number of dropped packets, especially for the lower VOQ sizes. As shown in Figure 2.16 and especially in 2.17, $pps = 1$ translates to very low VOQ size requirements, while ensuring low packet loss. This, however, is not true for higher values of *pps* and indirectly reinforces the fact that ensuring $pps = 1$ leads to big gains in delay (VOQ size is related to delay).



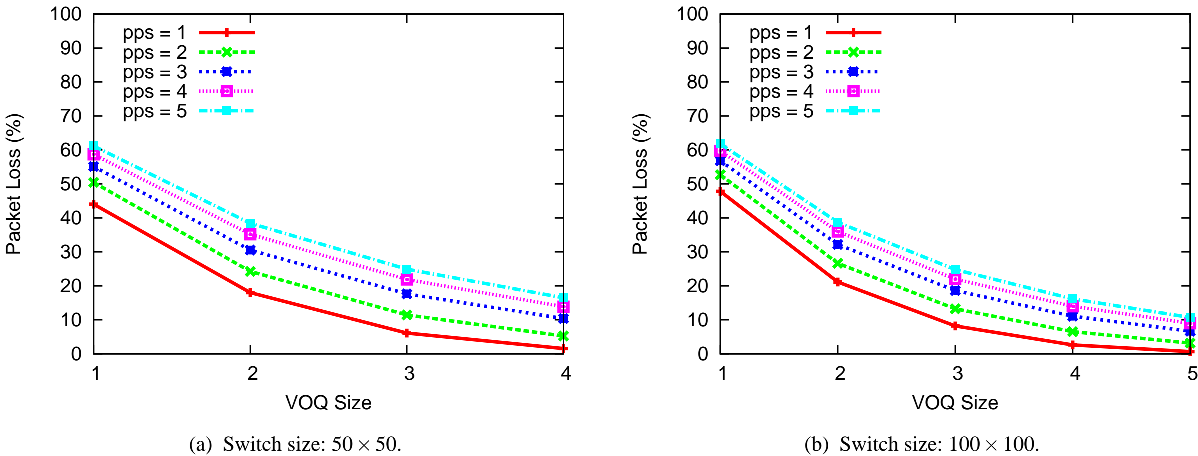
(a) Mean burst size 3

(b) Mean burst size 6

FIGURE 2.15. Percentage packet loss for bursty traffic with for different *pps*.

Different Frame Sizes:

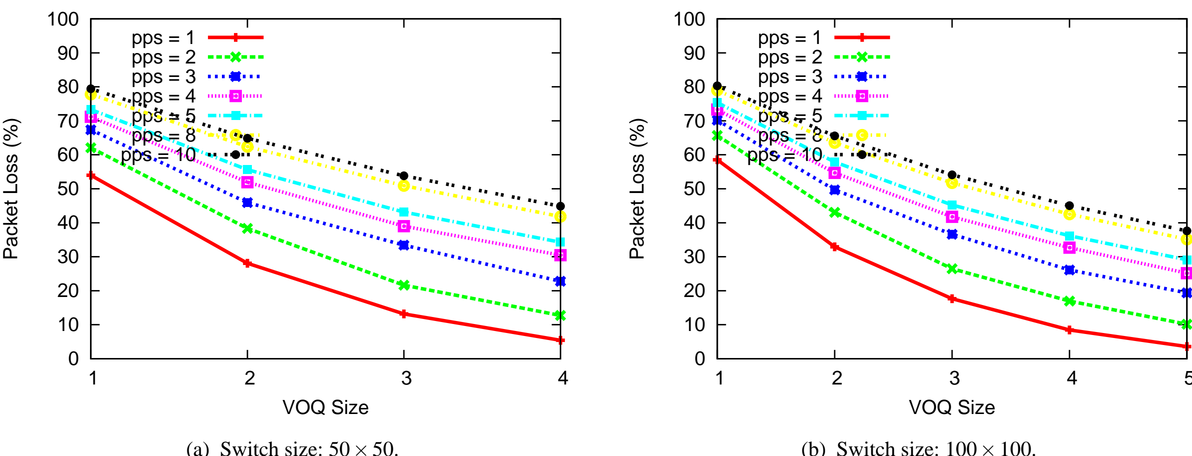
For bursty traffic we also varied the frame size in the interval $[1, 5]$ and observed the average delay for a 50×50 (resp., 100×100) switch and $1 \leq pps \leq 5$. Figure 2.18 (resp., Figure 2.19) shows the results for $b = 3$ (resp., $b = 6$).



(a) Switch size: 50×50.

(b) Switch size: 100×100.

FIGURE 2.16. Percentage packet loss for different VOQ sizes for traffic with mean burst size 3.



(a) Switch size: 50×50.

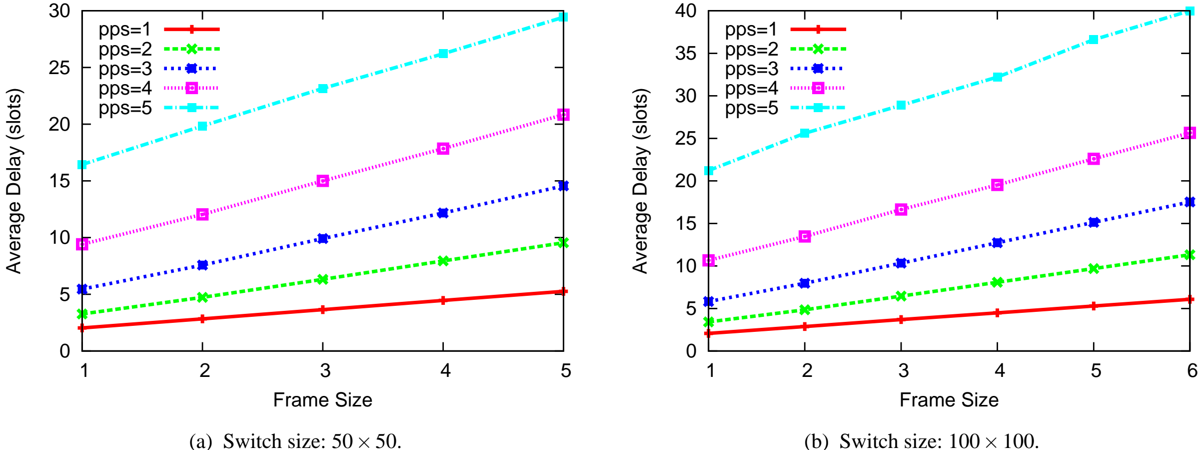
(b) Switch size: 100×100.

FIGURE 2.17. Percentage packet loss for different VOQ sizes for traffic with mean burst size 6.

2.6 Summary

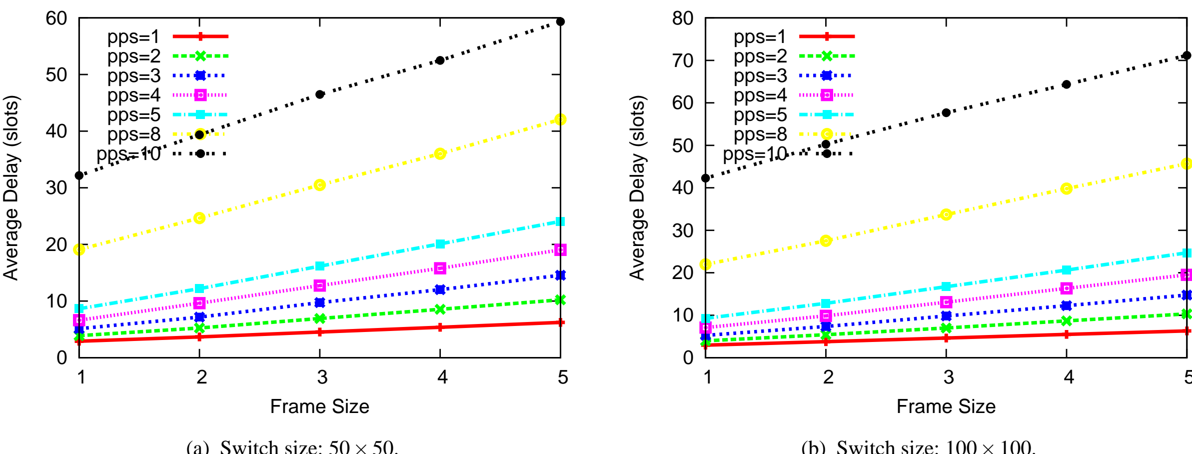
In this chapter we presented the results from our research on the effect of the relationship between packet-transmission time and schedule-generation time on performance parameters like delay, packet loss, and VOQ requirements for a crossbar-based input-queued switch. We showed that the logarithmic delay result of Neely *et al.* [59] does not hold for the more practical scenario where the packet-transmission time (slot) and schedule-generation time (round) are not the same.

We proved that logarithmic delay is achievable only when for every schedule, a single packet is transmitted between a scheduled input-output pair. This restriction is acceptable in practice only if the slots and rounds are of the same duration or if the packet arrival rate is low enough such that



(a) Switch size: 50×50. (b) Switch size: 100×100.

FIGURE 2.18. Average delay for different frame sizes for bursty traffic with mean burst size 3.



(a) Switch size: 50×50. (b) Switch size: 100×100.

FIGURE 2.19. Average delay for different frame sizes for bursty traffic with mean burst size 6.

in spite of the unutilized switch capacity (due to a round being much longer that a slot), stability is possible.

These results point to the following important conclusions.

- Logarithmic delay is possible only if the scheduling algorithm is fast enough to match the transmission time of a packet.
- Logarithmic queue length is sufficient to store all the packets when $pps = 1$.
- The investment in additional computational hardware to reduce pps could pay off in terms of reduced hardware requirement for buffer space.

Chapter 3
Fast Scheduling Algorithm on Mesh-of-Trees

3.1 Introduction

In Chapter 2 we discussed the relationship between packet-transmission time and schedule-generation time on performance parameters like delay, packet loss, and VOQ size for a crossbar-based input-queued switch. We proved that for uniform random traffic, when $pps \geq 2$ (i.e., the schedule generation time is more than the packet transmission time), packet delay is $\Omega(n)$. We also proved for bursty traffic with average burst size b that when $pps > 2b$, the delay is $\Omega(n)$. This implies that there is no alternative to reducing t_{round}, the time to schedule a round of packets on the switch, if one wants to achieve $O(\log n)$ packet delay. This underscores the need for a fast scheduling algorithm. We present one such algorithm in this chapter.

In an $n \times n$ input-queued switch, the set of non-empty virtual output queues (VOQs) determines the set of input-output connection requests. The main problem that the scheduling algorithm solves is that of selecting a subset of connection requests without violating the crossbar constraint. This problem is equivalent to a matching problem[1] on a $2n$-node bipartite graph, where the input ports constitute one of the partitions and the output ports constitute the other partition. Edges depict input-output connection requests. It is clear that every set of communications that satisfies the crossbar constraint corresponds to a (set of) matching(s) on this bipartite graph. Figure 3.1 shows an example of the relationship between crossbar scheduling and bipartite matching. Figure 3.1(a) depicts a 3×3 crossbar with colored bars at input ports showing waiting packets for output ports with the same color. Input-port 1 has queued packets destined for all the three output ports, input-port 2 has packets destined for output ports 1 and 2, and so on. Figure 3.1(b) shows the corresponding bipartite graph that has an edge corresponding to each input-output port pair with a

[1] A matching on a graph G is a subset M of edges of G such that no two edges are incident on a common vertex. A matching X is *maximal* iff there is no strict superset of X that is also a matching. Matching X is maximum iff no other matching has more edges than X. The matchings defined here are size-based matchings, in contrast to weight based matchings described in Footnote 2 on Page 42.

packet to transmit between them. Figure 3.1(c) shows a maximal matching of the bipartite graph of Figure 3.1(b). In Figure 3.1(d) crosspoints of the crossbar are set (shown as darkened circles inside boxes) to reflect the maximal matching of Figure 3.1(c).

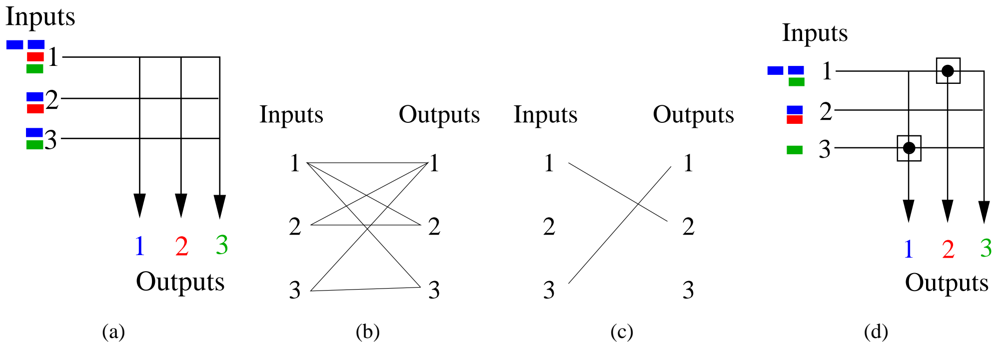


FIGURE 3.1. Example of equivalence between crossbar scheduling and bipartite matching.

While a matching ensures that the crossbar constraint is respected, a large matching allows a large number of packets to be transmitted in parallel. A maximum matching maximizes the number of parallel communications. The Fair Frame algorithm [59] uses a maximum matching. However, it is difficult to compute a maximum matching ($O(n^{1.5}\log n)$ [59] on a $2n$-node bipartite graph). A maximal matching on the other hand, is easier to compute ($O(n\log n)$ [59]) an can replace a maximum matching if the switch has a speedup or 2 [59] (speedup is defined in Footnote 1 on Page 12). Even this $O(n\log n)$ time is insufficient for unit *pps* for large values of n. In this section we devise a fast (polylog time) maximal matching.

Matching, in the context of input-queued switch scheduling, is a well-studied problem. Leonardi *et al.* [48] and Tassiulas *et al.* [76] used a maximum-weight matching[2] to generate a schedule. Giaccone *et al.* [31] presented three randomized algorithms for weight-based matching in a scheduling algorithm that achieve performance in terms of delay close to that of maximum-weight based algorithms. One of those algorithms has linear complexity and a second one nearly so ($O(n)$ and $O(n\log^2 n)$). Zheng *et al.* [81] presented a similar weight-based randomized matching algorithm for input-queued switches. The *i*SLIP algorithm designed by McKeown [55] generates a maximal-

[2] A weight-based matching uses a bipartite graph with edge weights (usually representing buffer loads). The sum of the weights of the edges in a matching is the largest possible for a maximum weight matching.

size bipartite matchings and is one of the most widely used distributed crossbar scheduling algorithms. The Cisco Gigabit Switched Router (GSR) uses *i*SLIP. The average time complexity of *i*SLIP is $O(\log n)$, but its worst case time complexity can be linear [55]. All these matching methods [48, 55, 76] result in an average packet delay of $O(n)$ slots.

Distributed switch scheduling methods, such as *i*SLIP, can generate a schedule in each round. This approach is faster, more efficient, and scales better for larger switches than sequential scheduling. In this chapter we present a distributed maximal-size bipartite matching algorithm to generate schedules for a crossbar switch. Our algorithm has a guaranteed polylog running time. Additionally, we use a reconfigurable structure that closely resembles a crossbar. More specifically, the approach is to use a mesh-of-trees (an implementable mesh-based structure with trees in rows and columns) inside a switch as the distributed control fabric. Coming back to the Fair Frame scheduling algorithm (see Chapter 2), each round computes a maximal matching from the set of input ports to the set of output ports corresponding to the buffered packets[3]. More precisely, let $I = \{i_1, i_2, \cdots, i_n\}$ and $J = \{j_1, j_2, \cdots, j_n\}$ be the sets of input and output ports. Construct bipartite graph $G = (I \cup J, E)$ such that $(i_x, j_y) \in E$ if and only if i_x has a packet destined to j_y. A matching ensures that each input connects (if at all) to just one output and vice versa. A maximal matching ensures that for uniform random traffic, if $pps = 1$ the delay is $O(\log n)$ [59]. A similar result holds for bursty traffic as well.

Bipartite matching on parallel architectures is also a well-studied problem. Fayyazi *et al.* [27] presented a linear-time PRAM algorithm for finding a maximum-weight matching in general bipartite graphs. Hanckowiak *et al.* [35] presented an algorithm for a distributed graph that computes a maximal matching of its own (unknown) topology in polylog number of communication rounds. Kelsen [44] presented a maximal-size bipartite matching algorithm that runs in time $O(\log^3 n)$ on an EREW PRAM with $\frac{m+n}{\log^3 n}$ processors. We base our mesh-of-trees algorithm on this algorithm. In the next section we briefly describe Kelsen's bipartite matching algorithm. We then design our

[3]Note that Fair Frame [59] computes a maximum and not a maximal bipartite matching, but a maximal bipartite matching algorithm will also guarantee stability and logarithmic delay with a speedup of 2. A speedup of 2 is common in today's routers. For example, Cisco CRS-1 has an internal speedup of 3.5 [39].

algorithm for the R-Mesh model, which can be efficiently simulated on a mesh-of-trees structure (proved in Lemma 3.3.3). Section 3.3 describes details about the R-Mesh architecture and its simulation on a mesh-of-trees, and Section 3.4 details the R-Mesh bipartite matching algorithm.

3.2 Kelsen's $O(\log^3 n)$ Bipartite Matching Algorithm on the PRAM

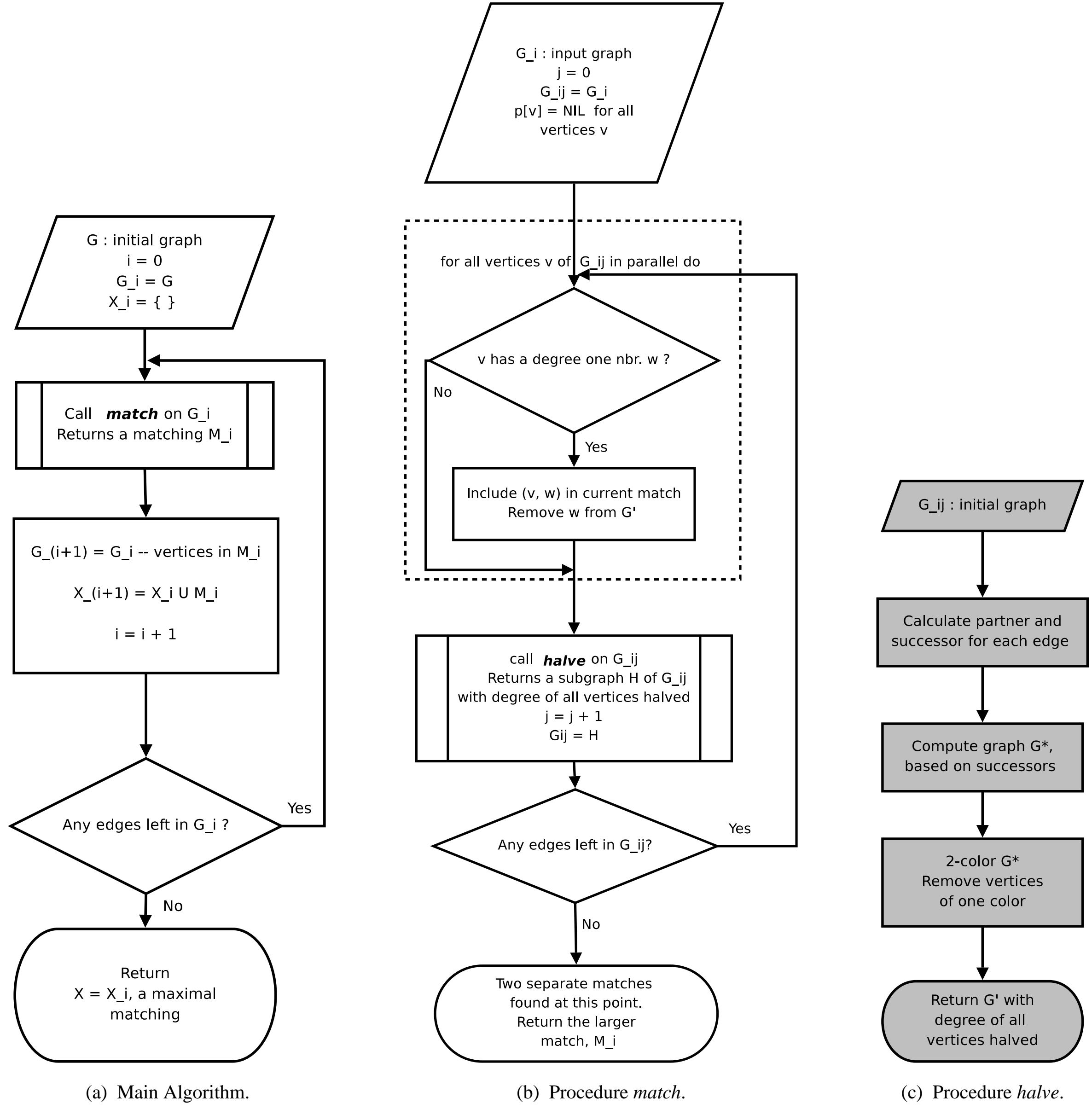
Kelsen [44] presented an algorithm that runs in $O(\log^3 n)$ time with $\frac{m+n}{\log^3 n}$ processors on an EREW PRAM to generate a maximal matching on a directed bipartite graph G with n vertices and m edges. The algorithm also applies to an undirected bipartite graph where each edge is replaced by two oppositely directed edges, as shown in Figures 3.3(a) and 3.3(b). Figure 3.2(a) shows the basic structure of the algorithm.

