Algorithmic Aspects of High Speed Switching

by

Saadeddine Mneimneh

Submitted to the Department of Civil and Environmental Engineering
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

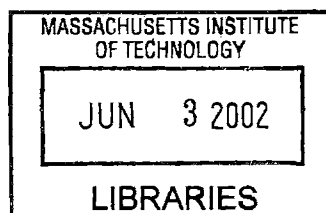
May 2002
[JUNE 2002]

© Massachusetts Institute of Technology 2002. All rights reserved.

Author ....
Department of Civil and Environmental Engineering
May 10, 2002

Certified by. ....
Kai-Yeung Siu
Associate Professor
Thesis Supervisor

Accepted by ....
Oral Buyukozturk
Chairman, Department Committee on Graduate Students



ARCHIVES

Algorithmic Aspects of High Speed Switching

by

Saadeddine Mneimneh

Submitted to the Department of Civil and Environmental Engineering
on May 10, 2002, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

A major drawback of the traditional output queuing technique is that it requires a switch speedup of N, where N is the size of the switch. This dependence on N makes the switch non-scalable at high speeds. Input queuing has been suggested instead. The introduction of input queuing creates the necessity for developing switching algorithms to decide which packets to keep waiting at the input, and which packets to forward across the switch. In this thesis, we address various algorithmic aspects of switching.

We prove in this thesis, that many of the practical switching algorithms still require a speedup to achieve even a weak notion of throughput. We propose two switching algorithms that belong to a family to which we refer in this thesis as priority switching. These two algorithms overcome some of the disadvantages in existing priority switching algorithms, such as the excessive amount of state information that needs to be maintained. We also develop a practical algorithm that belongs to a family to which we refer in this thesis as iterative switching. This algorithm achieves high throughput in practice and offers the advantage of not requiring more than one iteration, unlike other existing iterative switching algorithms which require multiple iterations to achieve high throughput. Finally, we address the issue of using switches in parallel to accommodate for the need of speedup. We study two settings of parallel switches, one with standard packet switching, and one with flow scheduling, in which flows cannot be split across multiple switches.

Thesis Supervisor: Kai-Yeung Siu
Title: Associate Professor

This research was supported by: NSF Award 9973015 (chapters 2, 3, and 6), Alcatel Inc. (chapter 4), and Tellabs Inc. (chapter 5, patent filed).

Acknowledgments

The acknowledgments section is an important part of a Ph.D. thesis because, *most of the times*, it comes out to be sincere. It contains names of people and pointers to the social aspect of the Ph.D. experience. I think a collection of acknowledgements out of many Ph.D. theses may reflect a complete picture of the human social Ph.D. experience.

I defended my thesis few weeks ago, and at the end of my defense, I felt that I was leaving something behind, and I couldn't just stop without thanking a number of people. I said few words. I will try to reproduce these words here because they summarized my MIT social Ph.D. experience.

I will traditionally start by thanking my advisor Sunny Siu. However, my thanking to him is non-traditional, for allowing me towards the ends of my Ph.D., to waste time and do some research, while physically away from MIT in Dallas with my wife, at a time when my wife and I both needed to be close.

I thank Steve Lerman for helping me in the beginnings and encouraging me to apply to MIT. Needless to say, coming to MIT changed my life (for the better, or at least that's what is apparent so far).

I thank Kevin Amaratunga for accepting to serve as a member on my Ph.D. committee in its first meeting on a very short notice (and for staying later on).

I thank Jud Harward for all the fights that we had when we worked together over a period of 4 years because I learned a lot from him.

I thank Cynthia Stewart for facilitating all the departmental logistics and for not going after me and pushing me to fill out the paper work (as she does with other students).

I thank all my MIT teachers for whom I owe most of my knowledge. I mention Arvind, David Gifford, Michael Sipser, Dan Spielman, David Karger, Nancy Lynch, Charles Rhors...

I thank all my fellow lebanese/arab students at MIT who created for me an appropriate microcosm and provided me with a sense of belonging. They shaped and

refined my thoughts through many cultural/political events that we organized together, and through many discussions that never ended. At MIT, we implemented together the "thinking outside the box" when all of us with common backgrounds met on an uncommon ground. The maturity of intellect that I gained from interactions with my lebanese/arab friends makes this document the least important part of my Ph.D. experience (speaking of this document, my friends actually helped me re-produce this document when my laptop computer was stolen few days ago with everything on it and no backup). I would like to mention some names here: Issam Lakkis (aka the Poet), Louay Bazzi (as of today, the smartest person at MIT), Rabih Zbib (aka the expert on *foreign* affairs), Nadine Alameh, Hisham Kassab, Dina Katabi, Omar Baba, Fadi Karameh and Assef Zobian (aka the druze connection), Ibrahim Abou-Faycal (aka "al-za3eem" i.e. the leader, but no one really believes that), Jean-Claude Saghbini (aka the sailor), Mahdi Mattar (aka "al-mountazar" i.e. the expected one), Walid Fayad (aka "malek el..." i.e. the king of...), Ziad Younes, Mona Fawaz, Alan Shihadeh, Husni and Khaled Idris (aka the Idris brothers), Maya Said, Yehia Massound, Karim Hussein, Maysa and Ziad Sabah, Hassan Nayfeh, Bilal Mughal, Richard Rabbat, Ahmad Kreydiyeh (aka kreydiyeh), Maya Farhoud, Ziad Zak, Ziad Ferzly, Bassam Kassab, Danielle Tarraf, Joe Saleh...

I **THANK** my parents Salma and Sami and my family for helping me come to MIT, for supporting me throughout the years (emotionally and financially), for calling me on the phone more than 375 times so far, and for doing every possible thing to make me happy at the cost of loosing my presence as one of the family. I **THANK** them for relieving me from my responsibilities towards the family, and for handling my absence for 7 years. There is nothing I can do to return their sacrifice.

I thank my wife Nora Katabi for helping me through the stressful times, for changing my life to the better, and for being part of it. I thank her for giving me the possibility to wake up every day and look at a face that I like (something I could not achieve by just looking at the mirror), and for giving me hope in life.

I also thank Farhoud, Farkoush, Saeid, and Arnoub el Hazz.

Finally, I thank Allah (i.e. God) for many things...

To my now bigger Family...
I LOVE you.

Second dedication:

To Cambridge, Boston, Charles River, and MIT Steps...

...Never Goodbye.

Contents

1 Introduction **17**

1.1 Output Queuing 18

1.2 The Speedup Problem 19

1.3 Input Queuing 19

1.3.1 HOL Blocking 20

1.3.2 Formal Abstraction 21

1.4 Input-Output Queuing 22

1.5 Traffic Models 24

1.5.1 SLLN Traffic 24

1.5.2 Weak Constant Burst Traffic 25

1.5.3 Strong Constant Burst Traffic 26

1.6 Guarantees 27

1.6.1 Throughput 27

1.6.2 Delay 29

1.7 Existing Switching Algorithms 29

1.7.1 Maximum Weighted Matching 30

1.7.2 Priority Switching Algorithms 31

1.7.3 Iterative Switching Algorithms 33

1.8 Thesis Organization 35

2 Some Lower Bounds on Speedup **37**

2.1 Traffic Assumptions 39

2.2 Priority Scheme 40

2.3 Lower Bounds 42
2.3.1 Output Priority Switching Algorithms 43
2.3.2 Maximum Size Matching 47
2.3.3 Maximal Matching 47
2.3.4 Input Priority Switching Algorithms 48
2.4 Summary 57

3 Two Priority Switching Algorithms **59**
3.1 Earliest Activation Time 60
3.2 Latest Activation Time 62
3.3 Implementation Issues 64
3.3.1 Time and Space Complexity 65
3.3.2 Communication Complexity 65
3.4 Summary 66

4 An Iterative Switching Algorithm **69**
4.1 The π-RGA Switching Algorithm 71
4.2 Stable Priority Scheme π 73
4.3 Bounded Bypass Priority Scheme π 76
4.4 Theoretical Results 78
4.4.1 SLLN Traffic 78
4.4.2 Weak Constant Burst Traffic 79
4.4.3 Strong Constant Burst Traffic 79
4.5 Experimental Results 80
4.5.1 Standard Switch 82
4.5.2 Burst Switch 83
4.5.3 Multiple Server Switch 84
4.6 Summary 87

5 Switching using Parallel Switches with no Speedup **89**
5.1 Motivation 90

5.2 The Parallel Architecture 92
5.2.1 Segmentation 94
5.2.2 Rate Splitting 95
5.2.3 Basic Idea 96
5.3 The Approach 98
5.3.1 Demultiplexer Operation 98
5.3.2 Switching Operation 103
5.3.3 Multiplexer Operation 114
5.3.4 Supporting Higher Line Speeds 117
5.4 Summary 119

6 Scheduling Unsplittable Flows Using Parallel Switches 121

6.1 The Problem 123
6.2 Theoretical Framework 124
6.3 Maximization 125
6.4 Number of Rounds 130
6.5 Speedup 133
6.6 Summary 137

7 Conclusion 139

7.1 Some Lower Bounds on Speedup 140
7.2 Two Priority Switching Algorithms 141
7.3 An Iterative Switching Algorithm 141
7.4 Switching Using Parallel Switches 142
7.5 Unsplittable Flows 142