The general idea of the algorithm is to start with an empty graph (matching) X_0 and iteratively add vertices and edges to it so that X_i (output of ith iteration) is always a matching on the original graph G. The algorithm terminates after iteration α when adding more edges to X_α causes it to not be a matching. Figure 3.2(a) illustrates the structure of the algorithm.

The algorithm has two main procedures: *match* and *halve*. In iteration i, procedure *match* (Figure 3.2(b)) takes as input graph G_i and computes a matching M_i incident on at least $1/6$ of the edges of G_i. This guarantees that X will be a maximal matching after $O(\log n)$ iterations of the loop in Figure 3.2(a). The exact number of iterations to produce a maximal match depends on how "good" (large) M is. Procedure *match* is also iterative and in iteration j calls procedure *halve* for graph G_{ij} to halve the degree of each vertex in G_{ij}.

Procedure *halve* is called by *match* and accepts graph G_{ij} as its parameter. Let $G_{ij} = \{V_{ij}, E_{ij}\}$, and, for each vertex $v \in V_{ij}$, let $\delta(v)$ denote its degree. Then, procedure *halve* returns a subgraph $\mathcal{H} = \{V_{ij}, \mathcal{E}\}$ of G_{ij} such that for each vertex v of $\mathcal{H}$, the degree of v in $\mathcal{H}$ is either $\lceil \frac{\delta(v)}{2} \rceil$ or $\lfloor \frac{\delta(v)}{2} \rfloor$.

The main idea of *halve* is to compute an Euler partition of G_{ij}, i.e., a decomposition of G_{ij} into edge-disjoint paths with the property that every vertex with odd (resp., even) degree is an end-vertex of exactly one (resp., zero) path. Kelsen defined two relationships among the edges of the



(a) Main Algorithm. (b) Procedure *match*. (c) Procedure *halve*.

FIGURE 3.2. Maximal size bipartite matching by Kelsen.

bipartite graph, *partners* and *successors*, and used that to find the Euler partitions. Kelsen paired among edges incident on the same vertex u of G_i and defined them as partners. Further, for an edge (u,v) in G_i, define $successor(u,v) = partner(v,u)$. For example, in Figure 3.3(b) edges (a,u) and (a,v) are partners of each other, and so are (u,a) and (u,c), since both these pairs of edges are incident on the same vertex a and u respectively. Moreover, $successor(a,u) = partner(u,a) = (u,c)$. Intuitively, during generating matchings, if an edge is included in a matching, then neither its partner nor its successor can be included in that matching. Graph G^* represents the set of Euler partitions of G_{ij}. Figure 3.3(c) shows G^* corresponding to the bipartite graph of Figure 3.3(b).

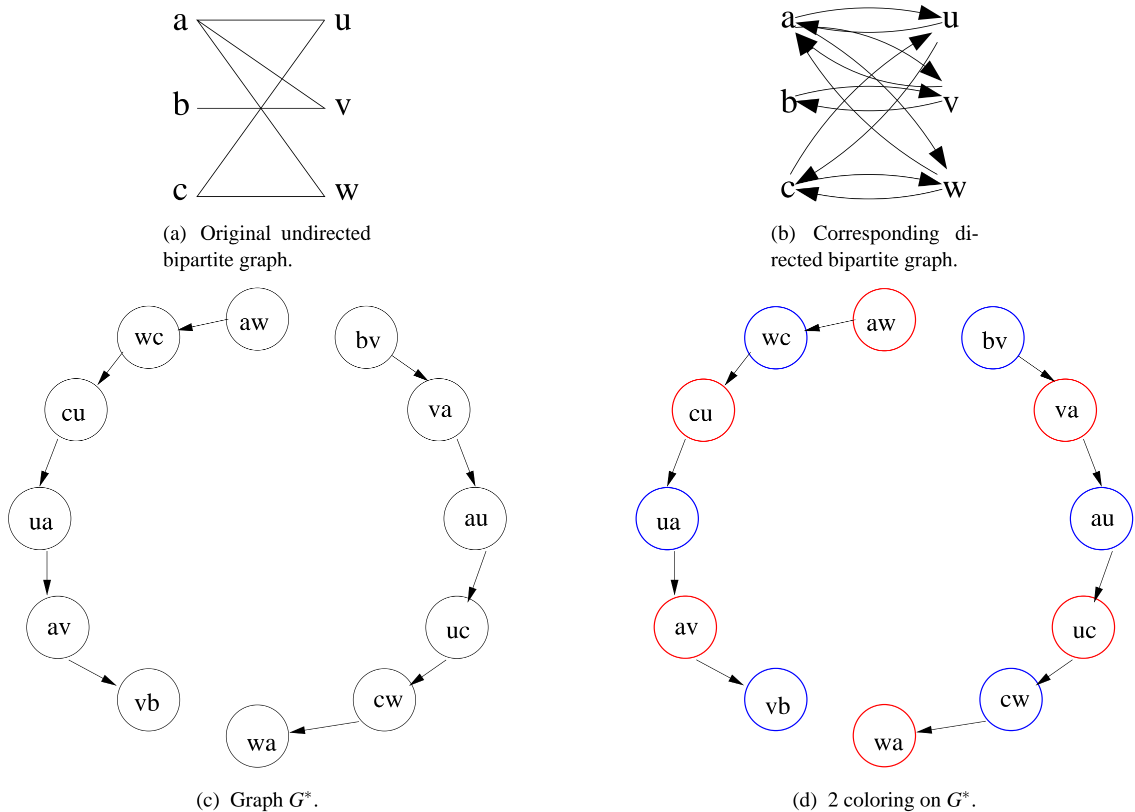


FIGURE 3.3. An example of Kelsen's algorithm.

Each edge of Figure 3.3(b) is a vertex in G^*. There is an edge in G^* between two vertices if one of the corresponding edges in G_{ij} is the *successor* of the other. Two coloring the edges on each path in G^* and selecting only one of the colors results in halving the degree of each vertex. For details of the correctness of this algorithm, refer to Kelsen [44].

Kelsen used parallel list ranking to accomplish the two coloring. We will take a different approach, exploiting the bus structure of the Euler partition as embedded in an R-Mesh, to achieve the same end.

3.3 Reconfigurable Mesh Preliminaries

A reconfigurable mesh (R-Mesh) is a well-studied reconfigurable architecture [77]. A linear directed R-Mesh (DR-Mesh) consists of an array of processing elements (PEs) connected by a mesh-based interconnection fabric. A PE connects to each of it four neighbors using a pair of oppositely directed links. Each PE has four pairs of ports, N, E, W, S, each of which has an incoming and an outgoing port represented by N_i, N_o, etc. A PE can internally connect an incoming port to any of its

outgoing ports to create a bus. We represent each internal connection inside a PE as an ordered pair, for example (N_iS_o) represents a connection from the north incoming to the south outgoing port[4]. Figure 3.4 shows a 3×5 DR-Mesh with 15 PEs. Figure 3.4 also shows a few pairs of buses: one connecting the W set of ports of PE(2, 0) with the N set of ports of PE(0, 1), another connecting the S set of ports of PE(2, 3) with the E set of ports of PE(2, 4), and a third connecting the W set of ports of PE$(0,0)$ with the W set of ports of PE$(0,2)$. Once a bus is created, any PE can write (resp., read) data to (resp., from) any of its ports that the bus spans.

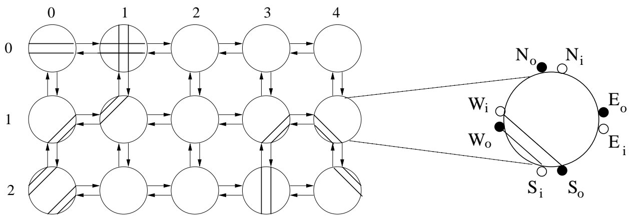


FIGURE 3.4. A 3×5 DR-Mesh.

In a general (undirected) R-Mesh, all ports and buses are undirected. The algorithm in this paper uses an R-Mesh to create pairs of parallel buses that are structured as though in a DR-Mesh. Although our algorithm does not use the directionality of these buses, we describe it as though it was run on a DR-Mesh because the description is simpler. Each PE connects corresponding pairs of ports (as in Figure 3.4), for example, if a PE connects N_i and S_o, then it also connects S_i and N_o. A 2×2 block of R-Mesh processors can emulate (in an undirected way) the connections of a DR-Mesh processor making such connections.

A horizontal-vertical R-Mesh (HVR-Mesh) is one in which every bus lies on a single row or column, or in other words none of the buses has any bends.

Lemma 3.3.1. (Matsumae and Tokura [53]): *An $n \times n$ HVR-Mesh can simulate each step of an $n \times n$ R-Mesh in $O(\log^2 n)$ time.* □

[4]In general, we use the notation (N,S) to represent the connections (N_iS_o) and (S_iN_o) simultaneously.

A $1 \times n$ HVR-Mesh is a segmentable bus[5]. A circuit-switched-tree (CST) (see also Chapter 4) is a binary tree-like structure where each leaf is a PE and each internal node is a switch.

Lemma 3.3.2. (El-Boghdadi *et al.* [24]): *An n-processor CST can simulate each step of an n-processor segmentable bus in $O(\log n)$ time.* □

The CST elements needed to simulate a segmentable bus are simple computational gates. Thus, the $O(\log n)$ time refers to computational gate delays. In a system with n components, a $\log n$ bit address is typically used. Even the decoder for this address would have $O(\log n)$ gate delay. Thus, the delay for emulating a segmentable bus would be a few clock cycles in practice.

An $n \times n$ HVR-Mesh is an $n \times n$ array of PEs with a segmentable bus in each row and each column. An $n \times n$ mesh-of-trees is analogous to an HVR-Mesh in that it is an $n \times n$ array of PEs with a CST in each row and column.

Lemma 3.3.3. *A $n \times n$ mesh-of-trees can simulate each step of an $n \times n$ R-Mesh in $O(\log^3 n)$ time.*

Proof. By Lemma 3.3.1, an HVR-Mesh can simulate any step of an R-Mesh in $O(\log^2 n)$ time. By Lemma 3.3.2, an n-processor CST can simulate a step of a row or column of a $n \times n$ HVR-Mesh in $O(\log n)$ time, so an $n \times n$ mesh-of-trees can simulate a step of an $n \times n$ HVR-Mesh in the same time. Hence, a mesh-of-trees can simulate an R-Mesh in $O(\log^3 n)$ time, which proves the lemma. □

Efficient R-Mesh solutions for various problems exist [77]. We describe solutions to two such problems – prefix sums and neighbor localization. We use these solutions later in R-Mesh algorithms.

Prefix Sums: For $0 \leq i \leq n$, the ith prefix sum of a set of bits $b_0, b_1, \cdots, b_n$ is $b_0 + b_1 + \cdots + b_i$. We use an $n+1 \times n$ R-Mesh as shown in Figure 3.5. We assume PE$(0,i)$ $(0 \leq i \leq n)$ holds b_i. To start, each PE$(0,i)$ broadcasts b_i down its column. Each non-top-row PE receives b_i and configures itself based on the value of b_i. If $b_i = 1$, then the PE configures itself as (NE, SW); that

[5] An n-PE segmentable bus is a linear bus with $n-1$ segment switches, each connected to a PE. PEs can open/close these switches to partition the bus into blocks of contiguous PEs. Each PE can write to its segment and all other processors incident on the segment can read the written data. A segmentable bus is equivalent to a one-dimensional R-Mesh.

is connecting N and E ports and S and W ports as two separate connections. Otherwise, the PE configures itself as (EW). These steps will create a bus connecting the W port of PE$(0,0)$ to the E port of PE$(n^{th}\ prefix\ sum, n)$ (shown in bold). PE$(0,0)$ then sends a signal on its W port. PE(i, j) (for any $0 \leq j \leq n$) receives that signal at its E port iff the jth prefix sum is i. For more details refer to Vaidyanathan and Trahan [77].

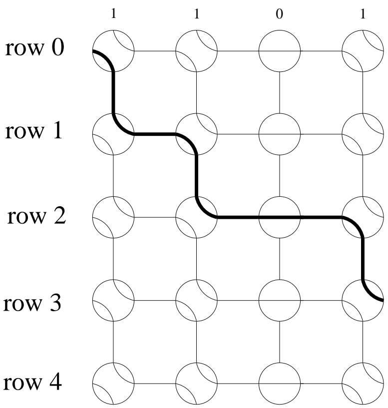


FIGURE 3.5. Prefix sums computation on an R-Mesh.

An R-Mesh is often implemented as a mesh-of-trees in practice. In that case, a one-dimensional R-Mesh with a row tree can compute prefix sums using the row tree in $\log n$ iterations.

Neighbor Localization: Given a one-dimensional R-Mesh with each PE flagged with a bit indicating it to be either active or inactive, neighbor localization finds a list of active PEs in the order of their position. In this algorithm, initially, each PE sets a variable *nbr* to *NIL*. Then, as shown in Figure 3.6, each inactive PE makes the connection (EW), while each active PE disconnects all ports internally. Now, each active PE writes its column index to the W port and reads the value written to its E port and stores it in *nbr*, at which point *nbr* contains the column index of the neighbor of each active PE. Refer to Vaidyanathan and Trahan [77] for more details.

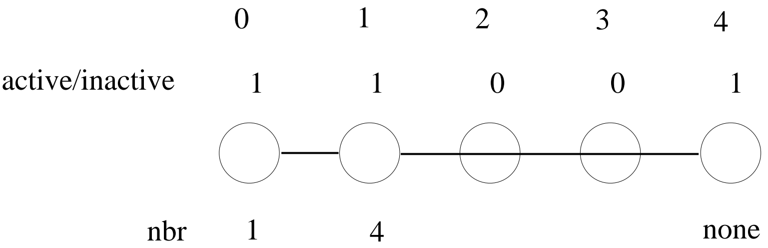


FIGURE 3.6. Neighbor localization on an R-Mesh.

3.4 R-Mesh Bipartite Matching Algorithm

For our purposes we assume an $n \times n$ crossbar switch has an $n \times n$ R-Mesh as the distributed computation fabric that create the schedule. We now place the matching algorithm in the context of the scheduling algorithm for the input-queued switch.

At the beginning of each frame, for each $1 \leq i,\ j \leq n$, processors calculate the traffic matrix in the form of a value $r(i,\ j)$ representing the number of rounds required to transmit all the packets in $queue(i, j)$ destined from input port i to output port j. The R-Mesh does not update $r(i,\ j)$ for incoming packets until the beginning of the next frame; note, however, that a PE decrements $r(i,\ j)$ by one whenever the corresponding input-output pair is present in a matching, that is, when a packet in queue (i, j) has been scheduled. We flag a PE($i,\ j$) as *active* (resp., *inactive*) if $r(i,\ j) > 0$ (resp., $r(i,\ j) = 0$). These flags constitute the adjacency matrix of graph G, input to the matching algorithm.

The R-Mesh implementation of each step of *match* (except for *halve*) is straightforward and can be achieved by neighbor localization. To accomplish the goal of the PRAM version of *halve* (Figure 3.2(c)), the R-Mesh likewise computes an Euler partition (but differently than the PRAM) and two-colors vertices (again differently than the PRAM). The DR-Mesh embeds an Euler partition of the input bipartite graph using two oppositely directed but parallel buses. While implementing *halve*, we create buses connecting active PEs in such a manner that each active PE connects to the next active PE (if any) on its row (resp., column) by a row (resp., column) bus-segment. We call two active PEs connected by a row (resp., column) bus-segment as row-neighbors (resp, column-neighbors). Additionally, at each active PE, each bus bends from a row to a column in one direction and from the same column to the row in the other. A limitation of any mesh-based structure is that no two buses in the same direction can overlap. We actually use this limitation to our advantage in creating an elegant bus structure that we use later to two-color the active PEs on a bus.

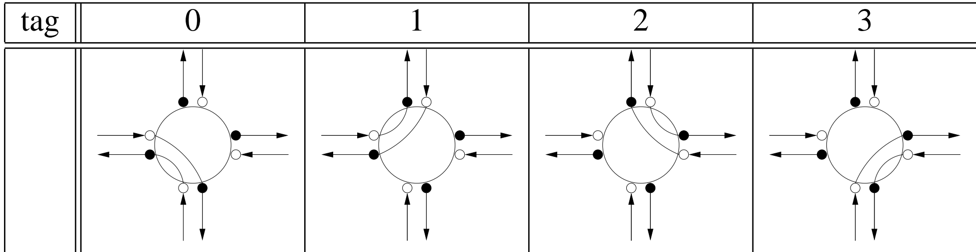
The following pseudo-code describes Procedure *halve*.

TABLE 3.1. *tag* of active PEs based on ps_{row} and ps_{col}.

	odd ps_{row}	even ps_{row}
odd ps_{col}	3	0
even ps_{col}	2	1

TABLE 3.2. Internal bus connections depending on *tag*.

tag	0	1	2	3



Procedure *halve*

<u>*Input:*</u> A $2n-$node bipartite graph G_i represented on an $n \times n$ R-Mesh as an adjacency matrix. Each edge in the graph is represented by a PE flagged as active. (PE(i, j) is flagged as active to denote the edge between vertices i and j.)

<u>*Output:*</u> A bipartite graph with degree of each vertex of the input bipartite graph halved, represented on an $n \times n$ R-Mesh. Each edge in the graph is represented by a PE flagged as active.

<u>*Algorithm:*</u>

1. Calculate prefix sums of active PEs in each row and each column. An active (resp., inactive) PE contributes a 1 (resp., 0). Each active PE receives its prefix sum from its row and from its column, denoted by ps_{row} and ps_{col}, respectively.

2. Each active PE disconnects all its internal connections and each inactive PE connects as (N,S) and (E,W).

3. Each active PE generates a *tag*, $0 \leq tag \leq 3$, using its ps_{row} and ps_{col} values as shown in Table 3.1.

4. Each active PE with $tag \neq 3$ makes the internal connections as shown in Table 3.2.

5. Call Procedure *elect-leader*, which flags the topmost-leftmost PE in each bus as leader.

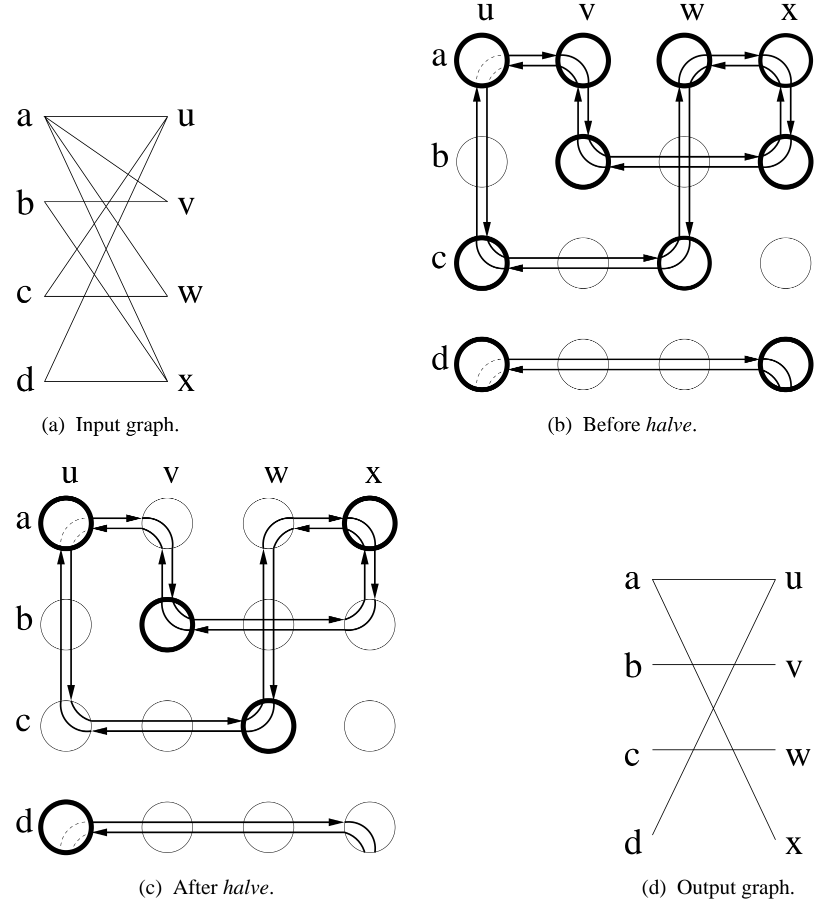
6. Each leader PE in row i and column j writes the value 0 to the E_o port. Additionally, if the leader PE had not received its own indices i and j in procedure *elect-leader*, it writes 1 to the S_o port.

7. Each active PE with some internal connection reads from both the incoming ports that have internal connections (see Table 3.2). If it reads a value 0 (resp., 1) either from its E_i or W_i (resp., N_i or S_i) port, then it makes its status inactive.

The main aspect of our implementation of *halve* is the way the bus structure creates the Euler partition. Figure 3.7 shows the execution as well as result of *halve* on a 4×4 DR-Mesh. Figure 3.7(a) shows the bipartite graph that is the input to *halve*. Kelsen's algorithm works on a directed graph. Hence, each edge in Figure 3.7(a) is actually two oppositely directed edges which we do not show here. PEs with bold (resp., thin) outlines in Figure 3.7(b) represent active (resp., inactive) PEs. Note that each active PE corresponds to an edge in the bipartite graph.

Kelsen's algorithm determined a partner and a successor for each edge. For example, edge (a,u)'s partner and successor will be (a,v) and (u,c) respectively. Our algorithm's bus creation mechanism automatically captures these partner and successor relationships. Edges of the bipartite graph denoted by two successive active PEs connected by a row (resp., column) bus are each other's partner (resp., successor).

Kelsen's algorithm created a graph G^* corresponding to the Euler partition of G_{ij}. A vertex in G^* corresponds to an edge in G_{ij} and an edge in G^* is based on successor relationships in G_{ij}. In our case, active PEs in the same row or column indicate edges in G_{ij} incident on the same vertex. Additionally, active PEs directly connected by a row/column bus indicate partner/successor. Since we are using a mesh structure, pairing neighboring active PEs on row and column directly produces an Euler partition. To achieve this pairing, Step 1 computes prefix sums for each active PE, and Step 4 creates internal connections pairing an odd PE with the next even one in each row and column by establishing a bus through each active PE spanning both its row and column neighbors. This bus corresponds to an Euler partition. In Kelsen's algorithm, edges (a,u) and (u,c) of Figure 3.7(a)



(a) Input graph. (b) Before *halve*. (c) After *halve*. (d) Output graph.

FIGURE 3.7. Procedure *halve*.

would have been nodes in G^* with an edge from (a, u) to (u, c). The bus directly connecting them in Figure 3.7(b) denotes the same.

For example, if we consider Figures 3.7(a) and 3.7(b), each PE in Figure 3.7(b) corresponding to an edge in Figure 3.7(a) is active. If we consider the edge $(b,\ v)$, then the corresponding PE, PE$(b,\ v)$ will have odd row and even column prefix sums. This signifies its row neighbor is to its right and column neighbor is above it. Accordingly, PE$(b,\ v)$ connects its north and east ports to make a bus spanning its row and column neighbors.

Once these buses corresponding to an Euler partition are created, we want to choose alternate active PEs on each bus to include in the matching. We achieve this by first picking a leader in each bus. The leader then picks one of the two oppositely directed but parallel buses and informs all

the others. Note that in the direction of the bus, active PEs alternate with bus entering on the row, leaving on the column and bus entering on the column, leaving on the row. We use this as the basis of two-coloring. In contrast, Kelsen used parallel list ranking to two-color the paths in his Euler partition.

Figure 3.7(c) shows the R-Mesh after the execution of *halve*. Note that half of the active PEs in each row and column have now become inactive. Figure 3.7(d) shows the resultant bipartite graph with the degree of each vertex halved.

We now present our procedure *elect-leader*. That procedure selects the topmost-leftmost active PE in each bus as the leader (the topmost-leftmost active PE will always have $tag = 3$).

Procedure *elect-leader*

Input: An $n \times n$ R-Mesh with one or more buses connecting active PEs. Each active PE has a *tag* in $[0,\ 3]$ associated with it.

Output: The topmost-leftmost PE in each bus flagged as the leader.

Algorithm:

1. Each active PE with $tag = 3$ and no internal connections repeats the following three steps until it receives either its own indices or no value:

    (a) Each active PE$(i,\ j)$ with $tag = 3$ writes the values i and j to its E_o and S_o port.

    (b) Each active PE$(i,\ j)$ with $tag = 3$ reads from its E_i (resp., S_i) port, and if there is a value, stores it in variables k_1 and ℓ_1 (resp., k_2 and ℓ_2).

    (c) i. If PE$(i,\ j)$ read a value at its E_i port in Step $1(b)$ and if $\ell_1 < j$ or $(\ell_1 = j$ and $k_1 < i)$, then make the internal connections corresponding to $tag = 3$ from Table 3.2.

    ii. If PE$(i,\ j)$ read a value at its S_i port in Step $1(b)$ and if $\ell_2 < j$ or $(\ell_2 = j$ and $k_2 < i)$, then make the internal connections corresponding to $tag = 3$ from Table 3.2.

2. Each active PE$(i,\ j)$ with $tag = 3$ and no internal connections additionally flags itself as *leader*.

The goal of *elect-leader* is to designate a single PE on each bus as leader. Our design of *elect-leader* ensures that the leftmost-topmost PE becomes the leader. The leftmost-topmost PE on each bus will always have $tag = 3$. However, each bus can have multiple PEs with $tag = 3$. For example, the longer bus in Figure 3.7(b) spanning eight active PEs has two PEs with $tag = 3$. The main idea in *elect-leader* is that in each iteration, each PE(i, j) with $tag = 3$ communicates with the next PE with $tag = 3$ along the bus in both the directions, and based on the information it receives from other $tag = 3$ PEs, PE(i, j) either drop out of contention from the leader election or participate in the next iteration.

Lemma 3.4.1. *The topmost-leftmost PE of any bus will have* $tag = 3$.

Proof. We prove this by contradiction. Let us assume that the topmost-leftmost PE of a bus has $tag = 0$. This means its ps_{row} calculated in Step 1 of *halve* is even, that is, the row contains an even number of active PEs including the leftmost in its row between column 0 and its column. This implies this PE will be connected to another PE to its left by a row bus-segment which contradicts our assumption. Similar arguments are valid for the other *tag*s thus proving the lemma. □

Lemma 3.4.2. *Procedure elect-leader elects a leader in* $O(\log n)$ *time.*

Proof. Let PE(i_1, j_1) and PE(i_2, j_2) be two *tag* 3 PEs directly connected by a bus. Since these are two distinct PEs, at least one of the relations $i_1 \neq i_2$ and $j_1 \neq j_2$ always holds. This means at least one of the four conditions of Steps $1(c)(i)$ and $1(c)(ii)$ is satisfied for one of these PEs. Hence, that PE drops-out of contention for being the leader. Since, the algorithm removes at least one out of two consecutive *tag* 3 PEs from contention in any iteration, in at most $\log n$ iterations all but one PE will be removed from contention. □

3.5 Time Complexity

An R-Mesh can calculate the prefix sums in *halve* in $O(\log n)$ time [18]. Procedure *elect-leader* will also take $O(\log n)$ time on an R-Mesh as each iteration of the loop eliminates at least half of the PEs on a bus with $tag = 3$ from the leader election. Hence, *halve* takes $O(\log n)$ time to complete.

Procedure *match* invokes *halve* until each vertex in the bipartite graph has a single neighbor. Since each iteration halves the degree of a vertex, $O(\log n)$ iterations of *halve* will reduce the degree of each vertex to at most 1. So, *match* completes in $O(\log^2 n)$ time.

As noted earlier, $O(\log n)$ iterations of *match* produce the maximal matching. Hence, the total time needed to generate the maximal matching on an R-Mesh is $O(\log^3 n)$.

In Lemma 3.3.3 we showed that an $n \times n$ mesh-of-trees can simulate a step of an $n \times n$ R-Mesh in $O(\log^3 n)$ time. Hence, our algorithm will run on a mesh-of-trees in $O(\log^6 n)$ time.

Theorem 3.5.1. *A maximal-size bipartite matching on a graph with n vertices in each partition can be generated in $O(\log^6 n)$ time on an $n \times n$ mesh-of-trees.* □

This complexity of the bipartite matching determines the time complexity of the Fair Frame algorithm. Hence, our matching algorithm in the Fair Frame scheduling method will complete a schedule in $O(\log^6 n)$ time.

Corollary 3.5.2. *Scheduling packets on an $n \times n$ switch can be completed in $O(\log^6 n)$ time on an $n \times n$ mesh-of-trees.* □

The above time complexity is the worst case. The best (and probably the average) case can be significantly faster. As noted in the remarks after Lemma 3.3.2, for all practical purposes the simulation of the segmentable bus runs in a few clock cycles. Thus in a way one, could say that the time complexity of the scheduling algorithm is $O(\log^5 n)$. Additionally, the simulation of Matsumae and Tokura [53] represents the worst case where arbitrary R-Mesh buses are possible. Our case involves only linear buses for which a better simulation may be possible. The $O(\log n)$ calls to *halve* assume nodes to have $O(n)$ degree. For a limited degree graph (such as, for example, when the frame has $\Theta(\log n)$ slots), *halve* will only need $\Theta(\log \log n)$ iterations.

3.5.1 Other Considerations

In Section 3.4, for simplicity we assumed that the number of rounds $r(i,\ j)$ needed to transmit all the packets in a queue $queue(i,\ j)$ was available at PE$(i,\ j)$. Even if we do not make that assumption, we can modify the algorithm to ensure no loss in efficiency. Since the frame size is

$O(\log n)$, at most $O(\log n)$ packets can arrive during a frame at an input i. So, we can interleave the queue-status update for the next frame with computation of the current frame. For example, if we assume input i receives a packet for output j and is connected to output k in the current frame, then i can broadcast a message on its row bus to indicate to PE$(i,\ j)$ a new arrival. At the start of the next frame, each PE$(i,\ j)$ would hold a current status of $queue(i,\ j)$.

3.6 Summary

In this chapter we designed a maximal-size bipartite matching algorithm that runs in polylog time on a mesh-of-trees. This is a significant improvement over existing sequential algorithms that run in linear time and distributed algorithms that are experimentally shown to run in logarithmic time, but can take linear time in the worst case.

Open problems include improving the algorithm by using the regularity of the bus structure in finding the leader. If we are successful, then this will speed-up the algorithm by a factor of $\log n$.

Chapter 4

Scheduling and Configuration of the Circuit-Switched Tree

In this chapter we present efficient scheduling and configuration algorithms for the *circuit-switched tree* (CST). The CST is an important interconnect used to implement dynamically reconfigurable architectures. A CST has a binary tree structure with sources and destinations as leaves and switches as internal nodes. These leaves communicate among themselves using the links of the tree. In a parallel processing environment these sources and destinations for a communication are processing elements (PEs). Key components for successful communication on a CST are scheduling individual communications and configuring the CST switches to establish data paths. Patterns and conflicts created by the positions of source and destination leaves generate various CST-switch scheduling and configuration problems. Here, we present algorithms to solve several such problems. The algorithms are distributed and require only local information, yet they capture the global picture to ensure proper communication. In the next section we present background information about the CST. The section includes applications of the CST, its structure, and a description of various communication sets created by relative positions of sources and destinations of communications on a CST. Sections 4.2 – 4.4 present the algorithms for CST-switch scheduling and configuration.

4.1 CST Background

Dynamic reconfiguration is a versatile computing technique. Various dynamically reconfigurable architectures like the reconfigurable mesh (R-Mesh) offer extremely fast solutions to many problems [77]. Though these theoretical models are very powerful, their assumption of constant delay for buses of arbitrary size and shape makes their implementation difficult.

Field-programmable gate arrays (FPGAs), on the other hand, provide a practical reconfigurable platform [8, 15, 62]. FPGAs typically depend on off-chip data for configuration. Thus they need large reconfiguration times making them unsuitable to implement extremely fast, R-Mesh-type, dynamically reconfigurable models. Though advances like partial reconfiguration [80], context-

switching [7, 45, 75], and configuration bitstream compression [62] have reduced reconfiguration times, they are yet to reach the speed and flexibility required by models like the R-Mesh.

The Self-Reconfigurable Gate Array (SRGA) architecture [73, 74] is an FPGA-like structure that can reconfigure using on-chip data at run time (see Figure 4.1). Hence, reconfiguration is very quick, possibly within a few clock cycles. Individual processing elements (PEs) use local information to reconfigure the SRGA at run time. This feature, that individual PEs act on local information to reconfigure at run time, makes the SRGA similar to the R-Mesh.

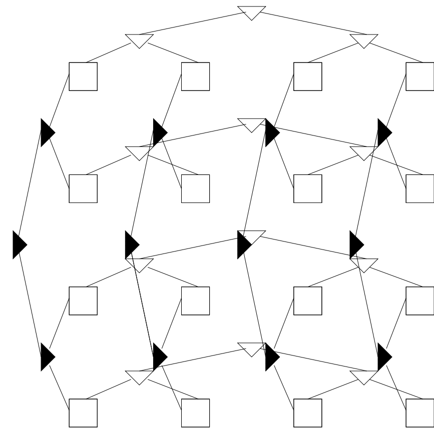


FIGURE 4.1. The Self-Reconfigurable Gate Array.

The CST [21] is a key component of the SRGA architecture acting as the building block for its interconnection fabric. The CST can implement other dynamically reconfigurable architectures with R-Mesh-like structures [23, 25]. The CST supports several communication patterns with important applications [24]. It may also serve as an interconnect in Infiniband-type networks [51] and networks-on-chip (NoCs) [3, 9, 10].

One of the most important aspects of using the CST to facilitate dynamic reconfiguration is the algorithm that configures the switches to establish dedicated data paths among PEs. The other, equally important, issue with using a CST for communications is, given a set of communications, how to schedule them in an efficient order. We define this scheduling of communications on the CST later.

A *communication* refers to the transfer of data from a source PE to a destination PE; that is, it is a point-to-point communication. Because the CST is a tree, each communication corresponds

to a unique path from the source leaf to the destination leaf. A *communication set* is simply a set of communications. Figure 4.2 depicts a communication set comprising three communications, $(0,1)$, $(2,7)$, and $(4,6)$ (each pair of numbers refers to source of the communication and the corresponding destination).

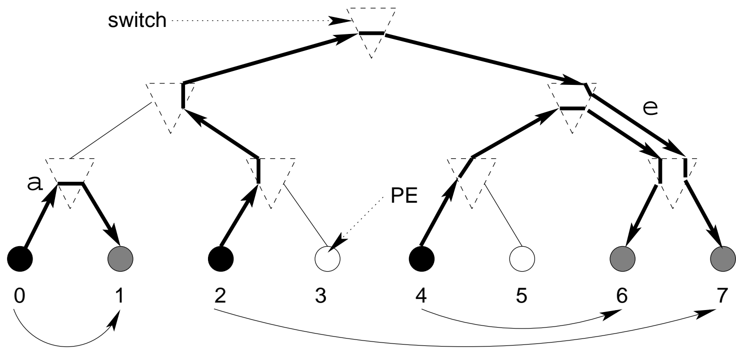


FIGURE 4.2. Communications on a CST; a dark (resp., shaded) leaf represents a source (resp., destination).

A CST can simultaneously perform multiple communications if their paths share no edge in the same direction. Define the *width* of a communication set as the maximum number of communications (paths) that share an edge in the same direction [21, 24]. The communication set of Figure 4.2 has a width of 2, as communications $(2,7)$ and $(4,6)$ use edge e in the same direction. On the other hand, the absence of either $(2,7)$ or $(4,6)$ would make the width of the communication set 1. The significance of the width is that a width-w communication set requires at least w rounds to complete. In the current example, the width-2 communication set requires two rounds to complete as $(2,7)$ and $(4,6)$ cannot be performed simultaneously (they share edge e).

In each round of performing communications, the CST configuration algorithm must select a subset of communications that are compatible, then configure switches accordingly. We define this partitioning of the communication set into blocks of compatible communications as *scheduling*. Note that a width-1 communication set has a trivial schedule since there are no conflicting communications.

4.1.1 Structure of the CST

The CST (Figure 4.2) is a balanced binary tree in which each leaf is a PE (or, in general, a source and/or destination of a communication) and each internal node is a switch. The edges of the CST

are full duplex links capable of carrying data in both directions simultaneously. Thus, each switch of the CST has three data inputs and three data outputs: an input-output pair to each of its two children and its parent (if any). Configuring a switch amounts to establishing connections from its inputs to its outputs. Several configurations are possible for any switch (Figure 4.3 shows some).

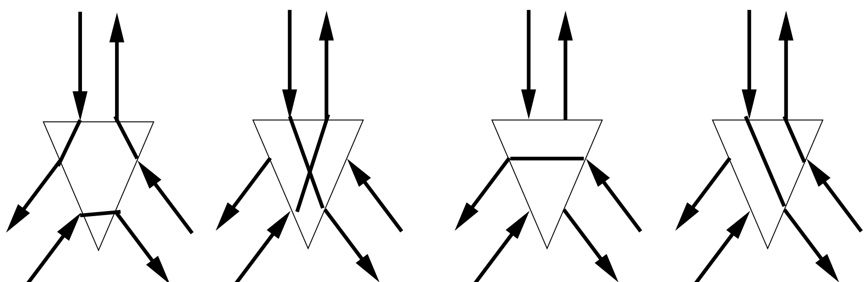


FIGURE 4.3. Some arbitrary configurations of a CST switch.

Configuring the switches of the CST establishes direct data paths among PEs. For example, in Figure 4.2, configuring switch a as shown plays a role in establishing a data path from PE 0 to PE 1. Hence, configuring the CST for a given set of communications amounts to configuring the switches to establish the required data paths. It should be pointed out that an input from a neighbor of a switch (child or parent) cannot be directed to the same neighbor. This restriction limits every path between leaves of an n-leaf CST to traverse at most $2\log n - 1$ switches. This upper bound on the path delay motivates the argument [7] that a signal can traverse up or down the tree very quickly (potentially in one clock cycle). The algorithms that we present here use a small constant number of traversals of the CST (upto 8 per round for the algorithm of Section 4.4).

4.1.2 Communications on a CST

In a width-w communication set ($w > 1$), two communications are *incompatible* if they share a common edge in the same direction. Communications are *source incompatible* (resp., *destination incompatible*) if they share an edge in the upward (resp., downward) direction. For example, communications $(2,7)$ and $(4,6)$ in Figure 4.2 are destination incompatible. Two incompatible communications cannot be simultaneously routed on a CST; they require two separate steps. Hence, a width-w communication set requires at least w steps to route in a CST. If a CST can perform these communications in exactly w steps, then call the communication set as *width partitionable* [21].

There exist communication sets that are not width partitionable. If a configuration algorithm optimally schedules and configures a width-partitionable, width-w communication set, then in each of w rounds, the width of the set of communications that is not yet scheduled decreases by one.

A *multicast* (s,D) consists of a source PE s and a set D of destination PEs; in a multicast, s sends a piece of information to all PEs in D. If $D=\{d\}$ is a single element set, then $(s,\{d\})$ or simply (s,d) is a *point-to-point communication*. We use the term *communication* loosely in this chapter to mean a point-to-point communication, and the term *multicast* denotes a set of multicasts. Figure 4.2 shows point-to-point communications $(0,1)$, $(2,7)$, and $(4,6)$. Figure 4.4 shows a set of two multicasts $M_1=(0,\{1,2\})$ and $M_2=(3,\{4,5,6\})$.