List of Figures

1-1 Output queued switch 18

1-2 Input queued switch 20

1-3 Input queued switch with *VOQ*s 21

1-4 Formal operation of the input queued switch 22

1-5 Input-Output queued switch with *VOQ*s 23

1-6 Priority switching algorithms 31

1-7 Iterative switching algorithms 34

2-1 The ϕ-Adversary 44

2-2 ℓ-*symmetric* cycles 50

2-3 The 3-*symmetric* ϕ-Adversary 51

4-1 The π-*RGA* switching algorithm 72

4-2 π-*RGA* and *pDRR* with one iteration for $B = 1$ 82

4-3 π-*RGA'* and *pDRR* with one iteration for $B = 256$ 83

4-4 Modified π-*RGA'* and *pDDR* with one iteration for $B = 256$ 84

4-5 *VOQ* activeness $= 3B$ packets with one iteration for $B = 256$ 84

4-6 Multiple server switch model 85

4-7 One iteration and one server for $B = 256$ 86

4-8 One iteration and two servers for $B = 256$ 86

4-9 One iteration and four servers for $B = 256$ 87

5-1 The parallel switches 93

5-2 Possibility of deadlock at the output 95

6-1 The unsplittable flow parallel architecture 123

Chapter 1

Introduction

Switching entails the forwarding of packets in a network towards their destinations. The switching operation occurs locally at a node in the network, usually viewed as a router. A switch is therefore the core component of a router, and hence a packet arriving on a link to the switch has to be forwarded appropriately on another link. In this thesis we look at the issues that arise when we consider high speed switching. These issues are not necessarily apparent from the high level description of the problem above, since the router can determine where to forward a packet by simply looking at the packet header and obtaining the required information. At high speed however, the detailed implementation of this task becomes an important aspect. Intuitively speaking, we can assume that the switch operates in successive time slots where in each time slot some packets are forwarded. Later we will see what packets can be forwarded simultaneously during a single time slot, depending on the switch architecture. We will assume that all packets have the same size and will take the same amount of time to be forwarded. If this is not the case, then we can assume that packets are divided into equal sized chunks that we traditionally call cells. However, we will use the term packet in this document keeping in mind that these packets might represent chunks of a real packet. The length of the time slot is determined by the speed at which the switch can forward packets, and as the time slot becomes shorter, the switch speed becomes higher and the problem of switching becomes more apparent, as we will see next in our first attempt to implement this task.

1.1 Output Queuing

Output queuing is the most intuitive and ideal way of implementing the switching operation. The idea behind output queuing is to make a packet available at its destination as soon as it arrives to the switch. The switch is modeled as a black box with input and output ports. We can assume without loss of generality that the number of input ports and the number of output ports are equal to N.

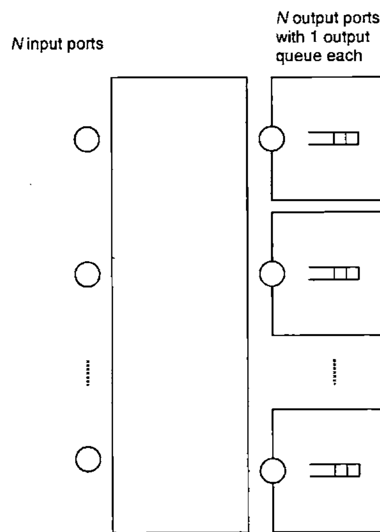


Figure 1-1: Output queued switch

In each time slot, packets arrive at the input ports and are destined to some output ports. At most one packet can arrive to an input port during a single time slot. At each output port, there is a FIFO queue that holds the packets destined to that output, hence the name output queuing. When a packet destined to output j arrives to the switch, it is immediately made available at output j by storing it in the appropriate output queue. At the end of the time slot, at most one packet can be read from each output queue. This is very idealistic and no scheme can do better since each packet is made available at its destination as soon as possible. However, as we will see in the following section, this scheme is very problematic at high speed.

1.2 The Speedup Problem

It is possible that during a single time slot, the output queued switch will forward multiple packets to the same output queue. For instance, if during a time slot, packets at different inputs arrive to the switch and they all need to go to a particular output j, then the switch has to store all these packets in the output queue corresponding to output j. Therefore, up to N packets can go to a particular output queue during a single time slot. This implies that the memory speed of that queue has to be N times more than the line speed, which is limited to one packet per time slot. At a moderate line speed, this does not constitute a problem. However, output queuing becomes hard to scale at high speed. The line speed can be high enough to make the speedup factor N impractical to achieve. Therefore, the use of output queuing becomes unfeasible at high speed. We need a way to eliminate the undesired speedup. In order to overcome the speedup problem, we restrict the number of packets forwarded to an output port to one per time slot. As a result, an alternative architecture in which packets are queued at the input is suggested. The architecture, called Input Queuing, will make it possible to forward at most one packet to each output port and thus eliminates the need for a speedup.

1.3 Input Queuing

In input queuing, FIFO queues are used at the input ports instead of the output ports as depicted in the figure below.

A packet that cannot be forwarded to its output port during a time slot will be kept in its queue at the input. Note that no output queues are needed in this architecture since at most one packet will be forwarded to an output port during a single time slot. This packet will be consumed by the output port by the end of the time slot, and hence there will be no need to store any packets at the output. In order not to recreate the same speedup problem at the input side however, only one packet will be forwarded from an input port during a single time slot as well. Therefore, the

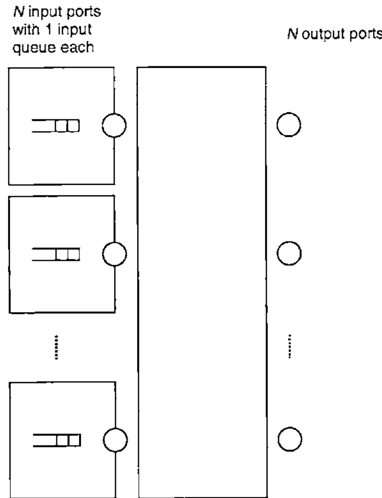


Figure 1-2: Input queued switch

set of packets that are forwarded during a particular time slot satisfies the condition that no two packets will share an input or an output. In other terms, among the forwarded packets, no two packets originate at the same input and no two packets are destined to the same output. We will see later how we can formally abstract this notion. Before doing so, let us examine a phenomenon that arises with input queuing known as Head Of Line blocking.

1.3.1 HOL Blocking

Head Of Line (HOL) blocking occurs when a packet at the head of the queue blocks all the packets behind it in the queue from being forwarded. This phenomenon can occur with input queuing when at a given time slot, two (or more) packets at different input ports need to be forwarded to the same output port, say output port j. Only one of these two packets can be forwarded; therefore, the one that will remain at the head of its queue will block other packets in the queue (which are possibly destined to outputs other than output j) from being forwarded. The HOL blocking phenomenon usually limits the throughput of the input queued switch [17]. One way to eliminate HOL blocking is by virtually dividing each input queue in to N queues, called Virtual Output Queues VOQs. A VOQ at an input will hold packets that are destined to one of the N outputs. Therefore, these VOQs can be indexed by both their input

and output ports. We denote by VOQ_{ij} the VOQ at input i holding packets destined to output j. In this way, two packets that are destined to different output ports cannot block each other since they will be stored virtually in two different queues. The architecture is depicted below:

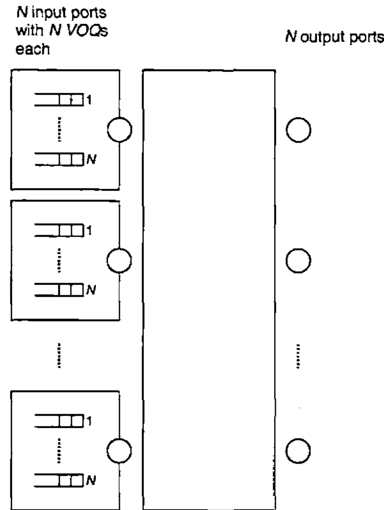


Figure 1-3: Input queued switch with VOQs

In the next section, we provide a formal abstraction for the operation of the input queued switch. We will see that the operation of the input queued switch can be modeled as a computation of a matching (definition below) in every time slot.

1.3.2 Formal Abstraction

We address in this section the question of how to formally abstract the operation of the input queued switch. We know that we can forward at most one packet from an input port and at most one packet to an output port during a single time slot. What is the theoretical framework that will give us this property? It is going to be the notion of a matching. Intuitively speaking, the switch will match input ports to output ports during each time slot. We start with few simple definitions:

Definition 1.1 (graph) *A graph $G = (V, E)$ consists of two sets V and E where V is a set of nodes and E is a set of edges. Each edge in E connects two nodes in V.*

The above is the standard definition of an undirected graph. Next we define a special type of graphs called a bipartite graph.

Definition 1.2 (bipartite graph) *A bipartite graph $G = (L, R, E)$ is a graph in which the set of nodes $V = L \cup R$ is such that L and R are disjoint and every edge in E connects a node in L to a node in R.*

We now define the matching.