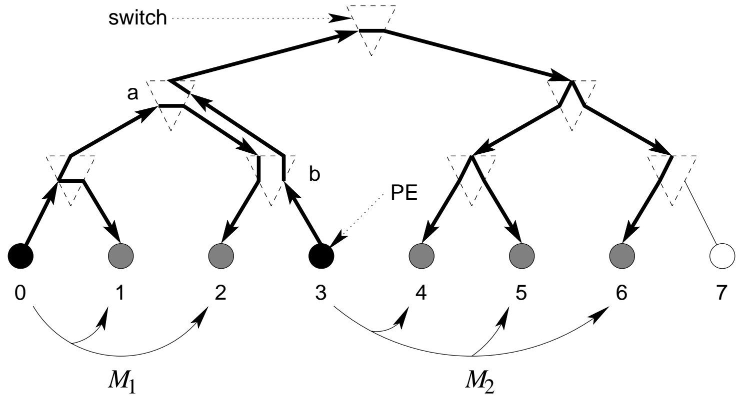


FIGURE 4.4. A multicast set.

A communication set is *right oriented* if, for every communication in that set, each destination is to the right of the corresponding source; similarly, it is *left oriented* if, for every communication in that set, each destination is to the left of the corresponding source. The communication set in Figure 4.2 is right oriented as each destination is to the right of its corresponding source. The algorithms presented here are for right-oriented communication sets; clearly they have trivial adaptations to left-oriented sets. Note that one can partition a communication set into a right-oriented set and a left-oriented set.

One can also classify communication sets according to the patterns the constituent communications form. A *well-nested* communication set is one in which the communications correspond to a balanced, well-nested parenthesis expression. Figures 4.2 and 4.5 show well-nested commu-

nication sets. Oriented, well-nested sets are width partitionable [21]. The well-nested property of a communication set also apply to multicasts. More formally, a well-nested multicast is one in which the source and any one destination of each multicast correspond to a balanced, well-nested parenthesis expression.

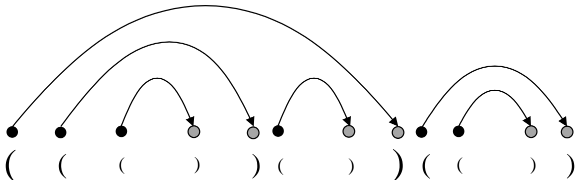


FIGURE 4.5. A well-nested communication set; a source (resp., destination) corresponds to a left (resp., right) parenthesis.

The SRGA architecture uses a CST for communication over each row/column. Sidhu *et al.* [73, 74] presented a CST routing algorithm that handles only one communication on a CST at a time. This limit is substantially short of the full communication and computation capacity of the architecture. The CST configuration algorithm of El-Boghdadi [21] allows multiple communications on a CST, but it restricts the communications to be edge-disjoint, i.e., no two communications can use any edge of the CST even in opposite directions.

4.1.3 CST Configuration

A CST configuration algorithm is key to communicating over a CST. The algorithm presented here configures CST switches to establish requisite data paths for a communication set.

Figure 4.6 shows the basic internal structure of a CST switch [21]. The switch contains two parts: the control unit and the data unit. The control unit receives information about communications that need to use that switch from the left and right children of the switch and uses them to generate information to send to the switch's parent. Subsequently, the control unit receives information from the parent switch instructing it to configure itself in some way to establish the required paths. The control unit uses this information to instruct the data unit to establish data paths. Furthermore, the control unit passes down appropriate control information for the configuration of its child switches.

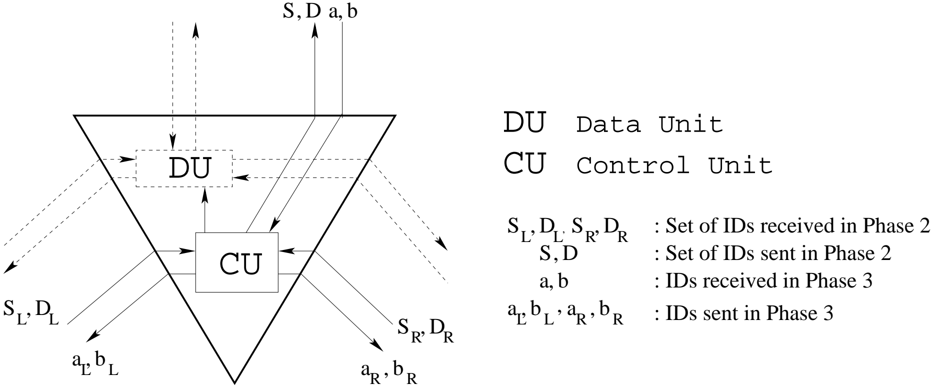


FIGURE 4.6. The internal structure of a CST switch.

We say that the source of a communication *matches* its corresponding destination at a switch u if the control information emanating from the source and the control information emanating from the destination meet at u. For this, u has to be the lowest common ancestor of the source and the destination. Further, the assumption that the communication set is right oriented means that for a source-destination pair to match, u must receive the source information from the left subtree and the destination information from the right subtree.

4.2 The Configuration Algorithm for Width-w Communication Sets

The algorithm we present has five phases, which correspond to one round of scheduling and switch configuration for a width-w communication set. Subsequent rounds repeat the same procedure, with the completed communications removed. For example, our algorithm will schedule the communications $(0,1)$ and $(2,7)$ of the communication set of Figure 4.2 in the first round followed by $(4,6)$ in the second round.

The five synchronous phases of the algorithm are as follows.

The General CST Configuration Algorithm

Phase 1: Assign an ID to each communication in the communication set to uniquely identify it.

Phase 2: Each PE that is a source or destination of a communication sends its ID to its parent. These IDs flow up the tree towards the root until meeting a match, with each switch recording the IDs that reach it.

Phase 3: Starting from the root, switches send control information down to the leaves. Based on this information, switches configure themselves. The sources and destinations of the communications for which the algorithm establishes a path receive back their own IDs, while the other sources and destinations receive a *null* symbol.

Phase 4: Source PEs for which Phase 3 configured their communication paths now write their data; corresponding destinations read.

Phase 5: Determine whether all communications have been completed. If any communication remains to be scheduled, then go to Phase 1. Otherwise, the algorithm terminates.

As pointed out earlier, these five phases will iterate at least w times for a width-w communication set.

Phase 1 assigns a common ID to the source and destination of each communication. In general, the IDs may need to be supplied to the algorithm as input. For certain communication classes, the algorithm itself can calculate IDs for the communications (as our algorithm for the well-nested class will do). Note that the IDs themselves need not necessarily be unique to uniquely identify each communication. As we will see later, for the well-nested class, non-unique IDs suffice.

In Phase 2, each switch receives a set of source IDs and a set of destination IDs from each of its children (these sets could be empty). If a switch u receives a source ID α from its left child and the same destination ID α from its right child, then that source-destination pair matches at u. In that case, u does not send ID α to its parent and instead sends to its parent only those IDs that do not match at u.

Formally, in Phase 2 (see Figure 4.6), each switch receives a set of source IDs S_L (resp., S_R) and destination IDs D_L (resp., D_R) from its left (resp., right) child. Each switch sends the sets of source IDs S and destination IDs D to its parent where $S = (S_L - D_R) \cup S_R$ and $D = D_L \cup (D_R - S_L)$. The set $S_L \cap D_R$ at any switch gives the IDs of the communications that match at that switch. Because the communications are right oriented, IDs in D_L and S_R do not find matches at the switch. We

note that for particular communication classes (Section 4.4), the algorithm could send a small set of identifiers without having to send the entire set.

In Phase 3, each switch configures itself based on the control information that it receives from its parent and the information received in Phase 2 from its children. It then generates control information for its children. At the end of this phase, scheduling is complete for some communications.

As shown in Figure 4.6, each switch receives two symbols a and b from its parent in Phase 3. Symbol a (resp., b) is the ID of a source (resp., destination) or is *null*. A switch receiving such an ID configures itself to establish data paths corresponding to that source or destination. The exact configuration depends upon the values of a and b as well as contents of the sets S_L, D_L, S_R, and D_R. After configuring, each switch generates control information (IDs) for its children and sends this down as a_L, b_L, a_R, and b_R as shown in Figure 4.6. We assume that the root receives $a = b = null$. As shown in Figure 4.7, we label the data ports of a switch as $\ell_i, \ell_o, r_i, r_o, p_i$, and p_o. Port ℓ_i (resp., ℓ_o) is the input (resp., output) port between u and its left child. Other ports are defined similarly. To configure a data path, u connects an input port to an output port.

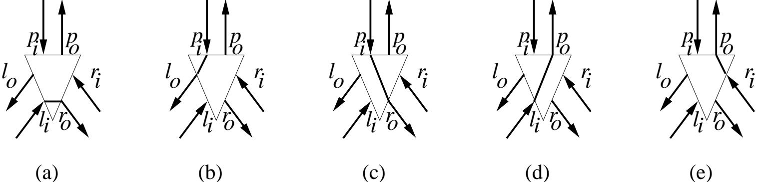


FIGURE 4.7. Some switch configurations.

In Phase 3, u will configure the data path as instructed by its parent. Additionally, u will route a communication (if any) that matches at u and is compatible with the communication(s) whose data path(s) u previously configured. In our algorithm, if both a and b are *null* for a switch u, then the parent of u is instructing u to not configure any data paths through u to or from its parent. Next, u will route a communication (if any) that matches at u configuring itself as shown in Figure 4.7(a). If u receives one or two IDs from its parent, then the parent is instructing u to configure data paths corresponding to the identified communication(s). Figure 4.7 shows various switch configurations. Note that a switch can connect its ports to simultaneously establish more than one of the five

configurations shown in Figure 4.7. One such configuration could be the superposition of those in Figures 4.7(a), 4.7(b), and 4.7(e).

The control information received by a PE at the end of Phase 3 informs it whether the CST contains the path for its communication. Those PEs whose paths have indeed been configured communicate, while the remainder again participate in the next round.

Figure 4.8 presents Phase 3 of the algorithm in pseudocode. Note that some cases, such as $a \in S_R$ and $b \in D_L$, can occur simultaneously, so the switch makes both port connections and sends out both symbols.

In Phase 5, after the completion of data transmission between the source and destination PEs of the configured communication(s), the algorithm determines the presence of any communication(s) not yet configured. Any source or destination PE that received a *null* symbol in Phase 3 sends a 1 to its parent. All the switches forward the OR of the symbols that it received from its children. If the result at the root is 1, then the root broadcasts a control signal instructing the PEs to initiate a new round.

Figure 4.9 shows an example of the execution of one round of the general CST configuration algorithm. Figure 4.9(a) shows three communications with IDs 1, 2, and 3. As shown in Figure 4.9(a) during phase 2 of the algorithm, each source/destination PE sends its ID to its parent, and these IDs flow up the tree until meeting their matches. For example, IDs corresponding to communication 2 match at the switch p. Hence, they are not propagated up the tree beyond p.

Figure 4.9(b) depicts Phase 3. In Phase 3, starting from the root, switches send control information (shown by numbers with boxes around them) down the tree, and based on this information each switch configures itself before sending the control information down. For example, the root picks the matching communication 1 and configures itself appropriately and sends down the control information corresponding to communication 1 as shown in the figure. Each switch configures itself based on the control information and the IDs that it received in Phase 2. If a switch does not

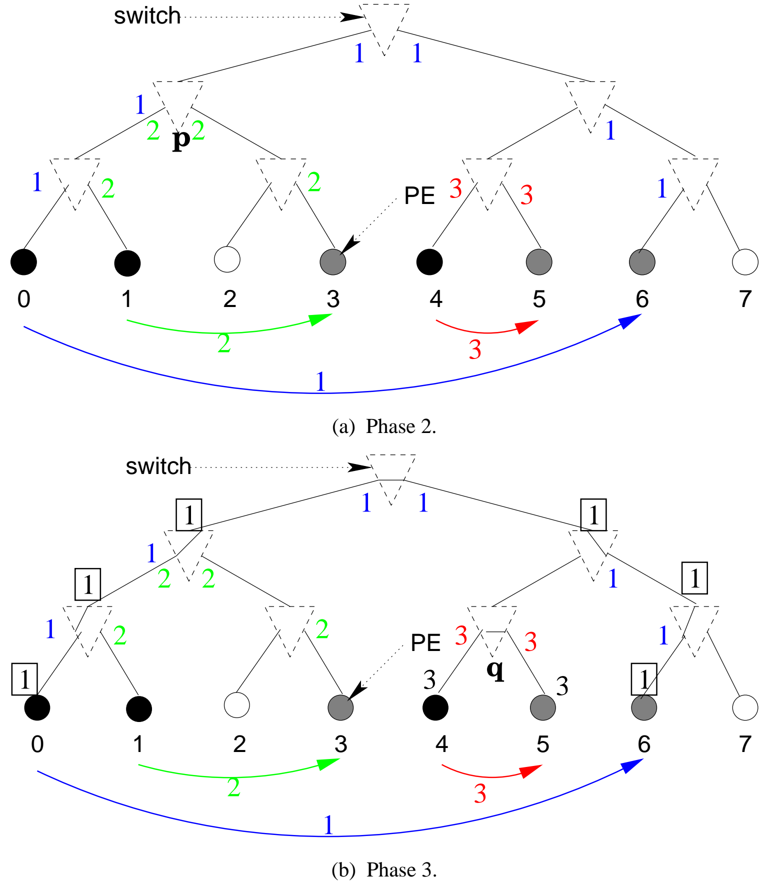
```
input : Indices a,b from parent (if any), sets S_L,D_L,S_R,D_R from Phase 2.
output : Internal switch configuration; indices a_L,b_L (for left child) and
         a_R,b_R (for right child).


Initialize a_L,b_L,a_R,b_R ← null
disconnect all ports of u
if a = b = null then                    /* u receives null symbols from its parent */
    if S_L ∩ D_R ≠ ∅ then
        e ← any one element of S_L ∩ D_R
        connect l_i to r_o                          /* Configuration */
        a_L ← e                                     /* Symbols for children */
        b_R ← e
    endif
else                                    /* u receives an ID from its parent */
    if a ∈ S_R then
        connect r_i to p_o                          /* Configuration */
        a_R ← a                                     /* Symbol */
    endif
    if b ∈ D_L then
        connect p_i to l_o                          /* Configuration */
        b_L ← b                                     /* Symbol */
    endif
    if a ∈ S_L − D_R then
        connect l_i to p_o                          /* Configuration */
        a_L ← a                                     /* Symbol */
    endif
    if b ∈ D_R − S_L then
        connect p_i to r_o                          /* Configuration */
        b_R ← b                                     /* Symbol */
    endif
    if a_L = b_R = null and S_L ∩ D_R ≠ ∅ then
        e ← any one element of S_L ∩ D_R
        connect l_i to r_o                          /* Configuration */
        a_L ← e                                     /* Symbols */
        b_R ← e
    endif
endif
```

FIGURE 4.8. Pseudocode for Phase 3 of the algorithm.

receive any control information from its parent (switch q), it is free to establish paths from any matching communication and send appropriate control information down the tree [1].

[1] A switch u is at liberty to establish connections for a communication matching at it even when u receives some control information (unlike b), as long as the matched communication is compatible with the communication indicated by the control information from the parent.



(a) Phase 2.

(b) Phase 3.

FIGURE 4.9. An example of the general CST configuration algorithm.

We now prove the correctness of the algorithm. For the proof we assume that the IDs assigned in Phase 1 are unique.

Lemma 4.2.1. *The general CST configuration algorithm establishes connections between matching source and destination pairs.*

Proof. Let c be any communication that matches at a switch u. Further, let z be the ID assigned to c in Phase 1. As IDs are unique, no other communication will be assigned the ID z. This ensures that the corresponding source and destination pairs of c will match at switch u and with no other destination or source.

In Phase 2, each of the source and the destination of c will send z to its parent. Switches forward z up the tree until reaching u. Switch u will receive z from both children simultaneously and, according to the algorithm, will not forward ID z to its parent. So, in Phase 2, each switch in the path from source or destination of c receives information about c.

In Phase 3 of some round, u will schedule c (see Figure 4.8). (This will happen when either u does not receive any ID from its parent or u receives ID(s) from its parent to configure a source (resp., destination) in its right (resp., left) subtree.) According to the algorithm, u connects ℓ_i to r_o and sends z to both children. The children of switch u will configure themselves, and each will send z down to the child from which it received z in Phase 2. In effect, z will retrace back the path it traversed in Phase 2, thereby establishing a simple path from the source of c to u then to the destination of c. □

Theorem 4.2.2. *The general CST configuration algorithm correctly schedules all communications in any finite communication set.*

Proof. Lemma 4.2.1 proves that for any communication, the algorithm will establish correct connections in some round. Hence, over all the rounds (a finite number), the algorithm will correctly connect each matching source-destination pair. □

There are, however, communication classes for which the algorithm completes the configuration and scheduling in more than the optimal number of rounds. Theorem 4.2.3 sets an upper bound on the number of rounds that the algorithm spends to schedule a width-w communication set.

Theorem 4.2.3. *The general CST configuration algorithm establishes connections for all communications of a width-w communication set in* $2w-1$ *rounds.*

Proof. Let c_1 be any arbitrary communication of a width-w communication set, matching at an arbitrary switch u. Hence, u will connect its left and right child to establish a connection corresponding to c_1 in some round.

By definition of a width-w communication set, at most $w-1$ communications are source incompatible with c_1 and at most $w-1$ communications are destination incompatible with c_1. So, at most $2w-2$ rounds include a scheduled communication incompatible with c_1. Since during any round, if no scheduled communication is incompatible with c_1, then u will necessarily schedule c_1 (by Phase 3 of the general CST configuration algorithm), hence the algorithm will schedule c_1 during any round on or before round $2w-1$, which proves the theorem. □

Remark: Erlebach *et al.* [26] in their research related to wavelength routing on directed fiber trees proved that any greedy algorithm needs at least $5/3w$ wavelenghts to route a set of communication request of load[2] w. Wavelength routing problem is similar to CST scheduling. Hence, we expect (suitable modified versions) of our algorithm to apply to wavelength routing as well.

4.3 Width-1 Communication Sets

The algorithm presented in Section 4.2 works correctly for any oriented, width-w communication set. The case where the width of the communication set is one merits special attention. In a width-1 communication set, no two communications share any edge of the CST in the same direction. We now show that this algorithm is optimal for any width-1 communication set. As earlier, we consider right-oriented sets.

Theorem 4.3.1. *The algorithm optimally schedules all the communications of an oriented, width-*1 *communication set in one round.*

Proof. Let C be any right-oriented width-1 communication set, and suppose $c \in C$ is a communication that is not scheduled in the first round by the algorithm. Let u be the switch at which c matches.

As C is of width 1, no other source (resp., destination) in the left (resp., right) subtree of u matches at u or at any switch above u. If u receives $a = b = null$ in Phase 3 of round 1, then according to the algorithm u will schedule c. Since c was not scheduled in round 1, u must receive an ID in that round. If at u, $a \neq null$ (resp., $b \neq null$), then a (resp., b) must be the ID of a source (resp., destination) in u's right (resp., left) subtree. The algorithm will configure u for c, contradicting that c is not routed in round 1. □

The general algorithm can also be simplified for a width-1 communication set as shown in the following lemma.

Lemma 4.3.2. *A modified algorithm with Phase 5 removed and Phases 2 and 3 merged is sufficient to route all the communications of a width-*1 *communication set.*

[2] Load in [26] is analogous to our width

Proof. By Theorem 4.3.1, the algorithm routes communications of a width-1 communication set in a single round. This makes Phase 5 redundant.

Because exactly one path connects each source-destination pair in a CST and because, in a width-1 communication set, no source-destination pair conflicts with any other, then each switch in Phase 2 can configure itself according to the source and destination IDs that it receives. If it receives a matching pair, then it connects the corresponding ports. If it receives an unmatched source, then it connects that incoming port to the outgoing parent port, and similarly for an unmatched destination. Because of the absence of conflict, it is not necessary to wait for control information from Phase 3. Consequently, Phases 2 and 3 can be merged into a single phase (see, for example, the algorithm in Section 4.3.1) making one pass from the leaves to the root, where switches configure themselves as soon as they receive information from their children. □

4.3.1 Modified CST Configuration Algorithms for Width-1 Sets

In this section we detail two algorithms for configuring the CST for oriented, width-1 communication sets, the first for well-nested point-to-point sets and the second for multicast sets. As before we assume that each PE holds only local information. Specifically, each PE holds whether it is a source, a destination, or neither. That is, a source is not aware of the identity of its destination and vice versa.

Structure of the Algorithms:

1. Each leaf (PE) generates an initial symbol that reflects its status in the communication in question; for example, a PE may indicate that it is a source or destination or neither.

2. Symbols propagate up the tree from the leaves to the root configuring switches on their way. Specifically, each switch receives two input symbols, α and β, from its children and produces an output symbol $f_s(\alpha,\beta)$ for its parent. (The root also generates this output symbol and then ignores it.) The switch also produces a second output $f_c(\alpha,\beta)$ to configure the data path(s) of the switch.

3. The algorithm completes when the root configures itself.

We will call an algorithm with the above structure a *one-pass* algorithm, as one traversal of the tree from leaves to the root suffices for the configuration. The algorithms in this paper use functions f_s and f_c that can be implemented with simple combinational logic. This allows for speedy configuration and data transmission, possibly within a few clock cycles [21, 74]. The main task of the configuration algorithm is to translate local information at the PEs to global information about the entire communication set. Specifically, each PE may be aware only of its status as a source, destination, or neither. This in itself is not sufficient to configure the switches. For example, although PEs 0 and 2 of Figure 4.2 are both sources, the switches at their parents assume different configurations (because of the information in PEs 1 and 3). Similarly, PEs 1 and 7 are both destinations, but their parents assume different configurations. In general, it is possible for the configuration of a switch quite far from PEs to be affected by the information at the PEs of its subtree.

One could view the configuration algorithm as a distributed algorithm initiated at the leaves and triggered by input symbols at the switches. Not all communication sets are amenable to this manner of handling.

Specifying the Algorithms:

Defining the following will specify a configuration algorithm of the form described as above.

- Set $\mathcal{C}$ of configurations of a switch. This is the same as that described by El-Boghdadi *et al.* [24]. Tables 4.2 and 4.4 show the configurations used in this work.

- Symbol set $\mathcal{S}$.

- Initial symbol assignment for PEs.

- Symbol function $f_s : \mathcal{S} \times \mathcal{S} \longrightarrow \mathcal{S}$.

- Configuration function $f_c : \mathcal{S} \times \mathcal{S} \longrightarrow \mathcal{C}$.

4.3.2 Oriented, Well-nested, Width-1 Point-to-Point Communication Sets CST Configuration Algorithm

Now we present the CST configuration algorithm for oriented, well-nested, width-1, point-to-point communication sets.

Well-Nested, Width-1 CST Configuration Algorithm

- $\mathcal{S} = \{\mathtt{s},\mathtt{d},\mathtt{b},\mathtt{n}\}$, where s denotes a source, d denotes a destination, b denotes a situation where both a source and a destination exist in a subtree, and n denotes neither a source nor a destination. (Elements of $\mathcal{S}$ are in a different font compared to s and d used to denote a source or a destination PE.)

- Initial symbol assignment: A leaf sends symbol s (resp., d or n) to its parent iff it is a source (resp., destination or neither).

- Symbol function: See Table 4.1. Blank cells in Tables 4.1 and 4.2 correspond to impossible situations.

- Configuration function: See Table 4.2.

TABLE 4.1. The function f_s for well-nested, width-1 CST configuration algorithm.

f_s	s	d	n	b
s		n	s	s
d	b		d	
n	s	d	n	b
b		d	b	b

We now address the correctness of the well-nested, width-1 CST configuration algorithm. For a 2^p PE CST, arrange the nodes in $p+1$ levels, $0,\cdots,p$, with PEs at level 0 and the root at level p. Let $\mathcal{T}_u$ denote the subtree rooted at any node u. Subtree $\mathcal{T}_u$ contains a *matched* source iff the source and its corresponding destination are both leaves of $\mathcal{T}_u$. Subtree $\mathcal{T}_u$ contains an *unmatched* source if the source is in $\mathcal{T}_u$ but the corresponding destination is not. Define matched and unmatched destinations similarly.

TABLE 4.2. The function f_c for well-nested, width-1 CST configuration algorithm.

f_c	s	d	n	b
s				
d				
n				
b				

Lemma 4.3.3. *Let u be any node at level ℓ, where $0 \leq \ell \leq p$, of the CST. Under the well-nested, width-1 CST configuration algorithm, let u generate symbol $\sigma \in \mathcal{S}$ to send to its parent, if any. The following assertions hold.*

1. *If $\sigma = \mathtt{s}$, then $\mathcal{T}_u$ has an unmatched source s and the algorithm establishes a path from s to the parent of u.*
2. *If $\sigma = \mathtt{d}$, then $\mathcal{T}_u$ has an unmatched destination d and the algorithm establishes a path from the parent of u to d.*
3. *If $\sigma = \mathtt{b}$, then $\mathcal{T}_u$ has an unmatched source s and an unmatched destination d. Moreover, the algorithm establishes paths from the parent of u to d, and from s to the parent of u.*
4. *If $\sigma = \mathtt{n}$, then $\mathcal{T}_u$ has no unmatched source or destination.*
5. *The algorithm establishes paths between all $($matched$)$ source-destination pairs of $\mathcal{T}_u$.*
6. *All unspecified entries of Tables 4.1 and 4.2 represent impossible situations.*

Proof. We prove the correctness of Assertion 1 by induction on the level of switches. As a base case, if u is at level $\ell = 0$, then u will generate $\sigma = \texttt{s}$ iff u is a source of a communication. Hence, $\mathcal{T}_u$ indeed has an unmatched source s corresponding to s and a path exists from s to u's parent.

Assume the assertion holds for all tree levels up to level k, and consider node (switch) u at level $k+1$. According to the symbol function (Table 4.1), u generates $\sigma = \texttt{s}$ only if it receives either a) s from the left child and n from the right child, or b) n from the left child and s from the right child, or c) s from the left child and b from the right child. For all these three cases, since u receives an s from one of its children, and since Assertion 1 is valid for u's children, there is an unmatched source in the subtree rooted at one of u's children and a path exists from that unmatched source to u.

For cases a) and b), the unmatched source s corresponding to s is not matched at u. Additionally, u establishes a path from its left child in case a) and right child in case b) to its parent (see Tables 4.2). For case c), the subtree rooted at u's left child has an unmatched source and the subtree rooted at u's right child has an unmatched source and an unmatched destination. Since, we are considering a right-oriented, well-nested, width-1 communication set, the unmatched source in the subtree rooted at u's left child must match the unmatched destination in the subtree rooted at u's right child. This leaves an unmatched source in the subtree rooted at u's right child. As shown in Table 4.2, u establishes a path connecting its right child to its parent. So in each of these three cases, there is an unmatched source in $\mathcal{T}_u$, and u establishes a path connecting the corresponding child to its parent, thereby establishing a path from the unmatched source to u's parent. This proves Assertion 1.

Similar arguments are valid for Assertions 2, 3, and 4 as well, thereby proving them.

We also prove the correctness of Assertion 5 by induction on the level of switches. For the base case, if u is at level $\ell = 0$, there cannot be any matched source-destination pair, which proves the assertion.

Assume the assertion holds for all tree levels up to level k, and consider node (switch) u at level $k+1$. Since the assertion holds for all nodes up to level k, the algorithm correctly creates all paths

corresponding to all matched source-destination pairs in subtrees rooted at each of u's children. This leaves us to prove that the paths corresponding to all the source-destination pairs that match at u are correctly created.

The correctness of Assertions $1-4$ proves that the algorithm correctly creates a path to u from the source and the destination nodes of a source-destination pair that matches at u. Switch u receives an `s` or a `b` from its left child and a `d` or a `b` from its right child corresponding to the matched source-destination pair. As shown in Table 4.2, in all of these cases u establishes a path connecting its left child to its right child. This means switch u indeed creates a path for any source-destination pair that matches at u, thus proving the assertion.

Each of the four unspecified entries of Tables 4.1 will lead to a width-2 communication set. Hence, they represent impossible situations, thus proving Assertion 6. □

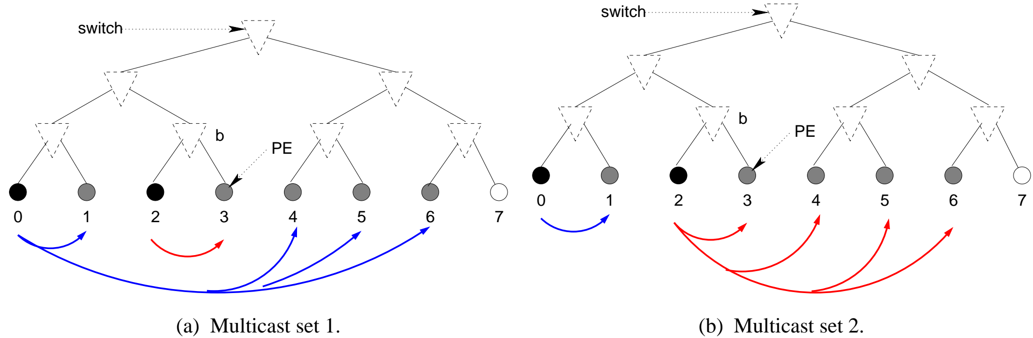
The proof of Assertion 5 of Lemma 4.3.3 establishes the following result.

Theorem 4.3.4. *The CST can be configured in one pass to perform all communications of any oriented, well-nested, width-*1 *communication set.* □

4.3.3 Width-1 Multicast Sets

In this section we address the more general situation of width-1, oriented, well-nested, multicast sets. Given only the flags indicating source, destination, or neither, a destination may not uniquely match a source and vice versa. For example, Figures 4.10(a) and 4.10(b) show two different width-1 multicast sets, each with two multicasts shown in red and blue, that have the same source-destination pattern. Hence, flags indicating only source, destination or neither are not sufficient for matching. Therefore, we assume that each multicast (s, D) has a unique ID associated with it that is known to s and all members of D. We do not assume that an ID encodes the destination set or the identity of the rightmost destination.

Even the assumption of known unique IDs is not sufficient, however, to configure the CST with a one-pass algorithm.



(a) Multicast set 1.

(b) Multicast set 2.

FIGURE 4.10. Two different multicast sets with same source-destination pattern.

Lemma 4.3.5. *No one-pass algorithm exists to configure a CST for width-*1 *multicast sets in which PEs hold communication IDs and are flagged only as source, destination, or neither.*

Proof. Again consider the example of Figures 4.10(a) and 4.10(b). Switch b receives symbols containing a source ID from PE 2 and a destination ID from PE 3 originating from members of the multicast shown in red in both the figures. In Figure 4.10(a) all connections of the multicast shown in red have been established and b does not need to forward any information about that multicast to the root. In Figure 4.10(b), b receives the same information, however, this time it needs to propagate information about the multicast shown in red to the root. Since b receives the same information in both the cases, it cannot distinguish the two situations under any one-pass algorithm with the given information. □

The situation changes, however, if the CST can identify for each multicast when all its destinations have been matched. For a right-oriented multicast, flagging the rightmost destination suffices. For simplicity, we assume the multicast set to be right oriented. The end of this section handles the general case. Now we present the CST configuration algorithm for right-oriented, width-1, multicast communication sets. For brevity, henceforth we call this algorithm width-1 multicast CST configuration algorithm.

Width-1 Multicast CST Configuration Algorithm

- Symbol set: Let $U = \{\mathtt{s}, \mathtt{d}, \mathtt{r}, -\}$ where $\mathtt{s}$, $\mathtt{d}$, and $\mathtt{r}$ denote source, non-rightmost destination, and rightmost destination. The character $-$ denotes a don't care entry. Let I be the set of all possible IDs of a multicast. Define $J = I \cup \{-\}$.

TABLE 4.3. The function f_s for width-1 multicast CST configuration algorithm.

f_s	$\mathtt{s},m_3,-,-$	$-,-,\mathtt{d},m_4$	$-,-,-,-$	$-,-,\mathtt{r},m_4$	$\mathtt{s},m_3,\mathtt{d},m_4$	$\mathtt{s},m_3,\mathtt{r},m_4$
$\mathtt{s},m_1,-,-$		if $m_1=m_4$ $\mathtt{s},m_1,-,-$ else $\mathtt{s},m_1,\mathtt{d},m_4$	$\mathtt{s},m_1,-,-$	if $m_1=m_4$ $-,-,-,-$ else $\mathtt{s},m_1,\mathtt{r},m_4$		$\mathtt{s},m_3,-,-$
$-,-,\mathtt{d},m_2$	$\mathtt{s},m_3,\mathtt{d},m_2$	$-,-,\mathtt{d},m_4$	$-,-,\mathtt{d},m_2$	$-,-,\mathtt{r},m_4$	$\mathtt{s},m_3,\mathtt{d},m_4$	$\mathtt{s},m_3,\mathtt{r},m_4$
$-,-,-,-$	$\mathtt{s},m_3,-,-$	$-,-,\mathtt{d},m_4$	$-,-,-,-$	$-,-,\mathtt{r},m_4$	$\mathtt{s},m_3,\mathtt{d},m_4$	$\mathtt{s},m_3,\mathtt{d},m_4$
$-,-,\mathtt{r},m_2$	$\mathtt{s},m_3,\mathtt{r},m_2$		$-,-,\mathtt{r},m_2$			
$\mathtt{s},m_1,\mathtt{d},m_2$		if $m_1=m_4$ $\mathtt{s},m_1,\mathtt{d},m_2$ else $\mathtt{s},m_1,\mathtt{d},m_4$	$\mathtt{s},m_1,\mathtt{d},m_2$	if $m_1=m_4$ $-,-,\mathtt{d},m_2$ else $\mathtt{s},m_1,\mathtt{r},m_4$		$\mathtt{s},m_3,\mathtt{d},m_2$
$\mathtt{s},m_1,\mathtt{r},m_2$		$\mathtt{s},m_1,\mathtt{r},m_2$	$\mathtt{s},m_1,\mathtt{r},m_2$	$-,-,\mathtt{r},m_2$		$\mathtt{s},m_3,\mathtt{r},m_2$

The symbol set is $\mathcal{S} = U \times J \times U \times J$. A typical member of $\mathcal{S}$ has the form $(\alpha,\beta,\gamma,\delta)$, where $\alpha,\gamma \in U$ and $\beta,\delta \in J$. The intuition behind this symbol set is as follows. Each node may have to send information about (at most) two multicasts; one with source in $\mathcal{T}_u$ and the other with destination(s) in $\mathcal{T}_u$. (Recall that $\mathcal{T}_u$ is the subtree rooted at node u of the CST.) The information of each multicast consists of an `s`, `d`, or `r` and its ID. Therefore, (α,β) and (γ,δ) represent the two multicasts. The don't-care symbol accounts for cases with fewer than two multicasts.

- Initial symbol assignment: A leaf sends symbol `s` (resp., `d` or `r`) to its parent iff it is a source (resp., non-rightmost destination or rightmost destination) along with its multicast ID.

- Symbol function: Table 4.3 gives f_s.

- Configuration function: Table 4.4 gives f_c.

We now address the correctness of the multicast algorithm. Consider any multicast (s,D). Let $r \in D$ be the rightmost destination. A subtree $\mathcal{T}_u$ has a *matched* source s iff $s,r \in \mathcal{T}_u$. Subtree $\mathcal{T}_u$ has a matched destination $d \in D$ iff $d \in \mathcal{T}_u$ and either $s \in \mathcal{T}_u$ or $\mathcal{T}_u$ has a destination $d' \in D$ such that d' is to the right of d. A multicast (s,D) is *completed* in subtree $\mathcal{T}_u$ if s and each $d \in D$ are leaves of $\mathcal{T}_u$.

TABLE 4.4. The function f_c for width-1 multicast CST configuration algorithm.

f_c	$\mathtt{s},m_3,-,-$	$-,-,\mathtt{d},m_4$	$-,-,-,-$	$-,-,\mathtt{r},m_4$	$\mathtt{s},m_3,\mathtt{d},m_4$	$\mathtt{s},m_3,\mathtt{r},m_4$
$\mathtt{s},m_1,-,-$						
$-,-,\mathtt{d},m_2$						
$-,-,-,-$						
$-,-,\mathtt{r},m_2$						
$\mathtt{s},m_1,\mathtt{d},m_2$						
$\mathtt{s},m_1,\mathtt{r},m_2$						

Notice that we have defined matched source and destinations in terms of the set D. The actual algorithm uses multicast IDs to ascertain a match and does not require determination of D. Arguments similar to the ones presented in proof of Lemma 4.3.3 hold for the following lemma as well.

Lemma 4.3.6. *Let u be any node at level ℓ, where $0 \leq \ell \leq p$, of the CST. Under the width-1 multicast CST configuration algorithm, let u generate symbol $(\alpha, \beta, \gamma, \delta) \in \mathcal{S}$ to send to its parent, if any. The following assertions hold.*

1. *If $\alpha = \mathtt{s}$, then $\mathcal{T}_u$ has an unmatched source s of multicast (s,D). The algorithm establishes paths from s to the parent of u and all (matched) destinations of D that are in $\mathcal{T}_u$.*

2. *If $\gamma = \mathtt{d}$ or $\mathtt{r}$, then $\mathcal{T}_u$ contains an unmatched destination of multicast (s,D) and the algorithm establishes a path from the parent of u to this destination.*

3. *If $\alpha = -$, then $\mathcal{T}_u$ contains no unmatched source.*

4. *If $\gamma = -$, then $\mathcal{T}_u$ contains no unmatched destination.*

5. *For a multicast (s,D) that is completed within $\mathcal{T}_u$, the algorithm establishes a path within $\mathcal{T}_u$ from s to all elements of D.*

6. *All unspecified entries of Tables* 4.3 *and* 4.4 *represent impossible situations.* □

So far, we have assumed a right-oriented multicast set. Clearly, this approach also works for a left-oriented set. If we drop the assumption of orientedness, then both the leftmost and rightmost (extreme) destinations need to be flagged. The width-1 multicast CST configuration algorithm will still work with minor modifications to Tables 4.3 and 4.4 and the definition of a match.

Theorem 4.3.7. *The CST can be configured in one pass to perform all communications of any width-1 multicast set.* □

4.4 Well-Nested, Width-w Communication Sets

In this section we apply the general CST configuration algorithm of Section 4.2 to configuring the CST for a well-nested, right-oriented, width-w communication set. Well-nested communication sets are width partitionable [21], so an optimal algorithm will take exactly w rounds to route all communications in the communication set.

Before providing the details, we define the *nesting depth* of a communication belonging to a well-nested communication set. A right-oriented communication (a,b) *covers* communication (a',b') iff the PE indices satisfy $a < a' < b' < b$; for example, communication $(2,7)$ covers communication $(4,6)$ in Figure 4.2. The nesting depth of a communication c is the number of communications that cover c. For example, the nesting depth of communication $(4,6)$ in Figure 4.2 is 1.

For a source at depth d, the next element to its right at depth d is its matching destination. It is easy to verify that nesting depths of well-nested communications possess the following properties.

1. No two communications with the same depth share a common CST edge in the same direction.

2. If a communication c_1 covers another communication c_2 and no other communication c_3 covers c_2 but not c_1, then the depth of c_1 is one less than the depth of c_2.

4.4.1 Algorithm Adaptation

We now provide details of the general CST configuration algorithm adopted to a well-nested communication set. Sections 4.4.2 and 4.4.3 prove this adaptation to be correct and optimal, respectively.

Phase 1: The PEs calculate the nesting depth of their corresponding communications and assign this value as the ID of each communication. To calculate the nesting depth of a communication, corresponding PEs compute prefix sums where each source contributes a 1 and each destination contributes a -1. Each source PE then subtracts 1 from its prefix sum. Figure 4.11 shows the computation of IDs for an example well-nested communication set. Dharmasena and Vaidyanathan [18] gave a prefix sums algorithm that traverses the CST from the leaves to the root and then back to the leaves.