Definition 1.3 (matching) *A matching in a graph $G = (V, E)$ is a set of edges in E that are node disjoint.*

Given the above definitions, we can now formally describe the operation of the input queued switch. In every time slot, the switch performs the following:

Formal Abstraction

let VOQ_{ij} be the jth queue at input i
construct a bipartite graph $G = (L, R, E)$ as follows:
  an input port i becomes node i in L
  an output port j becomes node j in R
  a non-empty VOQ_{ij} becomes edge (i, j) in E
compute a matching M in the bipartite graph $G = (L, R, E)$

Figure 1-4: Formal operation of the input queued switch

Since each edge represents a non-empty VOQ, the matching represents a set of packets (the HOL packet of each VOQ). Furthermore, since a matching is a set of edges that are node disjoint, the matching guarantees that these packets do not share any input or output ports, and hence they can be forwarded with no speedup.

1.4 Input-Output Queuing

Although we developed our theoretical framework for an input queued switch based on the idea that the switch has no speedup, it is possible to consider an input queued switch with speedup. In fact, it has been shown that a limited speedup (independent of N) is useful for providing certain guarantees in an input queued switch [6], [7], [18].

However, as before, this requires the use of output queues at the output ports as well since more than one packet can be forwarded to an output port during a single time slot. We call such an architecture an input-output queued switch. Below we present an input-output queued switch with VOQs.

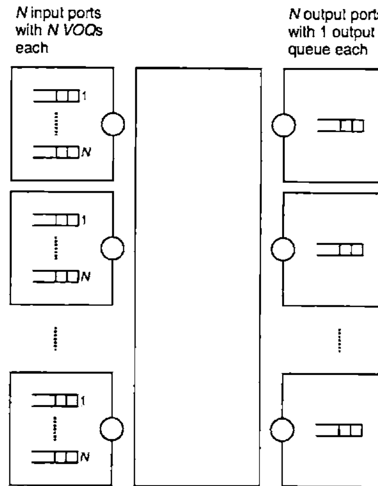


Figure 1-5: Input-Output queued switch with VOQs

Our theoretical framework based on matchings can still be used. However, an input-output queued switch with speedup will be able to compute matchings at a rate higher than one matching per time slot. For instance, with a speedup of 2, an input-output queued switch will compute two matchings per time slot. In general, the speedup needs not be necessarily an integer. We will model the input-output queued switch with continuous time as follows: With a speedup $S \geq 1$, the switch computes a matching every $\frac{1}{S}$ time units, keeping in mind that S needs not be an integer. The line speed will be one packet per time unit and hence S is, as before, a speedup with respect to the line speed. Therefore, the switch will have successive matching phases where each matching phase takes $\frac{1}{S}$ time units. When $S = 1$, i.e. a matching phase takes exactly one time unit, we get back our previous model of input queuing with no speedup. Note that in this case, no queues are necessarily required at the output.

To summarize what has been presented so far, we eliminated the speedup of N required with the idealistic output queuing using input queuing, by replacing the output queues with input queues instead. Moreover, we eliminated the phenomenon

known as HOL blocking by virtually splitting each input queue into N VOQs. We formally modeled the operation of the input queued switch as a computation of a matching in every time slot. Finally, we generalized our model to an input-output queued switch with a continuous time framework and a possible speedup S, where the switch computes a matching every $\frac{1}{S}$ time units.

The question to ask now is why aren't we done with the problem of switching. The answer to this question is the following: what we did so far is reduce the problem of switching into a problem of computing a matching in a bipartite graph. A graph contains possibly many matchings and, therefore, we need to decide on which matching to choose. This decision problem is at the heart of performing the switching operation in input queued switches. As it will be seen in Chapter 2, if we are not careful on which matchings to choose, it is possible for some VOQs to become starved and grow indefinitely. Therefore, some algorithms have been suggested in order to compute the matchings (one every $\frac{1}{S}$ units with a speedup S) without starving the VOQs (more formal definitions of this guarantee appear in Section 1.6). Before we look at some of these algorithms, we need to understand some aspects pertaining to the traffic of packets at the input. For this, we assume the existence of a traffic model.

1.5 Traffic Models

In this thesis, we will present three traffic models. A traffic model describes the arrival of packets to the switch as a function of time. A traffic model can be probabilistic or deterministic as it will be seen shortly. Before we proceed to the different traffic models, we need to define a quantity that tracks the number of packets arriving to the switch. Let $A_{ij}(t)$ be the number of packets arriving to the switch by time t that originate at input i and are destined to output j.

1.5.1 SLLN Traffic

This traffic is a probabilistic model that obeys the Strong Law of Large Numbers, hence its name SLLN. The Strong Law of Large Numbers says that if we have indepen-

dent and identically distributed (i.i.d.) random variables X_n, then $Pr[\lim_{n\to\infty} \frac{\sum_n X_n}{n} = E[X]] = 1$, where $E[X]$ is the expected value of the random variable X_n. We say that $\frac{\sum_n X_n}{n}$ converges to $E[X]$ with probability 1. In our context, regardless of whether packet arrivals are i.i.d. or not, we assume that $\lim_{t\to\infty} \frac{A_{ij}(t)}{t} = \lambda_{ij}$ with probability 1, for some λ_{ij}. In simpler terms, this means that it is possible to define a rate λ_{ij} for the flow of packets from input i to output j.

SLLN:

- $\lim_{t\to\infty} \frac{A_{ij}(t)}{t} = \lambda_{ij}$ with probability 1
- $\sum_k \lambda_{ik} \leq \alpha$
- $\sum_k \lambda_{kj} \leq \alpha$
- $\alpha \leq 1$

The second and third conditions of the SLLN model constrain the sum of rates at every input and output port to be less than or equal to α, which we will call the *loading* of the switch. Finally, we require that $\alpha \leq 1$. The reason behind this last constraint is that the traffic cannot exceed the line speed at any port, which is limited to one packet per time unit. Another reason behind this constraint is that a switch with no speedup cannot access more than one packet per time unit at any port, and hence a switch with no speedup will be overloaded if $\alpha > 1$. This constraint on the loading of the switch will be present in all the traffic models presented hereafter.

With the above probabilistic traffic model, it is possible to define a rate for the flow of packets from input port i to output port j. Next we define two traffic models where this rate does not necessarily exist; however, the models will characterize the traffic burst.

1.5.2 Weak Constant Burst Traffic

In some sense, the weak constant burst traffic is a stronger model than SLLN because it is deterministic. However, it does not define a rate for the flow of packets from

input i to output j. Alternatively, it provides a bound on the burst of packets at input i and output j. This bound is a constant independent of time. Nevertheless, the model does not constrain the flow of packets from input i to output j in any way, hence the use of the term *weak* in the burst characterization of this traffic model.

Weak Constant Burst

- $\forall t_1 \leq t_2$, $\sum_k A_{ik}(t_2) - A_{ik}(t_1) \leq \alpha(t_2 - t_1) + B$
- $\forall t_1 \leq t_2$, $\sum_k A_{kj}(t_2) - A_{kj}(t_1) \leq \alpha(t_2 - t_1) + B$
- $\alpha \leq 1$

The model simply says that for any time interval $[t_1, t_2]$, the maximum number of packets that can arrive at an input i or destined to an output j is at most $\alpha(t_2 - t_1) + B$, where B is a constant independent of time and, as before, α is the loading of the switch.

Note that α is not necessarily the rate of packets at input i or output j. In fact, such a rate might not be defined. Thus, α is just an upper bound on the rate if it exists. Next we define a stronger traffic model that also satisfies this constant burst property.

1.5.3 Strong Constant Burst Traffic

The following model implies the previous model and hence is stronger (more constrained).

Strong Constant Burst

- $\forall t_1 \leq t_2$, $A_{ij}(t_2) - A_{ij}(t_1) \leq \lambda_{ij}(t_2 - t_1) + B$
- $\sum_k \lambda_{ik} \leq \alpha$
- $\sum_k \lambda_{kj} \leq \alpha$
- $\alpha \leq 1$

The model basically says that during any time interval $[t_1, t_2]$, the number of packets from input i to output j is at most $\lambda_{ij}(t_2 - t_1) + B$, where B is a constant independent of time. As before, although λ_{ij} is not necessarily the rate of the flow of packets from input i to output j (and such a rate might not exist), it is an upper bound on the rate if it exists. We have the same constraints as before on the sum of λ_{ij}s at any input or output port. This model of course implies the weak constant burst model.

Note that both the weak constant burst and the strong constant burst models do not necessarily imply the SLLN model because $\lim_{t\to\infty} \frac{A_{ij}(t)}{t}$ might not exist. However, if that limit exists, then the strong constant burst model satisfies the SLLN model.

1.6 Guarantees

There are various service guarantees that one might want a switching algorithm to provide. In this thesis, we will address two basic guarantees. These are throughput and delay guarantees.