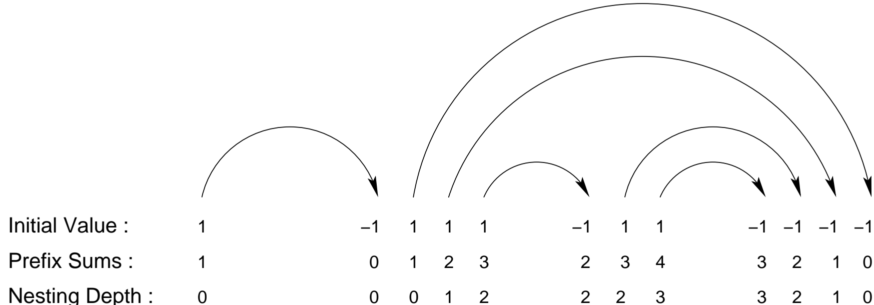


FIGURE 4.11. Computation of IDs for a well-nested communication set.

Phase 2: This phase is similar to that given in Section 4.2. The only difference is that instead of switch u sending a set S of sources (set D of destinations) to its parent, u sends the indices of the lowest valued and the highest valued source (destination).

Phases 3, 4, and 5: Same as the general CST configuration algorithm.

Each of Phases 1, 2, 3, and 5 requires at most two passes of the CST (leaves to root or root to leaves) and runs in $O(\log n)$ time. Each switch spends a constant amount of time for computing in each phase, so the $O(\log n)$ time complexity arises due to the $\log n$ height of an n-processor CST. Phase 4 takes $O(1)$ time and we show that there are w rounds involving all phases. Consequently, the algorithm to schedule and route a width-w, well-nested communication set runs in $O(w \log n)$ time.

4.4.2 Correctness of Phases 1 and 2

Lemma 4.4.1. *For each communication, the ID computed during Phase 1 corresponds to the nesting depth of that communication.*

Proof. A right-oriented, well-nested communication set corresponds to a balanced parenthetical expression where each source (resp., destination) maps to a left (resp., right) parenthesis.

The prefix sum for each source indicates the number of sources before it (including itself) whose destinations have not been encountered. This sum is one more than the source's depth as the communications with unmatched sources cover this communication. The prefix sum for each destination indicates the number of sources before it whose destinations have not been encountered. Consequently, the prefix sum at a source is one greater that the prefix sum at its matching destination. Subtracting one from each source prefix sum ensures that matching source-destination pairs have the same ID which will also correspond to the nesting depth. □

Lemma 4.4.2 establishes that the IDs of two communications can be the same, yet all the communications can be uniquely identified, as unmatched IDs are unique in any switch's subtree. Moreover the IDs satisfy Properties 1 and 2 for the nesting depths of a well-nested communication set.

Lemma 4.4.2. *In the subtree of any switch, no two unmatched sources can have the same ID and no two unmatched destinations can have the same ID.*

Proof. Let us assume that there are two unmatched sources with the same ID m in the subtree of a switch c. Let us call the left (resp., right) one of them s_L (resp., s_R). Clearly, c receives information from both of them and the corresponding matching destinations are outside the subtree rooted at c.

Since the communication set is well-nested, the destination corresponding to s_L will be somewhere to the right of the destination corresponding to s_R. Hence, the communication with source s_R is completely nested within the communication with source s_L. So (by Property 2) the ID of s_L computed during Phase 1 will be less than the ID of s_R contradicting our assumption that s_L and s_R have the same ID.

With similar arguments, one can also show that no two unmatched destinations in any switch's subtree can have the same ID. □

In Phase 2 of the algorithm, instead of switch u sending a set S (resp., D) of sources (resp., destinations) to its parent, u sends only the smallest and the largest values of S (resp., D). Lemma 4.4.3 proves that the elements of S (resp., D) will consist of contiguous IDs, so the closed interval formed by the smallest and the largest values identifies S (resp., D).

Lemma 4.4.3. *In Phase 2 of the algorithm, if a switch c sends $S = [p,q]$ (resp., $D = [p,q]$) to its parent, then there are $q-p+1$ unmatched sources (resp., destinations) with IDs $p, p+1, \cdots, q$ that form a continuous interval in the subtree rooted at c.*

Proof. We prove the correctness of the lemma by induction on the level of switches (with PEs at level 0 and the root at level $\log n$ for an n-leaf CST). We present the argument for the sources; the argument for the destinations is analogous.

As a base case, at level 0 (PEs), if a PE holds a source p, then it sends $S = [p,p]$ to its parent, identifying this unmatched source. If a processor holds no source, then it sends $S = null$ to its parent indicating no unmatched sources.

Assume the lemma holds for all tree levels up to level k, and consider switch c at level $k+1$. In Phase 2, suppose that c received $S_L = [l_1, l_2]$ and $S_R = [r_1, r_2]$ from its left and right children, respectively, and sent $S = [p,q]$ to its parent. For a right-oriented communication set, any pair

matching at c must have its source in the left subtree and destination in the right subtree. For a well-nested communication set, the matching pairs must be the innermost on the nesting, with largest IDs (Property 2), and by the inductive hypothesis, the sequence of their IDs must be contiguous. Let $[m, l_2]$ denote the interval of matching source IDs in the left subtree. Unmatched intervals are $S_L - D_R = [l_1, m-1]$ and $S_R = [r_1, r_2]$.

All sources between source $m-1$ and source r_1 match, contributing 0 to the prefix sum. Since IDs follow from prefix sums, $r_1 = (m-1)+1 = m$. Thus, $S = (S_L - D_R) \cup S_R = [l_1, m-1] \cup [m, r_2] = [l_1, r_2]$ is itself a continuous interval, proving the lemma. □

The fact that an interval tersely represents the sets S and D makes the algorithm very efficient. Moreover the operations needed to compute sets S and D in Phase 2 reduce to performing a small number of comparisons of $O(\log n)$ bit IDs.

4.4.3 Proof of Optimality

In this section we show that our algorithm is indeed optimal; that is, it routes all communications of a well-nested, width-w communication set in w rounds. A width-w communication set must have a subset of w sources (or w destinations) that use a common directed edge of the tree. Such a set is called a *maximum incompatible*. To prove a schedule optimal, it suffices to show that it routes one communication in each round from each maximum incompatible [21].

Consider any maximum incompatible I (Figure 4.12). Let the edge(s) used by all w communications be upward, so I contains source PEs. (An analogous argument holds for a destination incompatible.) Let switch u be the lowest common ancestor of the sources in I. Let v be the lowest level switch where at least one among these w sources matches. For right-oriented communications, the w sources must belong to the left subtree of v. Since the source incompatible is maximum, no other source can join these sources at any switch between u and v.

In Phase 3 of the algorithm, v will receive symbols a and b from its parent, where a is a source ID or is *null* and b is a destination ID or is *null*. If both a and b are *null*, then v will configure a

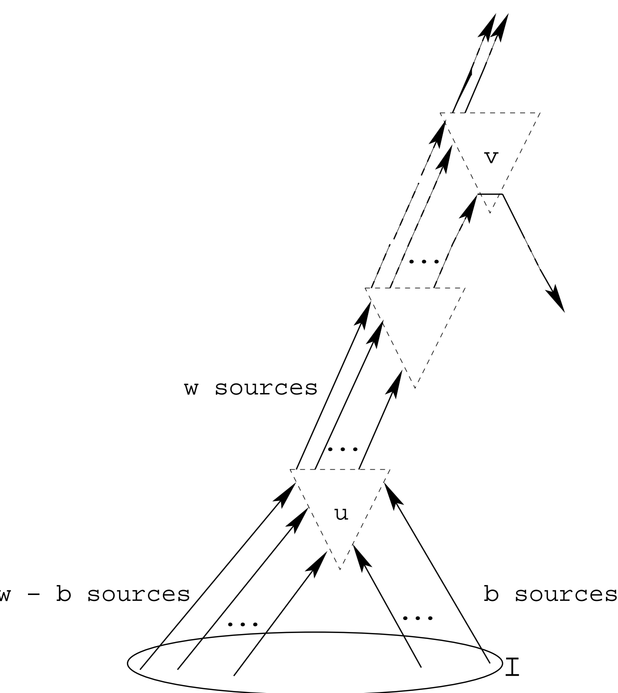


FIGURE 4.12. Part of a CST showing a maximum source incompatible for a width-w communication set.

communication that matches at v. By definition of v, at least one such communication exists and all matching communications have their sources in I.

If v receives an ID in Phase 3, then the following three cases exist. Assume that, in Phase 2, v received source and destination sets S_L, D_L from its left child and S_R, D_R from its right child.

1. If $a \in S_L$, then $a \in I$. Hence, one source from the maximum source incompatible is routed.

2. The case $b \in D_R$ is not possible in a width-w, oriented, well-nested set.

3. If $a \notin S_L$ and $(b \in D_L$ or $a \in S_R)$, then the communications corresponding to a and b cannot be incompatible with communications matching at v. By the last **if** statement in the pseudocode of Figure 4.8, v will configure a communication that matches at v and whose source is from I.

Hence, Phase 3 will route one source of each maximum source incompatible and subsequently the width will reduce by 1. This proves the algorithm to be optimal.

Theorem 4.4.4. *Every oriented, well-nested communication set can be routed optimally on the CST. Moreover, each switch step uses a constant-time computation and communicates a constant number of words with its neighbors.* □

Section 4.3 presented a simplified algorithm that suffices for a general width-1 communication set. For a well-nested, width-1 communication set, an even more simplified algorithm with Phase 1 removed suffices.

Phase 1 of the algorithm assigns IDs to each communication to uniquely identify them. In the width-1 case, the path between the source (resp., destination) of any communication and the switch where the communication matches is not shared by any other source (resp., destination). This implicit property of the width-1 case is sufficient to identify any source (resp., destination) within the subtree rooted at the switch where the communication corresponding to that source (resp., destination) matches. Hence, Phase 1 becomes redundant.

4.5 Summary

In this chapter we described our research in scheduling and configuration of communications on a CST. We presented various properties of communication sets like width and oriented-ness, and looked at both point-to-point and multicast communication sets.

We presented a CST configuration algorithm for a width-w communication set. We also presented special adaptations of that algorithm for width-1 point-to-point and multicast communication sets. Lastly, we also presented algorithm adaptations for width-w well-nested communication sets for which the algorithm is provably optimal.

Chapter 5

Routing Algorithm for an R-Mesh Based Fat-Tree Switch

5.1 Introduction

We presented a brief introduction to fat-trees [46, 47] in Chapter 1. A fat-tree is an extension of the simple tree topology where the bandwidth between successive levels increases exponentially as shown in Figure 5.1. Many of today's high-performance clusters, especially ones using the Infiniband, Myrinet, or Quadrics interconnection families employ a fat-tree structure [32, 33]. These three interconnection families taken together account for roughly one third of the current top 500 supercomputers [37]. Research related to interconnections in NoCs also widely use fat-tree or fat-tree type structures like the H-tree [9, 10, 54]. Fat-trees also have application in interconnection networks for high-performance disk storage architectures [78, 79].

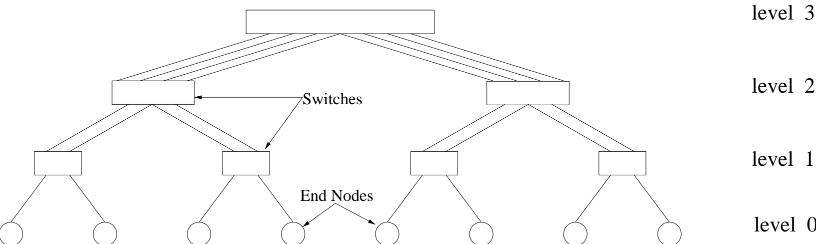


FIGURE 5.1. An 8-leaf fat-tree; multiple edges between two switches denote higher bandwidth.

In a tree a unique path exists between a source-destination pair. In a fat-tree, on the other hand, there are multiple edges between any two switches. Hence, there is a scope for deriving benefit by properly choosing (for a communication) one of the multiple links that connect two switches. The precise choice of the link depends on the link loads, switch bisection bandwidth and other communications attempting to use the switch at the same time. A general level$-(k+1)$ fat-tree switch (see Figure 5.1) has 2^{k+1} bidirectional ports connecting it to its parent. Additionally, it has 2^k ports to each child. More precisely, consider the level $k+1$ switch shown in Figure 5.2, where $0 \leq k \leq n$, with the root at level n. In routing from a source to a destination, the global (high-level) path is fixed. For example, we may already know that the path is from (say) left child to the parent

of the switch in Figure 5.2. However, the communication may use any of the $2^{\ell-1}$ input links from the left child and exit to any of the 2^{ℓ} output links to the parent.

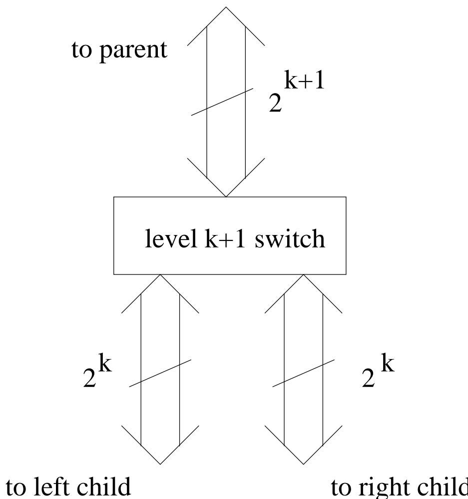


FIGURE 5.2. A level $k+1$ switch.

Routing, performance analysis, and implementation of fat-tree-based interconnection networks are well studied problems [34, 61, 72]. Lin *et al.* [51] used the *k*-ary *n*-tree structure to implement a fat-tree using fixed-size switches in InfiniBand networks. They also proposed a routing scheme based on assigning multiple LIDs (local identifiers, which are identifiers used to address all the end systems in an InfiniBand network) to all the end nodes to utilize multiple paths that exist between any source-destination pair in a fat-tree. Gomez *et al.* [32] designed a deterministic routing algorithm for a fat-tree (implemented as a *k*-ary *n*-tree). They used the idea of using a pre-defined path for the ascending part of a route for each source-destination pair to achieve the deterministic routing. Their algorithm balanced the overall network load. They also presented simulation results to show that the performance of the deterministic algorithm is comparable to or better than adaptive algorithms for similar network traffic. Gomez *et al.* [33] used their earlier algorithm developed in [32] to design a simplified switch architecture for a fat-tree which effectively almost halved the switch hardware complexity. Ding *et al.* [19] proposed a level-wise scheduling algorithm for a fat-tree interconnection network. This algorithm used global information to select upward routing paths instead of using just local information available at each switch. This reduced the number of conflicts between communications and hence improved the schedulability ratio.

With fat-tree based interconnects gaining popularity, researchers are also investigating various related research problems. Sem-Jacobsen *et al.* [70, 71] looked at dynamic fault tolerance and its effect on quality of service on fat-trees. Alonso *et al.* [1, 2] researched power related issues in fat-trees.

Usually, fat-tree switch routing algorithms (especially in the fat-tree switches of supercomputer interconnects), employ a table lookup. A table lookup is a centralized approach, and hence it suffers from the disadvantages that a centralized approach usually has in terms of scalability and performance. Hence, a distributed approach to connect ports within a switch merits investigation.

In our research related to a fat-tree switch, we design an R-Mesh[1] based algorithm to generate the intra-switch connections while achieving a certain degree of load balancing in a greedy manner. This algorithm is a work in progress and needs additional results (modeling and simulations) to evaluate its performance. In Chapter 3, we used the R-Mesh as the control plane to generate schedules for the crossbar. The crossbar established the corresponding paths by setting appropriate crosspoints. Here too we are using the R-Mesh as the control plane for a fat-tree switch. The data plane is assumed to be any mesh structure that accommodates the paths created by the R-Mesh's buses.

The R-Mesh itself creates the data paths by creating buses through it. Our research in this area is still preliminary, hence we present only the basic idea and the algorithm in the next section.

5.2 Routing Algorithm for a Fat-Tree Switch Implemented as an R-Mesh

In this section we outline the algorithm that we have designed to create configurations of an R-Mesh, each of which will establish a set of buses connecting source and destination ports of a fat-tree switch. Note that our main motivation for this algorithm is that we want to embed an R-Mesh inside a fat-tree switch and then use the reconfigurability of the R-Mesh to dynamically create paths from source ports to destination ports inside the switch.

[1] For detailed background on R-Mesh refer to Section 3.3.

We assume that the switch has 2^k ports connected to each of the left and right children and 2^{k+1} ports connected to its parent. We further assume that the R-Mesh embedded inside the switch has a size of $\ell \times 2^{k+1}$ where each PE in the 0th (top) row (or parent side) is connected to a port that is connected to the parent and each PE in the $(\ell-1)^{th}$ row (child side) is connected to a port that is connected to a left or a right child. We do not impose any restriction on the value of ℓ, and assume $1 \leq \ell \leq n$.

The algorithm that we present here creates buses for communications that are going from the left or right child of the switch to the parent; the other cases (from parent to child and between children) are analogous. In this context, the parent (resp., child) side of the R-Mesh can also be called the input (resp., output) side. Note that a bus corresponding to a child-to-parent communication can connect a specified input port to any output port on the parent side. Simple variations of the same idea can take care of the four other possible source-destination pairings.

We flag input ports as *non-empty* or *empty* (depending on whether they have any packets waiting to be sent at the input buffers). We flag output ports as *full* or *ready* (depending on whether they have room to accept a packet). The state of an output port α may be determined by an underlying flow-control mechanism that reflects possibly the state of the buffers at the input port of the neighboring switch to which the output port α is directly connected. We do not discuss this flow control. We also assume that whenever an input or output port dispatches a packet to an output port (within the switch) or an input port (in the neighboring switch) respectively, the corresponding states of the ports are automatically adjusted. Our algorithm runs continuously in rounds (as in the scheduling algorithm of Section 2.2). In each round it creates paths (as many as possible) from non-empty input ports to ready output ports. If it is not possible to connect some non-empty input port to an output port in the current round, then it is considered in a subsequent round. The algorithm has three stages:

(i) Construct straight buses.

(ii) Construct buses to the right.

(iii) Construct buses to the left.

The last two stages are analogous and so we only describe Stages (i) and (ii).

Consider the level $k+1$ switch in Figure 5.2 that uses a $\ell \times 2^{k+1}$ R-Mesh for some $1 \leq \ell \leq 2^{k+1}$. Figure 5.3 shows the internal structure of such a switch. The output ports (only those to the parents

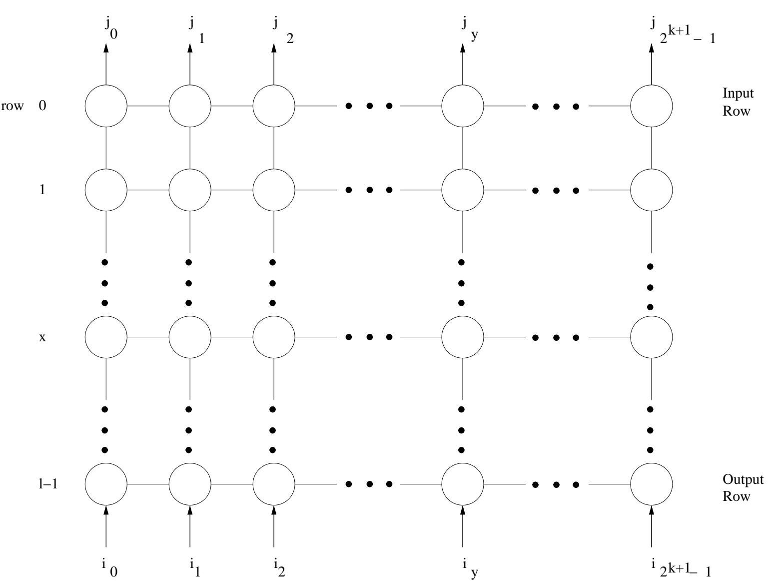


FIGURE 5.3. The R-Mesh inside the fat-tree switch.

are shown) are labeled j_y $(0 \leq y < 2^{k+1})$. The input ports (only those from the children are shown) are labeled i_y $(0 \leq y < 2^{k+1})$. At this point we do not specify which input ports connect to a left child or right child. However, the set $I = \{i_y : 0 \leq y \leq 2^{k+1}\}$ is partitioned into 2^k-element sets I_R and I_L that connect to the switches at the right and left children respectively. Denote the PE in row x and column y by PE(x, y) where $0 \leq x \leq \ell$ and $0 \leq y \leq 2^{k+1}$. Clearly, PE$(0, y)$ and PE$(\ell - 1, y)$ access output and input ports j_y and i_y respectively.