1.6.1 Throughput

Throughput basically means that as time evolves, the switch will be able to forward all the packets that arrive to the switch. There are many definitions of throughput and some definitions depend on the adopted traffic model. One possible definition of throughput under a probabilistic traffic model is for the expected length of each VOQ to be bounded. Therefore, if $X_{ij}(t)$ denotes the length of VOQ_{ij} at time t, we require that $E[X_{ij}(t)] \leq M < \infty$ [21], [23], [24]. One can show that this implies that for any $\epsilon > 0$, there exists a time t_0 such that for every $t \geq t_0$, $Pr[\frac{X_{ij}(t)}{t} \geq \epsilon] \leq \epsilon$. We call this type of convergence, convergence in probability. Therefore, $\frac{X_{ij}(t)}{t}$ converges to 0 in probability. Convergence in probability is weaker than convergence with probability 1 (see previous section). Other definitions of throughput require that under an SLLN traffic, $\lim_{t\to\infty} \frac{D_{ij}(t)}{t} = \lambda_{ij}$ with probability 1 [8], where $D_{ij}(t) = A_{ij}(t) - X_{ij}(t)$. Therefore, if $\lim_{t\to\infty} \frac{A_{ij}(t)}{t} = \lambda_{ij}$ in probability, the previous definition of throughput

implies that $\lim_{t\to\infty} \frac{D_{ij}(t)}{t} = \lambda_{ij}$ in probability. It is possible to show that if $E[X_{ij}^2(t)]$ is bounded, then $\lim_{t\to\infty} \frac{X_{ij}(t)}{t} = 0$ with probability 1, which in turn implies that $\lim_{t\to\infty} \frac{D_{ij}(t)}{t} = \lambda_{ij}$ with probability 1 if $\lim_{t\to\infty} \frac{A_{ij}(t)}{t} = \lambda_{ij}$ with probability 1.

In this thesis, we will use two definitions of throughput. A weak definition and a strong definition.

Definition 1.4 (weak throughput) *Let $X_{ij}(t)$ be the length of VOQ_{ij} at time t. Then $\lim_{t\to\infty} \frac{X_{ij}(t)}{t} = 0$.*

The above definition can be also expressed as follows: for every $\epsilon > 0$, there exists a time t_0 such that for any time $t \geq t_0$, $\frac{X_{ij}(t)}{t} \leq \epsilon$.

Note that in the above definition, the throughput does not rely on the fact that $\lim_{t\to\infty} \frac{A_{ij}(t)}{t}$ exists. Note also that the definition does not impose any strict bound on the size of the VOQs. Below we provide a stronger definition of throughput.

Definition 1.5 (strong throughput) *Let $X_{ij}(t)$ be the length of VOQ_{ij} at time t. Then there exists a bound k such that $X_{ij}(t) \leq k$ for all t.*

Obviously, strong throughput implies weak throughput.

It is useful to ensure that the queue size is bounded at any time since this will provide an insight to how large the queues need to be in practice. Most of the time however, this notion of strong throughput can be superseded by the delay guarantee described below. We will rely on the notion of weak throughput in Chapter 2 for proving some negative results on speedup, namely that some switching algorithms cannot achieve weak throughput without speedup.

If we have a throughput guarantee and the loading of the switch is α, we usually refer to this as α throughput. This notion is useful if we would like to observe the throughput guarantee as we change the loading of the switch. If there is a value α of the loading beyond which the switching algorithm cannot guarantee throughput, then we say that the algorithm guarantees α throughput.

1.6.2 Delay

Delay is a stronger guarantee than throughput and it basically means that a packet will remain in the switch for at most a bounded time.

Definition 1.6 (delay) *Every packet remains in the switch for at most a bounded time D.*

Obviously, delay implies strong throughput. To see why this is true, define $k = \lceil SD \rceil$ where S is the speedup of the switch. If the length of VOQ_{ij} exceeds k, then at least one packet will remain in VOQ_{ij} for more than D time units since the switch can forward at most $\lceil SD \rceil$ packets during an interval of time D from VOQ_{ij}, hence violating the delay bound. Therefore, the length of VOQ_{ij} cannot exceed k.

1.7 Existing Switching Algorithms

Now that we have defined some traffic models and possible guarantees, we can enumerate some of the existing switching algorithms. Recall that these will determine how to compute a matching every $\frac{1}{S}$ time units with a speedup S. So we will first consider some properties of matchings in general.

Definition 1.7 (maximal) *A matching M is maximal if there is no edge $(i, j) \notin M$ such that $M \cup (i, j)$ is a matching.*

In simpler terms, a maximal matching is a matching such that no edge can be added to it without violating the property of a matching. Therefore, any edge outside the matching shares a node with at least one edge in the matching.

Definition 1.8 (maximum size) *A matching M is a maximum size matching if there is no other matching M' such that $|M'| > |M|$.*

In simpler terms, a maximum size matching is a matching with the maximum possible number of edges. As a generalization to the maximum size matching we have the following definition.

Definition 1.9 (maximum weighted) *In a weighted graph where edge (i,j) has weight w_{ij}, a matching M is a maximum weighted matching if there is no other matching M' such that $\sum_{(i,j)\in M'} w_{ij} > \sum_{(i,j)\in M} w_{ij}$.*

In simpler terms, a maximum weighted matching is a matching that maximizes the sum of weights of its edges.

The following sections describe some of the existing switching algorithms and the ways by which they compute the matchings.

1.7.1 Maximum Weighted Matching

This algorithm has been known for a while and is one of the first switching algorithms suggested in the literature. It is based on computing a maximum weighted matching as follows. In every matching phase, the weight of edge (i,j), w_{ij}, is set according to some scheme. Then a maximum weighted matching based on these weights is computed. When w_{ij} is the length of VOQ_{ij} (or the time the oldest packet of VOQ_{ij} has been waiting in VOQ_{ij}) it has been shown that the expected length of any VOQ (or the expected wait for any packet) is bounded, with no speedup ($S = 1$) under an i.i.d. Bernoulli traffic in which a packet from input i to output j arrives to the switch with probability λ_{ij} (this satisfies SLLN) [21], [23]. In [28], which addresses a more general setting than an input queued switch, similar (but more elaborate) guarantees are provided using w_{ij} as the length of VOQ_{ij}, without assuming that arrivals are Bernoulli arrivals, but requiring the arrival process to have a finite second moment. When w_{ij} is the length of VOQ_{ij}, another result shows that this algorithm guarantees weak throughput with probability 1 under any SLLN traffic with no speedup ($S = 1$) [8]. Unfortunately, this switching algorithm has a time complexity of $O(NM \log_{(2+\frac{M}{N})} N)$, where M is the number of non-empty VOQs (i.e. edges in the bipartite graph, which could be $O(N^2)$ making the required time $O(N^3)$). This is the best known time required to compute a maximum weighted matching in a bipartite graph [27]. This is not very practical at high speed. A variation on the definition of the weights can reduce the problem of computing a maximum weighted matching to computing a

maximum size matching [24]. This will have a time complexity of $O(\sqrt{N}M)$, which is the best known time required to compute a maximum size matching in a bipartite graph [27]. Unfortunately, $O(N^{2.5})$ time complexity is still not practical at high speed. Therefore, alternative switching algorithms have been suggested.

1.7.2 Priority Switching Algorithms

In order to overcome the complexity of the above switching algorithms, which are based on computing a maximum weighted matching, a family of algorithms that compute a matching based on a priority scheme emerged. Below is the general framework by which these algorithms compute their matchings.

Priority Switching Algorithm

start with an empty matching $M = \emptyset$
prioritize all VOQs
repeat the following until M is maximal
  choose a non-empty VOQ_{ij} with a highest priority
  if $M \cup (i, j)$ is a matching, then $M = M \cup (i, j)$
  discard VOQ_{ij}

Figure 1-6: Priority switching algorithms

Obviously, the time required to compute the priorities has to be efficient (for instance, it has to be $o(N^3)$); otherwise, the use of such an algorithm is not justified. As an example, we can think of an algorithm that operates as follows: it computes a maximum weighted matching M as described in Section 1.7.1, and then assigns high priorities to all VOQ_{ij} such that $(i, j) \in M$. Finally, it performs the algorithm outlined in Figure 1-6 based on these priorities. This is a priority switching algorithm that provides the same guarantees as the maximum weighted matching algorithm. However, the use of this algorithm is not justified because it requires $O(N^3)$ time to compute the priorities. Therefore, a requirement for the use of a priority switching algorithm is that the priority scheme itself is efficient to obtain.

Many priority schemes have been suggested. One algorithm called *Central Queue* [16] assigns higher priority to VOQs with larger length (the way the algorithm is presented here is slightly different than how it was originally presented in [16]). This

algorithm can be shown to guarantee strong throughput with no speedup when $\alpha < \frac{1}{2}$ if $A_{ij}(t)$ does not exceed $\lambda_{ij}t$ by more than a constant for any time t. Moreover, if $A_{ij}(t)$ is always within a constant from $\lambda_{ij}t$, it was proved to provide a delay guarantee with no speedup when $\alpha < \frac{1}{2}$. This algorithm, of course, requires the switch to be less than half loaded.

Another algorithm called *Oldest Cell First* [6] assigns higher priority to VOQs with older HOL packet, where the age of the packet is determined by the time it has been waiting in its VOQ. This was proved to provide a delay guarantee under a weak constant burst traffic with a speedup $S > 2$. It also provides strong throughput under a strong constant burst traffic with a speedup of 2.

Yet another algorithm called *Lowest Occupancy Output Queue First LOOFA* [18] assigns higher priority to a VOQ_{ij} for which output queue j contains smaller number of packets (recall the architecture of an input-output queued switch). A special version of this algorithm, where ties are broken among equal priority VOQs using the age of their HOL packets, provides a delay guarantee under a strong constant burst traffic with a speedup of 2.

In Chapter 3, we are going to describe two priority switching algorithms that we propose. Both algorithms provide strong throughput with a speedup $S = 2$ and a delay guarantee with a speedup $S > 2$ under appropriate traffic models. The advantage of these two algorithms is that they require a considerably smaller amount of state information to compute the priorities than the previous priority switching algorithms.

Obviously, regardless of what the priority scheme is, the time complexity of a priority switching algorithm is $\Omega(N^2)$. Although this is still considered impractical at high speed, as discussed above these algorithms provide delay guarantees with appropriate traffic models and speedup. In Chapter 2, we will have a more general look at these algorithms and prove that the speedup requirement is inherent for these algorithms to provide even a weaker guarantee, like throughput, under a very restricted traffic model.

1.7.3 Iterative Switching Algorithms

So far, all the switching algorithms mentioned above require the need for a centralized global computation of matchings, which is the reason behind their high computational complexity. To overcome this requirement, a family of algorithms, called iterative, has been suggested. In these algorithms, the matching is computed in a distributed fashion where input and output ports interact independently in a simultaneous way. Such algorithms exploit some degree of parallelism in the switch that is acceptable, and in fact they were found to be very practical to implement in hardware.

As the name indicates, an algorithm belonging to this family works in multiple iterations within every matching phase, where in each iteration a partial matching is computed according to the following RGA (stands for *Request*, *Grant*, *Accept*) protocol. In each iteration, inputs and outputs interact independently in parallel: each unmatched input requests to be matched by sending requests to some outputs. Then each unmatched output grants at most one request. Finally, each unmatched input accepts at most one grant. If input i accepts a grant from output j, i and j are matched to each other. It is obvious that the outcome of the RGA protocol is a matching since each output grants at most one request and each input accepts at most one grant.

Since multiple inputs can request the same output, and similarly, multiple outputs can grant the same input, the matching computed in one iteration is not necessarily maximal. For instance, an input receiving multiple grants has to accept only one of them and reject the others. This implies that some of the granting outputs could have granted other requests, but since there is no direct communication among the output ports themselves, this cannot be anticipated. Nevertheless, the size of the matching may grow with more iterations. As we will see in Chapter 4, these iterative switching algorithms do not provide high throughput (i.e. throughput for high values of α) unless multiple iterations are allowed. The general framework of these algorithms is outlined in Figure 1-7 [1].

[1]Some iterative switching algorithms allow for an input and an output to be unmatched in a future iteration in favor of another matching, based on priorities at the input and output ports.

Iterative Switching Algorithm

start with an empty matching $M = \emptyset$
repeat for a number of iterations
  R: unmatched input i *Requests* some outputs
  G: unmatched output j *Grants* at most one request
  A: unmatched input i *Accepts* at most one grant
  if input i accepts a grant from output j
    $M = M \cup (i, j)$

Figure 1-7: Iterative switching algorithms

Iterative switching algorithms differ by how requests are prepared and how grants and accepts are issued. Examples of these algorithms are *PIM* (parallel iterative matching) [1], *iSLIP* [22], *iPP* (iterative ping-pong) [13], *DRR* (dual round robin) [20], and *pDRR* (prioritized dual round robin) [9]. In *PIM*, an unmatched input i sends requests for all outputs j such that VOQ_{ij} is non-empty. An output grants a request at random. Similarly, an input accepts a grant at random. This algorithm was proved to attain a maximal matching in $O(\log N)$ expected number of iterations and provides high throughput in practice. However, the impracticality that randomness brings at high speed lead to the development of the alternative algorithm *iSLIP*. *iSLIP* replaces randomness with the round robin order. As a result, each output maintains a pointer to the inputs, and grants a request by moving the pointer in a round robin fashion until it hits a requesting input. The accepts are issued in a similar manner at the inputs. Other iterative algorithms (except for *iPP*) are variations on this idea. The time complexity of these algorithms is dominated by the complexity of one iteration, which basically consists of the RGA protocol. Depending on the algorithm, this could be $O(\log N)$ or $O(N)$, keeping in mind that ports operate in parallel. These algorithms provide a better alternative at high speed; however, they do no provide strong theoretical guarantees as we will see in Chapter 4.

Figure 1-7 does not reflect that possibility. Such algorithms are usually based on computing what is knows as stable marriage matchings [12] where input and output ports change their match repeatedly in successive iterations until the matching is stable and no more changes occur. Stable marriage matching algorithms require in general N^2 iterations to stabilize. For an example, see [7] which presents an emulation of output queuing using an input queued switch with a speedup of 2.

1.8 Thesis Organization

The thesis is organized as follows. Chapter 2 will establish some lower bounds on the speedup required for different classes of switching algorithms to guarantee throughput. In Chapter 3, we propose two priority switching algorithms. The two algorithms will provide strong throughput with a speedup $S = 2$ and a delay guarantee with a speedup $S > 2$ under appropriate traffic models. They offer the advantage of requiring a smaller amount of state information than other priority switching algorithms. In Chapter 4, we propose an iterative switching algorithm that provides high throughput in practice with one iteration only. The algorithm will also provide, with only one iteration, a delay guarantee with a speedup $S > 2$ as well as strong throughput with a speedup of 2. The property of requiring one iteration only makes it possible to scale the switch at higher speeds since one matching phase will need to fit only one iteration of the RGA protocol described above. Chapter 5 will investigate the use of multiple input-output queued switches with no speedup in parallel in order to achieve a delay guarantee while eliminating the speedup requirement imposed on the switch. Chapter 6 continues with the idea of using parallel switches (not necessarily input-output queued) and exploits a setting in which flows cannot be split across multiple switches. Finally, we conclude the thesis in Chapter 7.

Chapter 2

Some Lower Bounds on Speedup

In this chapter, we establish some lower bounds on the speedup required to achieve throughput with different classes of switching algorithms. We will use the notion of weak throughput defined in Chapter 1. This will strengthen the results since an algorithm that cannot achieve weak throughput, cannot achieve strong throughput as well. We show a lower bound on the speedup for two fairly general classes of priority switching algorithms: input priority switching algorithms and output priority switching algorithms. These are to be defined later in the chapter, but for now, an input priority scheme prioritizes the VOQs based on the state of the VOQs while an output priority scheme prioritizes the VOQs based on the output queues. For output priority switching algorithms, we show that a speedup of 2 is required to achieve weak throughput. We also show that a switching algorithm based on computing a maximum size matching in every matching phase does not imply weak throughput unless $S \geq 2$. The bound of $S \geq 2$ is tight in both cases above based on a result in [8]. The results states that when $S \geq 2$, a switching algorithm that computes a maximal matching in every matching phase, achieves weak throughput with probability 1 under an SLLN traffic. Finally, we show that a speedup of $\frac{3}{2}$ is required for the class of input priority switching algorithms to achieve weak throughput.

Our model of a switch will be essentially the same general model of an input-output queued switch depicted in Figure 1-5. As before, the switch operates in matching phases, computing a matching in every phase. We will assume that the

switch computes a maximal matching in every phase. A switch with speedup S takes $\frac{1}{S}$ time units to complete a matching phase before starting the next phase. Therefore, if $S > 1$, output queues are also used at the output ports since packets will be forwarded to the output at a speed higher than the line speed. We review below some of the known results regarding the speedup of the switch.

Charny et al. proved in [6] that any maximal matching policy (i.e. any switching algorithm that computes a maximal matching in every matching phase) achieves a bounded delay on every packet in an input queued switch with a speedup $S > 4$ under a weak constant burst traffic. We will prove that the simple policy of computing any maximal matching does not imply weak throughput for a speedup $S < 2$. In fact, as mentioned earlier, we prove that even a maximum size matching policy does not imply weak throughput for $S < 2$.

Since switches with speedup are not desired due to their manufacturing cost and impracticality, it is very legitimate to look at what loading α a switch with no speedup (i.e. $S = 1$) can tolerate. The first work that addresses this issue appears in [16]. They provided a switching algorithm (called *Central Queue* algorithm) that computes a $\frac{1}{2}$-approximation of the maximum weighted matching, where they used the length of VOQ_{ij} as the weight for edge (i, j) (recall the required restrictions on the traffic described in Section 1.7.2 for this algorithm to provide throughput and delay guarantees). This work is a generalization of the result described in [28] applied to the special setting of a switch. The $\frac{1}{2}$-approximation algorithm used in [16] is a priority switching algorithm where VOQs with larger length are considered first as candidates for the matching. The *Central Queue* algorithm achieves strong throughput when $\alpha < \frac{1}{2}$. The results obtained in this chapter will prove that it cannot achieve weak throughput unless $S \geq \frac{3}{2}\alpha$, and hence with no speedup ($S = 1$) it cannot achieve weak throughput for $\alpha > \frac{2}{3}$.

In [6], the authors provide an algorithm called *Oldest Cell First* that guarantees a bounded delay on every packet with a speedup $S > 2$ under a weak constant burst traffic. The same algorithm can be proved to achieve strong throughput with a speedup of 2 under a strong constant burst traffic. This switching algorithm is a

priority switching algorithm and assigns higher priority to VOQs with older HOL packets. We will similarly prove that this algorithm cannot achieve weak throughput unless $S \geq \frac{3}{2}$.