We now describe the algorithm; examples in Section 5.2.1 illustrates how it works.

Initially, the algorithm marks all non-empty input ports as S (or source) ports and all ready output ports as D (or destination) ports.

Stage (i)– Constructing straight buses:

For each column y whose ports i_y and j_y are S and D ports, respectively, a vertical bus through the column suffices to connect these ports. All PEs in such columns are configured to connect their south port to their north port.

Beyond this point these ports are no longer considered in Stages (ii) and (iii).

Stage $(ii)-$ Constructing buses to the right:

In this stage we connect input ports to output ports in columns to the right. This stage has three steps described below.

Step 1 $-$ Compute Prefix Sums:

For each source port i_y, set a source weight $W_s(y)$ to 1. For all other non-source ports $i_{y'}$, set $W_s(y')$ to 0. Similarly use $W_d(y)$ to flag all destination port j_y.

Compute the "source prefix sums" $P_s(y)$ of the source weights. That is, for each $0 \le y \le 2^{k+1}$,

$$P_s(y) = \sum_{u=0}^{y} W_s(u).$$

Similarly, compute the destination prefix sums

$$P_d(y) = \sum_{u=0}^{y} W_d(u).$$

Step 2 $-$ For each column y with a destination port j_y, determine the row $r(y)$ through which some source port could connect to j_y.

$$r(y) = (P_d(y) - P_s(y) - 1)(mod\ \ell) = (\ell + P_d(y) - P_s(y) - 1)(mod\ \ell)$$

Recall that we are using an $\ell \times 2^{k+1}$ R-Mesh.

Step 3 $-$ Set up buses to the right:

This step establishes the connections between an input port and an output port to its right, whenever possible.

Each PE of the R-Mesh configures its ports as described in Table 5.1.

Stage $(iii)-$ Constructing buses to the left:

All source-destination pairs connected in Stages (i) and (ii) are removed from consideration at this stage. However, their connections remain.

TABLE 5.1. PE configurations for creating buses to the right.

PE	Configurations
PE(x,y) is in source column y	
PE(x,y) is in destination column y	
All other PEs	

Stage (iii) works just as Stage (ii), except in the opposite direction (right to left). If this stage calls upon a PE to configure itself in a manner that conflicts with its configuration created in an earlier stage, then the algorithm defers to the earlier configuration. Figure 5.4 shows example of allowed and conflicting configurations.

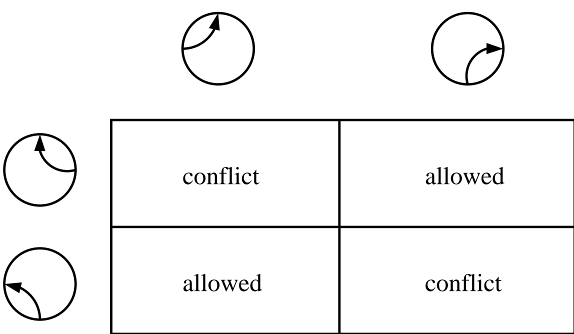


FIGURE 5.4. Examples of allowed and conflicting configurations.

5.2.1 Examples Illustrating the Algorithm

We illustrate the algorithm with two examples each on a 3×10 R-Mesh. We use 10 columns only for the purpose of illustrations. The fat-tree switch has 2^{k+1} columns for some $k \geq 0$.

Example 5.2.1: Suppose we have the source and destination ports shown in as Table 5.2. Stage

TABLE 5.2. Position of sources and destinations on the R-Mesh.

Column y	0	1	2	3	4	5	6	7	8	9
Source/Destination	S	S	S	D	D	D	D,S	D	S	S

(i) connects the ports i_6 and j_6. After Stage (i), these ports are no longer considered in subsequent stages. Figure 5.5 illustrates Stage(ii). For clarity, only those connections between PEs that are part of buses are shown.

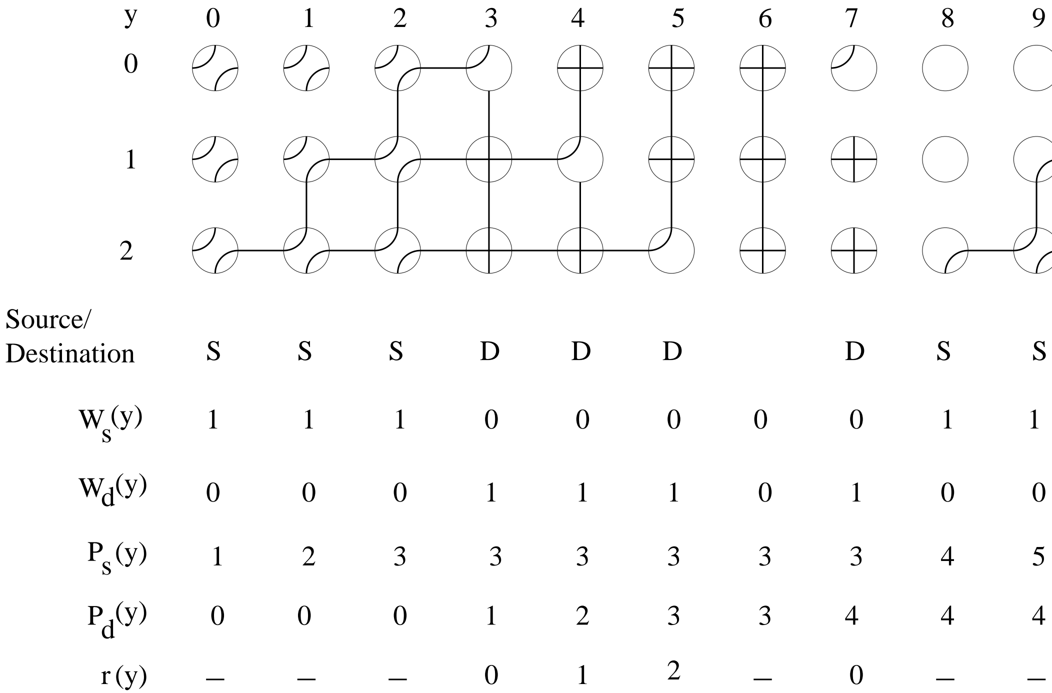


FIGURE 5.5. Example illustrating Stage (ii).

Observe that even though column 7 has a ready destination port, the sources in column 8 and 9 do not find it as they are looking to the right. In Stage (iii) they will seek to create buses to the left shown in Figure 5.6.

Figure 5.7 shows the final bus configuration of the R-Mesh. This results in the following pairings: (i_0, j_3), (i_1, j_4), (i_2, j_5), (i_6, j_6), and (i_9, j_7). Notice that while there is a bus emanating from i_8, it does not head to any output port. PEs at output ports write a symbol on their buses and PEs at input ports try to read this symbol. Those that do not read the symbol (input port i_8 in this example) are reconsidered in the next round.

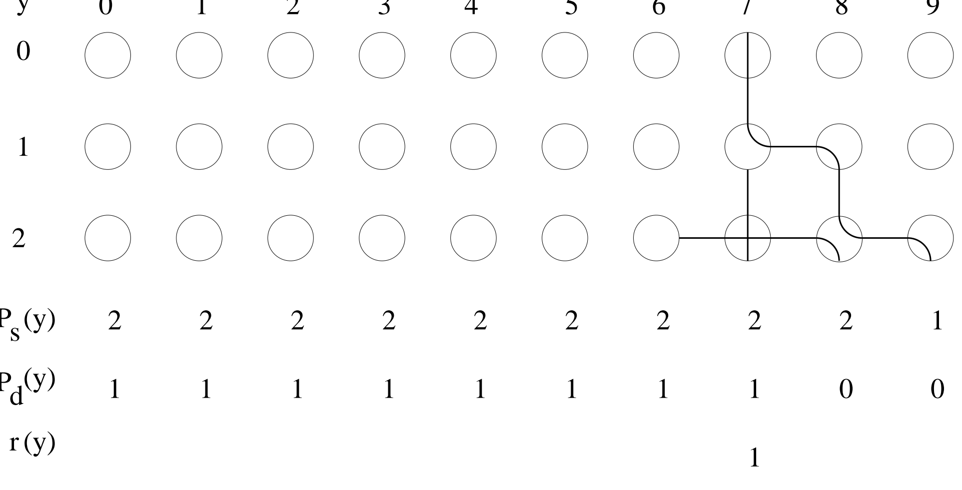


FIGURE 5.6. Example illustrating Stage (*iii*).

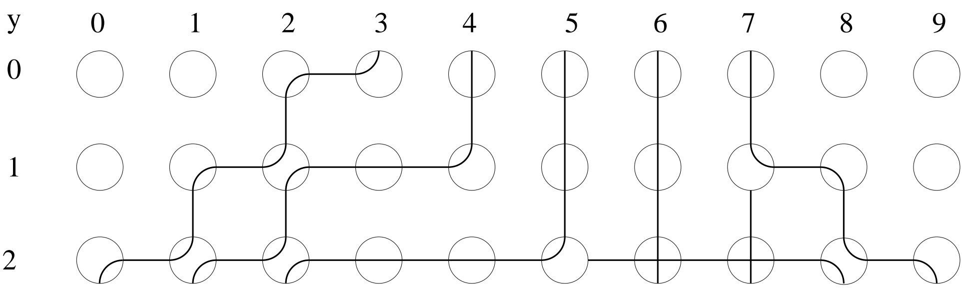


FIGURE 5.7. Final bus configurations.

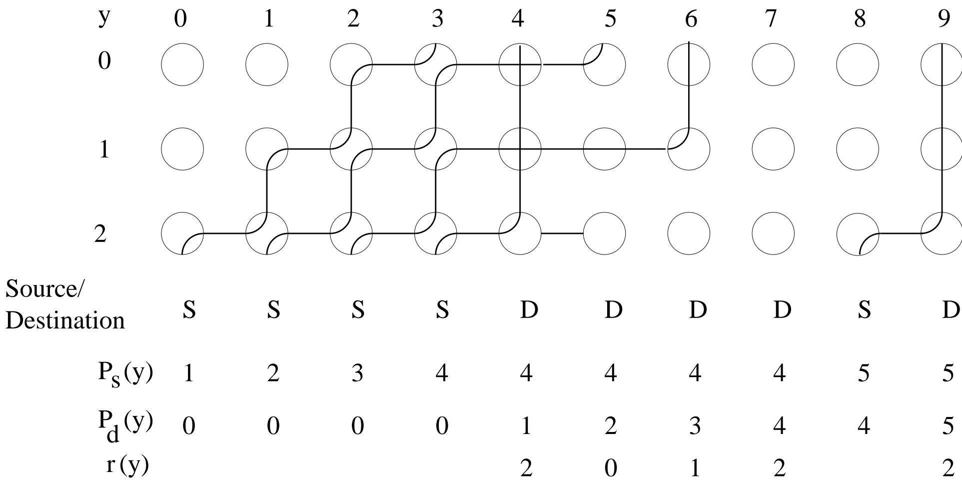


FIGURE 5.8. Bus configurations of example 2.

Example 5.2.2: Here we omit columns which has a non-empty source and a ready destination port. Thus we omit Stage (*i*) from the following illustration. Figure 5.8 shows Stage (*ii*). Although i_0 has a bus emanating from it, it does not lead to a destination port. It leads to j_3 which is full in our

illustration. Also, i_0 cannot make any connections in Stage (iii). Thus, i_0 will have to wait till the next round.

5.3 Summary

In this chapter we introduced the fat-tree switch. We discussed some of the related research in this field that motivated us to look at embedding an R-Mesh in a fat-tree switch for generating connections. We presented the broad algorithm that creates buses connecting source-destination pair while taking into account associated loads in a coarse fashion.

Chapter 6
Summary of Results and Open Problems

In this dissertation we presented our research on interconnection-network switch scheduling and configuration. We studied the following interconnection-network switches:

1. Crossbar-based input-queued switches used in many of today's high-performance routers;

2. The circuit-switched tree (CST), a tree interconnect with applications in reconfigurable structures and NoCs; and

3. Fat-tree switches used extensively in high-performance computing clusters.

In this chapter we summarize our main results and identify directions for future research.

6.1 Crossbar-Based Input-Queued Switches

A crossbar-based input-queued switch with VOQs is one of the most popular switch architectures. Such switches have very high throughput, do not suffer from the head-of-line blocking problem, are non-blocking, and have a simple internal data-fabric structure. These advantages resulted in many switch and router manufacturers adopting this switch architecture in their production models. A recent result [59] showed that $O(\log n)$ packet delay is possible on an $n \times n$ input-queued crossbar with the use of a slotted, frame-based scheduling algorithm. This algorithm defines a slot as the time to send a packet and a frame as a collection of slots that uses a fixed (unupdated) traffic matrix for scheduling[1]. During each slot, the algorithm generates a schedule and transmits a packet between each scheduled input-output pair.

One of the assumptions made in the above algorithm is that the time to generate a schedule (round time) is no more than the time to transmit a packet. In practical systems this assumption does not hold as the time needed to send a packet is typically much smaller than the time needed

[1] The algorithm updates the traffic matrix at the beginning of each frame and accounts for packets arriving during a frame at the beginning of the next frame.

to generate a schedule. In Chapter 2, we proved that if the difference in packet transmission time and schedule generation time forces a crossbar scheduling algorithm to transmit multiple packets per schedule, then the delay is $\Omega(n)$. This proof also showed that transmitting a single packet per schedule is the only case that achieves a logarithmic delay. We also performed extensive simulations for uniform as well as bursty traffic to support our theoretical result in practical environments.

We further underscored the importance of a fast scheduling algorithm by showing that large schedule times result in the need for large buffers. This additional buffer requirement can negate any saving in computing hardware and power consumption that can be obtained from slowing down the scheduling algorithm.

Open Problems: Our work decoupled the schedule time and the packet transmission time of the crossbar-based scheduling algorithm of Neely *et al.* [59] and studied the effect of this on packet delay and VOQ requirements. Other results in this area use similar slotted scheduling scheme without treating schedule time (round) and transmission time (slot) differently [50, 66]. It is worth investigating whether our approach of decoupling round and slot times will add new insights to these results.

The simulation framework of Section 2.5 can facilitate studying other relationships such as frame-sizes (minimum number of rounds needed to schedule "most" packet arrivals from the previous frame) needed for a given *pps*. Although we know that this frame size will be linear for *pps* ≥ 2, our results clearly show that the constants for the linear growth depend on the value of *pps*.

We have presented simulation results that show trade-offs among delay, packet loss, VOQ requirement, and *pps*. Developing analytical relationships among these quantities is another open problem.

6.2 Fast Scheduling Algorithm on Mesh-of-Trees

The main result of Chapter 2 pointed to the fact that in order to achieve logarithmic delay in an input-queued crossbar, the schedule generation must be extremely fast and should be comparable to the time needed to transmit a packet. We designed a fast mesh-of-trees based scheduling algorithm in Chapter 3. This algorithm runs in polylog time ($O(\log^4 n \log\log n)$ for an $n \times n$ crossbar). In

designing this algorithm, we also constructed a polylog time maximal matching algorithm for an R-Mesh.

Open Problems: A sub-procedure of our R-Mesh-based maximal-size bipartite matching algorithm is a leader-election algorithm that executes in $O(\log n)$ time. In the main algorithm, we generate several very regular-shaped buses on which the leader election commences. An open problem is to investigate whether it is possible to utilize this regularity of the bus structure to construct a faster leader-election algorithm. This would significantly increase the speed of the entire scheduling algorithm. If leader election takes $O(t)$ time, then the scheduling algorithm runs in $O(t \log^3 n \log\log n)$ time. A constant time leader election would cut the time by a $\Theta(\log n)$ factor.

Fairness of a scheduling algorithm is an important issue in any switch. Our R-Mesh based matching algorithm does not explicitly address fairness. However, we expect simple modifications to suffice, such as running the same algorithm on an R-Mesh with a suitable permutation of row and column indices, to work well for this purpose. For this a reconfigurable-torus (which is not significantly different from the R-Mesh) can be used. One could permute the PE indices (without permuting port indices) by simply "declaring" different processors in the torus to be node $(0, 0)$ – the origin. Since a torus is vertex and edge symmetric, there exists an R-Mesh (subgraph of torus) consistent with these new origin.

Our algorithm includes efficient solutions to several graph problems like degree-halving, two-coloring, and leader election on a mesh-based structure in order to achieve maximal bipartite matching. All these methods could have other computational applications. For example, bipartite matching is used in protein structure matching, 3-D object recognition, and multi-objective optimization. It would be useful to extend our R-Mesh algorithm to other applications such as those identified above.

Current GPGPUs [60] have a large number of small processing units with a rich interconnect between them. Is it possible to port R-Mesh algorithms to exploit the parallelism inherent in GPGPUs?

The R-Mesh is very well suited to handle faults (by dynamically constructing buses to bypass faulty PEs and links)[2]. Can our algorithm exploit this ability of the R-Mesh to impart fault tolerance to the control and data plane of the switch. Addressing soft faults algorithmically is another possible direction.

The algorithm for an $n \times n$ switch runs on an $n \times n$ R-Mesh. In general, the R-Mesh may not scale well to run larger sized problems than the available hardware [77]. That is, if a $p \times p$ R-Mesh is used for an $n \times n$ switch (where $p < n$), then how efficient will our algorithm be?

6.3 Circuit-Switched Tree Switches

The CST is an important interconnect used to implement dynamically reconfigurable architectures. A CST-type structure can have applications in other devices like NoCs that employ a tree interconnect. In Chapter 4 we presented our research related to CST scheduling and configuration. The main problem in a CST is twofold: scheduling – given a set of incompatible communications, partition them into compatible subsets in a distributed manner; and configuration – given a set of compatible communications, create the corresponding paths on the CST in a distributed fashion using only local information available at the leaves. We designed an efficient algorithm to achieve these goals. Our algorithm performed the scheduling and the configuration concurrently in an iterative way. In each iteration the algorithm generated a schedule and configured the CST based on that schedule. We also presented modified versions of the algorithm for special cases like width-1 communication sets and multicasts. Finally, we presented efficient adaptations of our algorithm for an important communication class called well-nested, for which our algorithm is optimal.

Open Problems: While our algorithm has an approximation ratio of 2 for any communication set, there exists another algorithm (developed for optical networks) with an approximation ratio of $\frac{5}{3}$ that is provably optimal [26]. However, our algorithm is considerably simpler and requires minimal computational capability at each node. It is worthwhile to simulate practical optical routing

[2]Refer to Fernandez-Zepeda *et al.* [28] for research related to finding a fault-free sub-R-Mesh in a faulty R-Mesh.

examples to ascertain whether the simplicity of our algorithm is a good trade-off considering its slightly higher approximation ratio.

We proved that our algorithm is optimal for the important communication class of well-nested communication sets. In related research, El-Boghdadi [22] developed power-aware, optimal routing algorithms for well-nested communications and showed that his algorithm is more efficient in terms of power-awareness compared to ours. In the future, one can to explore the possibility of extending our algorithm as well as power-aware CST configuration and scheduling algorithms to other important classes other than well-nested communications.

6.4 Fat-Tree Switch

A fat-tree is an important interconnection structure that is used extensively in many of today's high-performance clusters as well as other areas like high-capacity disk-storage interconnections. In Chapter 5 we presented our preliminary research related to R-Mesh-based routing in fat-tree switches. We presented an algorithm to generate configurations of an R-Mesh to match the input port requests with available output ports of a fat-tree switch. For reference below, call this the *basic algorithm*.

Open Problems: The general idea is to iteratively apply the basic algorithm to schedule and route as many connection requests as possible. The following paragraphs outline an approach to build on the basic algorithm.

Incremental Algorithm: The current approach is to apply the basic algorithm (starting with an R-Mesh with no connections) at each round. That is, a given subset of connection requests is scheduled, routed, then removed from the traffic matrix. Next the current connections are torn down and remaining requests are routed all over again. There may be significant advantages to reusing connections made in previous rounds. In cases where a particular connection can be employed for several packets, the connection can be used across multiple rounds. All these point to the need for an incremental algorithm that builds on a previous configuration, rather than start all over again each time.

Adding Weights: It is well known that good load balancing achieves good throughput. Weights can be associated with input and output ports of the R-Mesh (fat-tree switch) to reflect their loads; for example, a full input port has large weight as it should be addressed as soon as possible. A weight-aware schedule that prioritizes connections by weights could balance loads across the switch and, hence, across the network.

Simulation: Most of the work described above requires simulations both for the evaluation of the methods and for determining parameter values. For example, the simulation can determine the granularity of weights for the ports. Currently, we have a two-level (Boolean) weighting system for the ports. Is much to be gained by changing this to b-bit weights? As another example, the number of rounds needed to schedule a set of communications is a good measure of the algorithm's effectiveness. This can be ascertained by simulations. Finally, overall network performance can also be evaluated in the context of the proposed methods.

6.5 Other Directions

Here we identify research directions other than those tied to specific chapters.

The ideas of CSTs and fat-trees can be generalized to a tree in which switches are connected by links of arbitrary bandwidth. For example, a level k fat-tree switch has 2^k links to the parent and 2^{k-1} links to each child. These quantities are all 1 for the CST. In general a switch can have $f_p(k)$, $f_l(k)$, and $f_r(k)$ links to its neighbors. Particular cases of this abstraction can be of importance in current and future technological settings. For instance, if $f_p(k)$, $f_l(k)$, and $f_r(k)$ are 1 in a fat-tree up to a certain level, k_0, and a different constant value for higher levels, then we have a two-tier tree that is possibly much leaner at the top than a fat-tree. This could be used in settings such as stacked-die interconnects with a relatively lower interconnect density between dies than within a die. The same abstraction works for pin-limited settings that restrict the number of connections between chips and boards.

•••

On the whole, in this dissertation we have studied scheduling and configuration of three broad categories of network switches that are currently used in settings ranging from local and wide area networks to high-performance clusters and networks-on-a-chip. History has taught us that these environment boundaries are not rigid. For example, concepts that were used a decade ago only is long-haul networks are now used within a chip. Our work is at a sufficiently high level of abstraction to allow porting of results from this dissertation across application boundaries and hold relevance beyond the present state-of-the-art.

References

[1] M. Alonso, S. Coll, J. M. Martinez, V. Santonja, P. Lopez, and J. Duato. “Dynamic Power Saving in Fat-Tree Interconnection Networks Using On/Off Links,” *20th Intl. Parallel and Distributed Processing Symposium*, pp. 8, Apr. 2006.

[2] M. Alonso, S. Coll, V. Santonja, J. M. Martinez, P. Lopez, and J. Duato, ”Power-Aware Fat-Tree Networks Using On/Off Links,” *Lecture Notes in Computer Science - Third International Conference on High Performance Computing and Communications*, vol. 4782, pp. 472–483, 2007.

[3] L. Benini and G. De Micheli, “Networks on Chips: A New SoC Paradigm,” *IEEE Computer*, pp. 70–78, Jan. 2002.

[4] A. Bermudez, R. Casado, F. J. Quiles, and J. Duato, “Handling Topology Changes in InfiniBand,” *IEEE Tran. on Parallel and Distributed Systems*, February 2007, (Vol. 18, No. 2), pp. 172–185.

[5] K. Bolding, S-C. Cheung, S-E. Choi, C. Ebeling, S. Hassoun, T. A. Ngo, and R. Wille, “The Chaos Router Chip: Design and Implementation of an Adaptive Router,” *IFIP Transactions A*, vol. A-42, pp. 311–320, 1994.

[6] K. Bolding, S-C. Cheung, S-E. Choi, C. Ebeling, S. Hassoun, T. A. Ngo, and R. Wille, “The Chaos Router Chip: Design and Implementation of an Adaptive Router,” *in Proc. International Conference on VLSI*, pp. 311–320, 1993.

[7] K. Bondalapati and V. K. Prasanna, “Hardware Object Selection for Mapping Loops onto Reconfigurable Architectures,” *Proc. Int. Conf. Par. and Distr. Proc. Techniques and Appl.*, pp. 1104–1110, 1999.

[8] K. Bondalapati and V. K. Prasanna, “Reconfigurable Computing Systems,” *Proc. IEEE*, 2002, vol. 90, no. 7, pp. 1201–1217.

[9] A. Bouhraoua and M. E. Elrabaa, “An Efficient Network-on-Chip Architecture Based on the Fat-Tree (FT) Topology,” *in Proc. International Conference on Microelectronics*, pp. 28–31, 2006.

[10] A. Bouhraoua and M. E. Elrabaa, “Addressing Heterogeneous Bandwidth Requirements in Modified Fat-Tree Networks-on-Chips,” *in Proc. 4th IEEE International Symposium on Electronic Design, Test and Applications*, pp. 486–490, 2008.

[11] J. Carbonaro, and F. Verhoorn, “Cavallino: The Teraflop Router and NIC,” *in Proc. Hot Interconnects Symposium IV*, pp. 157–160, 1996.

[12] H. J. Chao and B. Liu, *High Performance Switches and Routers*, Wiley-IEEE Press, 2007.

[13] S-T. Chuang, A. Goel, N. McKeown, and B. Prabhakar, “Matching Output Queueing with a Combined Input Output Queued Switch,” *IEEE Journal on Selected Areas in Communications*, vol. 17, no. 6, pp. 1030–1039, 1999.

[14] S-T. Chuang, S. Iyer, and N. McKeown "Practical Algorithms for Performance Guarantees in Buffered Crossbars," *in Proceedings of IEEE INFOCOM*, pp. 981–991, 2005.

[15] K. Compton and S. Hauck, "Reconfigurable Computing: A Survey of Systems and Software," *ACM Computing Surveys*, 2002, vol. 34, no. 2, pp. 171–210.

[16] W. J. Dally and B. Towles, *Principle and Practices of Interconnection Networks*, Morgan Kaufmann, 2005.

[17] W. J. Dally, L. R. Dennison, D. Harris, K. Kinhong, and T. Xanthopoulos, "Architecture and Implementation of the Reliable Router," *Proc. Hot Interconnects Symposium II*, pp. 197–208, 1994.

[18] H. P. Dharmasena and R. Vaidyanathan, "The Mesh with Binary Tree Networks: An Enhanced Mesh with Low Bus-Loading," *The Journal of Interconnection Networks*, vol. 5, no. 2, June 2004, pp.131–150.

[19] Z. Ding, R. R. Hoare, and A. K. Jones, "Level-wise Scheduling Algorithm for Fat Tree Interconnection Networks," *in Proc. Super Computing 06*, pp. 9–9, Tampa, Florida, 2006.

[20] J. Duato, S. Yalamanchilli, and L. Ni, *Interconnection Networks an Engineering Approach*, Morgan Kaufmann, 2003.

[21] H. M. El-Boghdadi, "On Implementing Dynamically Reconfigurable Architectures," Ph.D. Thesis, Dept. Electrical and Computer Engg., Louisiana State University, 2003.

[22] H. M. El-Boghdadi, "Power-Aware Routing for Well-Nested Communications on the Circuit Switched Tree," *Journal of Parallel and Distributed Computing*, vol. 69, no. 2, pp. 135–142, 2009.

[23] H. M. El-Boghdadi, R. Vaidyanathan, J. L. Trahan, and S. Rai "Implementing Prefix Sums and Multiple Addition Algorithms for the Reconfigurable Mesh on the Reconfigurable Tree Architecture," *Proc. Int. Conf. Parallel and Distrib. Proc. Techniques and Appl.*, 2002, vol. 3, pp. 1068–1074.

[24] H. M. El-Boghdadi, R. Vaidyanathan, J. L. Trahan, and S. Rai, "On the Communication Capability of the Self-Reconfigurable Gate Array Architecture," *9th Reconfigurable Architectures Workshop* in *Proc. Int. Parallel and Distrib. Proc. Symp*, (2002).

[25] H. M. El-Boghdadi, R. Vaidyanathan, J. L. Trahan, and S. Rai, "On Designing Implementable Algorithms for the Linear Reconfigurable Mesh," *Proc. Int. Conf. on Parallel and Distrib. Proc. Tech. and App*, (2002), pp. 241–246.

[26] T. Erlebach, K. Jansen, C. Kaklamanis, M. Mihail, and P. Persiano, "Optimal Wavelength Routing on Directed Fiber Trees," *Theor. Comput. Sci.*, 221(1-2), pp. 119–137, 1999.

[27] M. Fayyazi, D. Kaeli, and W. Meleis, "Parallel Maximum Weight Bipartite Matching Algorithms for Scheduling in Input-Queued Switches," *Proc. 18th. Intl. Parallel and Distributed Processing Symposium*, pp. 4b, 2006.

[28] J. A. Fernandez-Zepeda, A. Estrella-Balderrama, and A. G. Bourgeois, "Designing Fault Tolerant Algorithms for Reconfigurable Meshes" *Intl. J. Foundations of Computer Sci.*, vol. 16, no. 1, pp. 71–88, 2005.

[29] C. Fraleigh, S. Moon, B. Lyles, C. Cotton, M. Khan, D. Moll, R. Rockell, T. Seely, and S. C. Diot, "Packet-Level Traffic Measurements from the Sprint IP Backbone," *Network, IEEE*, vol. 17, no. 6, pp. 6–16, December 2003.

[30] M. Galles, "Scalable Pipelined Interconnect for Distributed Endpoint Routing: The SPIDER Chip," *in Proc. Hot Interconnects Symposium*, pp. 7–22, 1996.

[31] P. Giaccone, B. Prabhakar, and D. Shah, "Randomized Scheduling Algorithms for High-Aggregate Bandwidth Switches," *IEEE J. Select. Areas Commun.*, vol. 21, pp. 546–559, 2003.

[32] C. Gomez, F. Gilabert, M.E. Gomez, P. Lopez, and J. Duato, "Deterministic versus Adaptive Routing in Fat-Trees," *in Proc. Parallel and Distributed Processing Symposium*, pp. 1–8, 2007.

[33] C. Gomez, F. Gilabert, M. E. Gomez, P. Lopez, and J. Duato, "RUFT: Simplifying the Fat-Tree Topology," *in Proc. International Conference on Parallel and Distributed Systems (ICPADS)*, pp. 153–160, 2008.

[34] R. I. Greenberg and L. Guan, "An Improved Analytical Model for Wormhole Routed Networks with Application to Butterfly Fat-Trees," *Proc. ICPP 97*, pp. 44-48.

[35] M. Hanckowiak, M. Karonski, and A. Panconesi, "On the Distributed Complexity of Computing Maximal Matchings," *SIAM J. Discrete Math.*, vol. 15, no. 1, pp. 41–57, 2001.

[36] T. Hoefler, T. Schneider, and A. Lumsdaine, "Multistage Switches are Not Crossbars: Effects of Static Routing in High-Performance Networks," *Proc. IEEE International Conference on Cluster Computing*, 2008, pp. 116–125.

[37] `http://www.top500.org/`

[38] `http://www.cray.com/Assets/PDF/products/xt/CrayXT5mBrochure.pdf`

[39] `http://www.cisco.com/en/US/products/ps5763/`

[40] `http://www.huawei.com/products/datacomm/detailitem/view.do?id=960&rid=69`

[41] `http://www.huawei.com/products/datacomm/detailitem/view.do?id=958&rid=70`

[42] `http://www.omnetpp.org/`

[43] S. Iyer and N. McKeown, "Maximum Size Matchings and Input Queued Switches," in *40th Annual Allerton Conf. on Communication, Control, and Computing*, 2002.

[44] P. Kelsen, "Optimal Parallel Algorithm for Maximal Matching," *Information Processing Letters*, vol. 52, no. 4, pp, 223–228, 1994.

[45] D. I. Lehn, K. Puttegowda, J. H. Park, P. Athanas, and M. Jones, "Evaluation of Rapid Context Switching on a CSRC Device," *Proc. Intl. Conf. on Engineering of Reconfigurable Systems and Algorithms (ERSA02)*, 2002, pp. 154–160.

[46] F. T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays · Trees · Hypercubes*, Morgan Kaufmann, 1992.

[47] C. E. Leiserson, "Fat Trees: Universal Networks for Hardware Efficient Supercomputing," *IEEE Trans. on Computers,* vol. 34, no. 10, pp. 892-901, 1985.

[48] E. Leonardi, M. Mellia, F. Neri, and M. A. Marsan, "Bounds on Average Delays and Queue Size Averages and Variances in Input-Queued Cell-Based Switches," in *IEEE INFOCOM*, pp. 1095–1103, 2001.

[49] X. Li and I. Elhanany, "Stability of Frame-Based Maximal Weight Matching Algorithms with Reconfiguration Delay," *Workshop on High Performance Switching and Routing*, pp. 942–944, May, 2005.

[50] Y. Li, S. Panwar and H. J. Chao, "Frame-Based Matching Algorithms for Optical Switches", *Proc. Workshop on High Performance Switching and Routing*, pp. 97–102, Jun 2003.

[51] X.-Y. Lin, Y.-C. Chung, and T.-Y. Huang, "A Multiple LID Routing Scheme for Fat-Tree Based Infiniband Networks," *Proc. Int. Parallel and Distrib. Proc. Symp*, pp. 11, 2004.

[52] J. Lou and X. Shen, "Frame-Based Packet-Mode Scheduling for Input-Queued Switches," *to appear in IEEE Transactions on Computers*, July, 2009.

[53] S. Matsumae and N. Tokura, "Simulation Algorithms among Enhanced Mesh Models," *IEICE Transactions on Information and Systems*, vol. E82-D, no. 10, pp. 1324–1137, 1999.

[54] H. Matsutani, M. Koibuchi, and H. Amano, "Performance, Cost, and Energy Evaluation of Fat H-Tree: A Cost-Efficient Tree-Based On-Chip Network," *in Proc. IEEE International Parallel and Distributed Processing Symposium*, 2007.

[55] N. McKeown, "The iSLIP Scheduling Algorithm for Input-Queued Switches," *IEEE/ACM Transactions on Networking*, vol. 7, no. 2, pp. 188–201, 1999.

[56] N. McKeown, A. Mekkittikul, V. Anantharam, and J. Walrand, "Achieving 100% Throughput in an Input-Queued Switch," *IEEE Transactions on Communications*, vol. 47, no. 8, August 1999.

[57] L. Mhamdi, "A Partially Buffered Crossbar Packet Switching Architecture and its Scheduling," *Proc. IEEE Symposium on Computers and Communications*, pp. 942–948, July 2008.

[58] M. Mitzenmacher and E. Upfal, *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*, Cambridge University Press, 2005.

[59] M. J. Neely, E. Modiano, and Y. S. Cheng, "Logarithmic Delay for $n \times n$ Packet Switches Under the Crossbar Constraint," *IEEE/ACM Transactions on Networking*, vol. 15, no. 3, pp. 3–9, June 2007.

[60] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable Parallel Programming with CUDA," *Queue*, vol. 6 , no. 2, pp. 40–53, 2008.

[61] S. R. Ohring, M. Ibel, S. K. Das, and M. J. Kumar, "On Generalized Fat Trees," *Proc. 9th International Parallel Processing Symposium*, pp 37–44, April 1995.

[62] J. H. Pan, T. Mitra, and W-F. Wong, "Configuration Bitstream Compression for Dynamically Reconfigurable FPGAs," *in Proc. IEEE/ACM International Conference on Computer Aided Design*, pp. 766–773, 2004.

[63] D. Pang and Y. Yang, "Localized Independent Packet Scheduling for Buffered Crossbar Switches," *IEEE Transactions on Computers*, vol. 58, no. 2, pp. 260–274, February 2009.

[64] G. Papadopoulos, G. A. Boughton, R. Greiner, and M. J. Beckerle, "*T: Integrating Building Blocks for Parallel Computing," *in Proc. Supercomputing*, pp. 624–635, 1993.

[65] H. Qiu, Y. Li, P. Yi, and JiangXing Wu, "PIFO Output Queued Switch Emulation by a One-cell-Crosspoint Buffered Crossbar Switch," *Proc. Internation Conference on Communications, Circuits and Systems*, June 2006, pp. 1767–1771.

[66] R. Rojas-Cessa and C. Lin, "Captured-Frame Eligibility and Round-Robin Matching for Input-Queued Packet Switches," *IEEE Communications Letters*, vol. 8, no. 9, Sept. 2004, pp. 585–587.

[67] S. Ross, *Probability Models for Computer Science*, Harcourt/Academic Press, 2002.

[68] S. L. Scott, "Synchronization and Communication in T3E Multiprocessor," *in Proc. 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 26–36, 1996.

[69] S. L. Scott and G. Thorson, "The Cray T3E Network, Adaptive Routing in a High Performance 3D Torus," *in Proc. Hot Interconnects Symposium IV*, 1996.

[70] F. O. Sem-Jacobsen and T. Skeie, "Maintaining Quality of Service with Dynamic Fault Tolerance in Fat-Trees," *Lecture Notes in Computer Science - 15th International Conference High Performance Computing*, vol. 5374, pp. 451–464, 2008.

[71] F. O. Sem-Jacobsen, T. Skeie, O. Lysne, and J. Duato, "Dynamic Fault Tolerance with Misrouting in Fat Trees," *in Proc. International Conference on Parallel Processing*, pp. 33–42, 2006.

[72] H. Sethu, C. B. Stunkel, and R. F. Stucke, "IBM RS/6000 SP Interconnection Network Topologies for Large Systems," *Proc. International Conference on Parallel Processing*, 1998, pp. 620–627.

[73] R. Sidhu and V. K. Prasanna, "Efficient Metacomputation Using Self-Reconfiguration," *Proc. 12th. Int. Workshop on Field Prog. Logic and App.*, 2002, Springer Verlag Lecture Notes in Computer Sc., vol. 2438, pp. 698–709.

[74] R. Sidhu, S. Wadhwa, A. Mei, and V. K. Prasanna, "A Self-Reconfigurable Gate Array Architecture," *Int. Conf. on Field Programmable Logic and Applications*, 2000, Springer Verlag Lecture Notes in Computer Sc., vol. 1896, pp. 106–120.

[75] H. Singh, M.-H. Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh, and E. M. Chaves Filho, "MorphoSys: An Integraged Reconfigurable System for Data-Parallel and Computation-Intensive Applications," *IEEE Trans. Comput.*, 2000, vol. 49, no. 5, pp. 465–481.

[76] L. Tassiulas and A. Ephremides, "Stability Properties of Constrained Queueing Systems and Scheduling Policies for Maximum Throughput in Multihop Radio Networks," *IEEE Transactions on Automatic Control*, vol. 37, no. 12, pp. 1936–1949, Dec, 1992.

[77] R. Vaidyanathan and J. L. Trahan, *Dynamic Reconfiguration: Architectures and Algorithms*, Kluwer Academic/Plenum Publishers, 2004.

[78] Z. Wang, K. Zhou, D. Feng, and J. Liu, "Disk Tree: A Fat-Tree Based Heterogeneous Multi-Tier Storage Architecture," *in Proc. 4th International Workshop on Storage Network Architecture and Parallel I/Os*, pp. 47–54, 2007.

[79] Z. Wang, K. Zhou, D. Feng, L. Zeng, and J. Liu, "FTRAID: A Fat-Tree Based Parallel Storage Architecture for Very Large Disk Array," *in Proc. International Conference on Networking, Architecture, and Storage*, pp. 185–192, 2007.

[80] Xilinx, "Virtex Series Configuration Architecture User Guide," Xilinx application note XAPP 151, 2000.

[81] Y. Zheng and W. Gao, "Randomized Parallel Scheduling Algorithm for Input Queued Crossbar Switches," *Proceedings of the Fifth International Conference on Computer and Information Technology*, pp. 424–428, 2005.

[82] J. Zhou, X. Lin, C. Wu, and Y. Chung, "Multicast in Fat-Tree-Based InfiniBand Networks," *Proceedings of the 2005 Fourth IEEE International Symposium on Network Computing and Applications (NCA05)*, pp. 239–242, 2005.

[83] www.zurich.ibm.com/~fab/Osmosis/

Vita

Krishnendu Roy received his bachelor of science with honors in computer science in 2000, and bachelor of technology in information technology in 2003, both from University of Calcutta, Calcutta, India. He joined the Department of Electrical and Computer Engineering at Louisiana State University in August 2003. Krishnendu obtained his master of science in electrical engineering - computers area in December 2005, and is expected to complete his doctor of philosophy in electrical engineering - computers area in August 2009. Krishnendu will join the Mathematics and Computer Science Department at Valdosta State University, Valdosta, Georgia, U.S.A in August 2009.