In another work [18], Krishna et al. provide an algorithm called *Lowest Occupancy Output Queue First LOOFA* that guarantees a bounded delay on every packet with a speedup of 2 and a strong constant burst traffic, and uses a more sophisticated priority scheme. This algorithm has also a work conservation property that we are not going to address here. The same lower bound of $S \geq \frac{3}{2}$ applies for this algorithm as well in the sense that *LOOFA* does not imply weak throughput unless $S \geq \frac{3}{2}$.

2.1 Traffic Assumptions

We define a restricted model of traffic under which we are going to prove our lower bound results on S. Note that a more restricted traffic yields stronger results.

Definition 2.1 *An α-shaped traffic is a traffic that satisfies the following:*

- $\forall t_1 \leq t_2$, $A_{ij}(t_2) - A_{ij}(t_1) = \lambda_{ij}(t_2 - t_1) \pm O(1)$, *where λ_{ij} is a constant*
- $\forall t_1 \leq t_2$, $\sum_k A_{ik}(t_2) - A_{ik}(t_1) = \sum_k \lambda_{ik}(t_2 - t_1) \pm O(1)$
- $\forall t_1 \leq t_2$, $\sum_k A_{kj}(t_2) - A_{kj}(t_1) = \sum_k \lambda_{kj}(t_2 - t_1) \pm O(1)$
- $\sum_k \lambda_{ik} \leq \alpha$
- $\sum_k \lambda_{kj} \leq \alpha$
- $\alpha \leq 1$

The above conditions state that the rate of the flow from input i to output j exists and is equal to λ_{ij}. Moreover, the burst $B = O(1)$ of the flow from input i to output j, as well as the aggregate flow at any input and any output, is independent of the size of the switch N. Note that this traffic satisfies the SLLN model as well as the strong constant burst model.

The α-*shaped* traffic is the model under which we are going to prove the various lower bound results. As a consequence, the results will hold for all traffic models defined in Chapter 1, namely the SLLN traffic, the weak constant burst traffic, and the strong constant burst traffic.

2.2 Priority Scheme

In this section, we formally define a priority scheme. Recall from Chapter 1 that a priority scheme imposes an order on the VOQs by which they are considered for the matching. We first define an active VOQ to be a non-empty VOQ.

Definition 2.2 *An active VOQ is a non-empty VOQ.*

Definition 2.3 *A priority scheme π defines for every matching phase m a partial order relation π_m on the active $VOQs$.*

We will use the notation $VOQ_{ij} \prec_{\pi_m} VOQ_{kl}$ to denote that VOQ_{ij} has higher priority than VOQ_{kl} during matching phase m. We will also use the notation $VOQ_{ij} \not\prec_{\pi_m} VOQ_{kl}$ to denote that VOQ_{ij} does not have higher priority than VOQ_{kl} during matching phase m.

Note that since π_m is a partial order relation, two VOQs might be unordered by π_m. In order for this to cleanly reflect the notion of equal priority, we define a well-behaved priority scheme as follows:

Definition 2.4 *A well-behaved priority scheme π is a priority scheme such that for every matching phase m, if VOQ_{ij} and VOQ_{kl} are unordered by π_m, and VOQ_{kl} and VOQ_{mn} are unordered by π_m, then VOQ_{ij} and VOQ_{mn} are unordered by π_m.*

The above condition on the priority scheme reflects the notion of equal priority. Hence if during a particular matching phase, VOQ_{ij} and VOQ_{kl} have equal priority, and VOQ_{kl} and VOQ_{mn} have equal priority, then VOQ_{ij} and VOQ_{mn} will have equal priority. This condition defines an equivalence relation on the VOQs which will help

us later to explicitly extend the partial order relation to a total order relation by which all VOQs are ordered.

In practice, a priority switching algorithm breaks ties among the VOQs with equal priorities. We will assume that ties are broken using the indices of the ports, and hence we assume the existence of a total order relation on the (i, j) pairs which is used for breaking ties. Adopting the assumption that breaking a tie among two VOQs involves only the two VOQs in question and no other information, this is the most general deterministic way of breaking ties, since anything else that is more sophisticated can be incorporated into the priority scheme itself. The definition below captures the idea.

Definition 2.5 *Let π be a well-behaved priority scheme and ϕ be a total order relation on the (i, j) pairs. We define the ϕ extension of π to be the priority scheme π^ϕ as follows: For any matching phase m, if $VOQ_{ij} \prec_{\pi_m} VOQ_{kl}$, then $VOQ_{ij} \prec_{\pi_m^\phi} VOQ_{kl}$. For any matching phase m, if VOQ_{ij} and VOQ_{kl} are unordered by π_m, then $VOQ_{ij} \prec_{\pi_m^\phi} VOQ_{kl}$ iff $(i, j) \prec_\phi (k, l)$.*

It can be shown that if π is a well-behaved priority scheme, then π^ϕ is a priority scheme such that for every matching phase m, π_m^ϕ orders all active VOQs. The fact that π is well-behaved means that π_m induces the equal priority equivalence relation on the active VOQs. This in turn implies that we can extend π as described above without violating the property of an order relation. We omit the proof of this fact.

Note that our definition of a priority scheme is general enough to tolerate changing the definition of the partial order relation in every matching phase. Therefore, it is possible to prioritize the VOQs based on their lengths in one matching phase, and based on the age of their HOL packets in another.

Recall that a priority switching algorithm computes its matchings based on the given priority scheme (see Figure 1-6). We now define, for a given priority scheme π, a matching that describes the outcome of a priority switching algorithm.

Definition 2.6 *For a given priority scheme π, a matching computed in matching phase m is π-stable iff it satisfies the following condition: if an active VOQ_{ij} is not*

served by the matching, then either an active VOQ_{ik} is served by the matching and $VOQ_{ij} \not\prec_{\pi_m} VOQ_{ik}$, or an active VOQ_{kj} is served by the matching and $VOQ_{ij} \not\prec_{\pi_m} VOQ_{kj}$.

The notion of a π-*stable* matching is more general than the process depicted in Figure 1-6 by which a priority switching algorithm for priority scheme π computes its matchings. In other terms, a priority switching algorithm for the priority scheme π will always compute a π-*stable* matching. Although the most intuitive and straight forward way of computing a π-*stable* matching is as depicted in Figure 1-6, Definition 2.6 does not impose any restriction on how the π-*stable* matching is computed.

In the next section, we prove lower bound results on the speedup under an α-*shaped* traffic.

2.3 Lower Bounds

We will start by stating, without proof, the following simple lemma:

Lemma 2.1 *If an event E occurs every $\tau \neq 0$ time units, then the number of times $E_{[t_1,t_2]}$ the event occurs in the interval $[t_1, t_2]$, satisfies the following:*

$$\frac{t_2 - t_1}{\tau} - 1 < E_{[t_1,t_2]} \leq \frac{t_2 - t_1}{\tau} + 1$$

We will later use this lemma to argue a lower bound on the number of packets arriving from a particular input i during an interval of time, and an upper bound on the number of matching phases during the same interval of time. Using these bounds, we will prove our different results by showing that the number of packets arriving to the switch at a particular input is more than the number of times that input is served by the matching phases. In order to obtain such a scenario for a given algorithm, we make use of an adversary. The adversary will supply the switch (the algorithm) with an α-*shaped* traffic that will force the algorithm to fail in achieving weak throughput unless the speedup is high enough.

We will denote by a matching policy a switching algorithm that computes a matching that satisfies the policy in every matching phase. For instance, a π-*stable* matching

policy denotes a priority switching algorithm for the priority scheme π. We will also use loosely the notion of *reducibility*. For instance, when we say that weak throughput is not reducible to some matching policy, we mean that a switching algorithm that computes a matching that satisfies the matching policy in every matching phase, does not necessarily imply weak throughput. As an example, a matching policy could be merely any maximal matching with no other conditions on the matching. Therefore, if we say that weak throughput is not reducible to a maximal matching policy, we mean that an algorithm that computes a maximal matching in every matching phase does not necessarily imply weak throughput.

2.3.1 Output Priority Switching Algorithms

In this section, we establish a lower bound on the speedup for a class of priority switching algorithms that employ an output priority scheme defined below:

Definition 2.7 *An output priority scheme π is a priority scheme that satisfies the following: for every matching phase m, there exists a partial order relation π'_m on the output ports such that $VOQ_{ij} \prec_{\pi_m} VOQ_{kl}$ iff $j \prec_{\pi'_m} l$.*

Note that according to this definition, VOQ_{ij} and VOQ_{kj} are unordered by an output priority scheme (because $j \not\prec_{\pi'_m} j$ for any matching phase m), reflecting the fact that neither has priority over the other because they share the same output. An example of an output priority scheme is *lowest output occupancy* where a VOQ_{ij} for which there are less packets in output queue j has higher priority. This scheme was used in *LOOFA* [18].

Below we describe the first adversary that we are going to use:

The ϕ-Adversary:

Let ϕ be any total order relation on the (i, j) pairs. We will assume, without loss of generality, that $(1, 1)$ is the highest ranked according to ϕ. Similarly, after discarding $(1, k)$ and $(k, 1)$ for all $k = 1...N$, we assume that $(2, 2)$ has the highest rank according to ϕ among the remaining pairs. We continue until we obtain pairs

$(3,3)...(N-1,N-1)$ in the same way. The adversary produces an α-*shaped* traffic as shown in Figure 2-1.

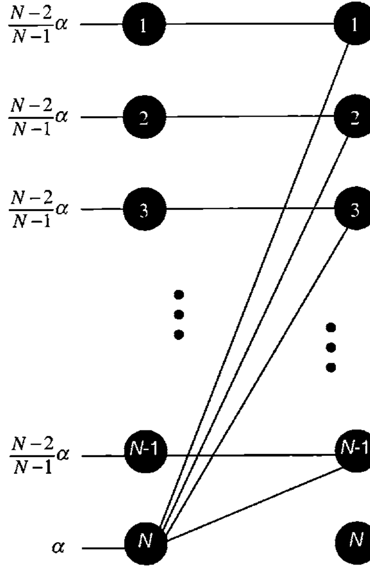


Figure 2-1: The ϕ-Adversary

At input N, the flow of rate α is divided equally among the $N-1$ outputs in a round robin fashion. The adversary produces a packet at input N every $\frac{1}{\alpha}$ time units. Similarly, the adversary produces a packet at input i, where $i = 1...N-1$, every $\frac{N-2}{N-1}\alpha$ time units. It can be shown that this traffic is α-*shaped.* More precisely, using Lemma 2.1 and the fact that the adversary uses a round robin order to generate packets for the first $N-1$ outputs, we can show that during any time interval $[t_1, t_2]$, the number of packets from input N to any of the first $N-1$ outputs satisfies the following:

$$\lceil\frac{\alpha T-1}{N-1}\rceil \leq A_{Nj}(t_2) - A_{Nj}(t_1) \leq \lceil\frac{\alpha T+1}{N-1}\rceil$$

where $T = t_2 - t_1$. This confirms with the first condition of an α-*shaped* traffic. The condition is also true for all other flows. We can show that the rest of the conditions are also satisfied. Note also that no overloading occurs since at any port, the sum of the rates of all flows is at most:

$$\frac{\alpha}{N-1} + \frac{N-2}{N-1}\alpha = \alpha$$

Lemma 2.2 *For any well-behaved output priority scheme π and any total order relation ϕ on the (i,j) pairs, a π^ϕ-stable matching policy, under the ϕ-Adversary, cannot serve inputs 1 and N during the same matching phase.*

Proof: By the property of an output priority scheme π, for any matching phase m, VOQ_{jj} and VOQ_{Nj} are unordered by π_m. Therefore, we have that $VOQ_{jj} \prec_{\pi_m^\phi} VOQ_{Nj}$ for any matching phase m, by the definition of the ϕ-Adversary. Hence, the π^ϕ-*stable* matching policy will choose the matching $\{(1,1),(2,2),...,(N-1,N-1)\}$ whenever possible. Since the ϕ-Adversary provides the same traffic for flows $(1,1)$, $(2,2)$, ..., and $(N-1,N-1)$, the matching policy will always be able to pick the corresponding edges together. In other words, it is not possible that VOQ_{ii} is active and VOQ_{jj} is not for $i,j = 1...N-1$. As a result, inputs 1 and N cannot be served during the same matching phase. ∎

Theorem 2.1 *For any well-behaved output priority scheme π and any total order relation ϕ on the (i,j) pairs, a π^ϕ-stable matching policy cannot achieve weak throughput under an α-shaped traffic unless $S \geq 2\alpha$.*

Proof: Consider the ϕ-Adversary. Pick a time t. By Lemma 2.1, we have at most $tS+1$ matching phases by time t, each of which is forwarding at most one packet from inputs 1 and N by Lemma 2.2. By Lemma 2.1, the number of packets arriving to input 1 and N by time t is at least:

$$\alpha t - 1 + \frac{N-2}{N-1}\alpha t - 1$$

Therefore, at time t, the number of packets remaining at inputs 1 and N is at least:

$$(\frac{2N-3}{N-1}\alpha - S)t - 3$$

For $S < 2\alpha$, there exists a large enough N, say N_0, such that $\frac{2N_0-3}{N_0-1}\alpha - S = \delta > 0$. If weak throughput is to be achieved, then for every $\epsilon > 0$, there must exist a large enough t, say t_0, such that for every VOQ_{ij}, $\frac{X_{ij}(t)}{t} \leq \epsilon$ for any $t \geq t_0$. Assume that

weak throughput is achieved and let $\epsilon < \frac{\delta}{N_0}$ and t_0 be as defined above. Let $t \geq t_0$ be such that $\frac{\delta}{N_0} - \frac{3}{N_0 t} > \epsilon$. Since at inputs 1 and N_0 we have $1 + (N_0 - 1) = N_0$ active VOQs, there exists a VOQ_{ij} such that the number of packets remaining in VOQ_{ij} at time t is at least $\frac{\delta t - 3}{N_0}$. Therefore,

$$\frac{X_{ij}(t)}{t} \geq \frac{\delta}{N_0} - \frac{3}{N_0 t} > \epsilon$$

Since $t \geq t_0$, we have a contradiction. ■

We have proved that any switching algorithm based on an output priority scheme that breaks ties using the indices of the ports cannot achieve weak throughput under an *α-shaped* traffic unless $S \geq 2$. The implication of this result is that a speedup of at least 2 is required for an output priority switching algorithm to provide throughput with a full loading of the switch. Below we prove a corollary.

Corollary 2.1 *For any output priority scheme π, weak throughput is not reducible to a π-stable matching policy unless $S \geq 2$.*

Proof: There exists an output priority scheme π' such that for any matching phase m, π'_m is a total order relation on active VOQs. Hence, π' is a well-behaved output priority scheme. Note that $\pi'^{\phi} = \pi'$ for any total order relation ϕ on the (i, j) pairs. Moreover, $VOQ_{ij} \prec_{\pi_m} VOQ_{kl} \Rightarrow VOQ_{ij} \prec_{\pi'_m} VOQ_{kl}$. Therefore, since a *$\pi'$-stable* matching policy is a *π-stable* matching policy, the result is immediate from Theorem 2.1 using $\alpha = 1$. ■

The basic version of *LOOFA*, described in [18], considers first the VOQs with the lower output queue occupancy as candidates for the matching. As a consequence, it only guarantees that some *π-stable* matching policy will be used, where π is the *lowest output occupancy* priority scheme. Therefore, we proved that this switching algorithm does not imply weak throughput for $S < 2$. *LOOFA* assumes that at most one packet arrives to any input per time unit. The ϕ-Adversary satisfies this condition (see Figure 2-1).

2.3.2 Maximum Size Matching

Consider the switching algorithm that computes a maximum size matching in every matching phase. N. McKeown et al. proved in [21] that such an algorithm, with probability 1, will not achieve weak throughput unless $S \geq 1.037\alpha$, when arrivals to the switch are i.i.d. Bernoulli arrivals and a random maximum size matching is computed. We are going to consider the lower bound on S when this switching algorithm is deterministic. Consider the ϕ-Adversary described earlier. Note that for any priority scheme π and any total order relation ϕ on the (i,j) pairs, a π^{ϕ}-*stable* matching policy is a maximum size matching policy under the ϕ-Adversary. To see this, note that the maximum possible size for a matching is $N-1$ when the first $N-1$ outputs are matched. Note also that, whenever possible, the π^{ϕ}-*stable* matching policy will choose the matching $\{(1,1),(2,2),...,(N-1,N-1)\}$ where VOQ_{ii} for $i = 1...N-1$ are either active together or non of them is. As a consequence, we have the following result:

Corollary 2.2 *Weak throughput is not reducible to a maximum size matching policy unless* $S \geq 2$.

Proof: Immediate from Theorem 2.1 using $\alpha = 1$ since, as argued above, for any priority scheme π and any total order relation ϕ on the (i,j) pairs, under the ϕ-Adversary, a π^{ϕ}-*stable* matching policy is a maximum size matching policy. ■

2.3.3 Maximal Matching

Since a maximum size matching is also a maximal matching, we have the following result:

Corollary 2.3 *Weak throughput is not reducible to a maximal matching policy unless* $S \geq 2$.

Proof: Immediate from Corollary 2.1 ■

In a recent paper [8], Dai et al. proved that with $S \geq 2$, any maximal matching policy guarantees weak throughput with probability 1 under an SLLN traffic. We just proved that this is not true when $S < 2$. Therefore, since both a *π-stable* matching and a maximum size matching are maximal matchings, the lower bound results obtained so far are tight.

Charny et al. proved in [6] that a delay guarantee, and therefore strong throughput also, is reducible to a maximal matching policy if $S > 4$ under any weak constant burst traffic. It can be shown that strong throughput is reducible to a maximal matching policy if $S = 4$ under a strong constant burst traffic. The question of achieving strong throughput with any maximal matching policy under constant burst traffic models for $S \in [2, 4]$ remains to be answered.

2.3.4 Input Priority Switching Algorithms

In this section, we will prove a lower bound on the speedup for another class of priority switching algorithms that use input priority.

We can define an input priority scheme in a similar way to the output priority scheme by reversing the role of input and output ports, and hence obtaining the same results above. However, we choose to define an input priority scheme more intelligently to take into account the input and output ports of each packet.

Before we do so, we introduce a definition of the state of a VOQ.

Definition 2.8 *For a matching phase m, let A_{ijm} be a function of time such that $A_{ijm}(t) = A_{ij}(t)$ if $t \in [0, \frac{m}{S}]$, and $A_{ijm}(t) = A_{ij}(\frac{m}{S})$ otherwise. Similarly, let D_{ijm} be a function of time such that $D_{ijm}(t) = D_{ij}(t)$ if $t \in [0, \frac{m}{S}]$, and $D_{ijm}(t) = D_{ij}(\frac{m}{S})$ otherwise. The state of a VOQ_{ij} during matching phase m, S_{ijm}, is the tuple (A_{ijm}, D_{ijm}).*

In other terms, the state of VOQ_{ij} during matching phase m is the history of packet arrivals and departures to and from VOQ_{ij} up to the beginning of matching phase m.

Definition 2.9 *An input priority scheme π is a priority scheme that satisfies the*

following: for every matching phase m, there exists a partial order relation π' on the states of the VOQs such that $VOQ_{ij} \prec_{\pi_m} VOQ_{kl}$ iff $S_{ijm} \prec_{\pi'_m} S_{klm}$.

According to the definition of an input priority scheme, VOQs with equal VOQ states are unordered and therefore have equal priority. An example of an input priority scheme is *largest queue length* where VOQs with more packets have more priority. This scheme was used in the *Central Queue* algorithm [16] as mentioned earlier. Another example is *oldest packet* where the VOQs with the older HOL packets have more priority. This scheme was used in the *Oldest Cell First* algorithm [6] as mentioned earlier.

Next we will prove a similar lower bound result for the speedup required by priority switching algorithms with an input priority scheme. Before we do so, we start with few definitions and lemmas.

Definition 2.10 *A ϕ-ordered $K_{N,N}$ is an $N \times N$ complete bipartite graph with a total order relation ϕ on its edges.*

Definition 2.11 *In a ϕ-ordered $K_{N,N}$, an ℓ-symmetric cycle is a cycle n_1, n_2, ..., $n_{2\ell}$, n_1 that satisfies the following: $(n_{i-1}, n_i) \prec_\phi (n_i, n_{i+1})$ iff $(n_{i-1+\ell}, n_{i+\ell}) \prec_\phi (n_{i+\ell}, n_{i+1+\ell})$ for $i = 1...\ell$, where n_0 is the same as $n_{2\ell}$ and $n_{2\ell+1}$ is the same as n_1.*

Lemma 2.3 *For any $\ell > 2$ and a large enough N, any ϕ-ordered $K_{N,N}$ contains an ℓ-symmetric cycle.*

Proof: Let L and R be the two disjoint sets of nodes of $K_{N,N}$. Consider the bipartite graph induced by any $k_L = (\ell-1)\ell!+1$ nodes in L and any $k_R = (\ell-1)k_L!+1$ nodes in R. Let U and V be the two disjoint sets of nodes of the new bipartite graph. Every node v in V orders the k_L nodes of U according to the order of their respective edges to node v. Since there are at most $k_L!$ possible orders, we can find at least ℓ nodes in V that define the same order ϕ_u on U. Let these nodes be v_1, v_2, ..., v_ℓ and let V_ℓ be the set $\{v_1, v_2, ..., v_\ell\}$. Now every node u in U orders the ℓ nodes of V_ℓ according to the order of their respective edges to node u. Since there are at most $\ell!$

possible orders, we can find at least ℓ nodes in U that define the same order ϕ_v on V_ℓ. Let these nodes be u_1, u_2, ..., u_ℓ and let U_ℓ be the set $\{u_1, u_2, ..., u_\ell\}$. Therefore, we obtain two ordered sets U_ℓ and V_ℓ that satisfy the following properties:

$$(u_i, v) \prec_\phi (u_j, v) \text{ iff } u_i \prec_{\phi_u} u_j \ \forall u_i, u_j \in U_\ell, \forall v \in V_\ell$$
$$(u, v_i) \prec_\phi (u, v_j) \text{ iff } v_i \prec_{\phi_v} v_j \ \forall u \in U_\ell, \forall v_i, v_j \in V_\ell$$

Without loss of generality, let u_1, u_2, ..., u_ℓ be the ordered elements of U_ℓ and let v_1, v_2, ..., v_ℓ be the ordered elements of V_ℓ. We can verify that the two cycles of Figure 2-2 are ℓ-*symmetric* cycles. ■

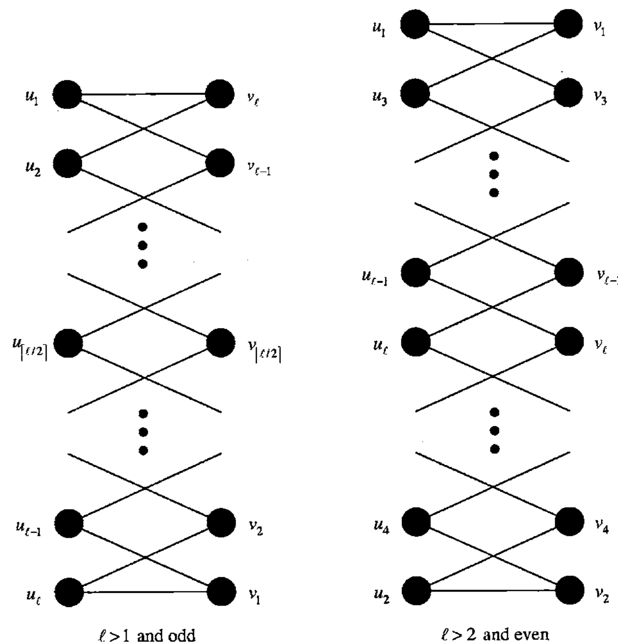


Figure 2-2: ℓ-*symmetric* cycles

We define the following adversary:

The 3-*symmetric* ϕ-Adversary:

Consider a 3-*symmetric* cycle in the ϕ-*ordered* $K_{N,N}$ as shown in Figure 2-3. The adversary generates a packet for every flow shown in Figure 2-3 every $\frac{2}{\alpha}$ time units producing an α-*shaped* traffic.

Note that the traffic of the 3-*symmetric* ϕ-Adversary is a theoretic one where two packets from the same input can arrive to the switch simultaneously. This is possible

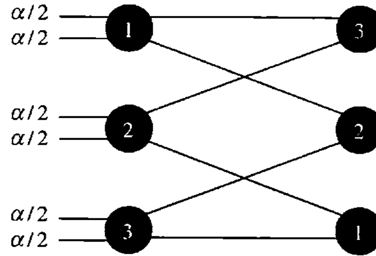


Figure 2-3: The 3-*symmetric* ϕ-Adversary

if the *VOQ*s at an input can be accessed independently. For instance, each *VOQ* is a physically separate queue. In any case, the use of such a theoretic traffic can be justified by the following reasoning: in practice, time is discretized to fixed size intervals of length δ, and hence as long as packets arrive during the same interval, they will have the same time-stamp. Therefore, we can consider a discrete version of the 3-*symmetric* ϕ-Adversary. The discrete adversary will write the two packets in parallel to the memory of the input queued switch by writing a bit of each in an alternating fashion. Since any two write operations to the memory will complete in a particular order, specifically the last two write operations, we still have that one packet will arrive before the other. However, we can prove that for any rational $\alpha\delta$, there exists a line speed (equivalently, a packet size) beyond which any two simultaneous packets in the 3-*symmetric* ϕ-Adversary will arrive during the same interval of length δ in the discrete adversary. Similarly, we can prove that for any rational $\frac{\alpha}{S}$, there exists a line speed beyond which any two simultaneous packets in the 3-*symmetric* ϕ-Adversary cannot straddle the beginning of a matching phase in the discrete adversary. Therefore, if $\alpha\delta$ and $\frac{\alpha}{S}$ are both rational, there exists a line speed beyond which any two simultaneous packets in the 3-*symmetric* ϕ-Adversary will appear to arrive simultaneously in the discrete adversary. The proof of this fact relies on the following two lemmas:

Lemma 2.4 *If an event E_1 occurs every $\tau_1 \neq 0$ time units and an event E_2 occurs every $\tau_2 \neq 0$ time units and $\frac{\tau_1}{\tau_2}$ is rational, then there exists $0 < \epsilon \leq \tau_2$ such that for any time t, if event E_1 occurs at time t, event E_2 cannot occur in the interval $(t, t+\epsilon)$.*

Proof: Since $\frac{\tau_1}{\tau_2}$ is rational, there exist two integers a and b such that $\frac{\tau_1}{\tau_2} = \frac{a}{b}$. Define $Y = \{y \text{ is an integer} \mid 0 \leq y < b \text{ and } \lceil \frac{a}{b}y \rceil - \frac{a}{b}y > 0\}$. If $Y = \emptyset$, define $\epsilon \leq \tau_2$. Otherwise, if $Y \neq \emptyset$, define $\epsilon \leq \min_{y \in Y}(\lceil \frac{a}{b}y \rceil - \frac{a}{b}y)\tau_2$.

Assume event E_1 occurs at time t_1 for the mth time and event E_2 occurs at time t_2 for the nth time. This means that $t_1 = m\tau_1$ and $t_2 = n\tau_2$, where m and n are both integers. Therefore,

$$t_2 - t_1 = n\tau_2 - m\tau_1 = (n - \frac{\tau_1}{\tau_2}m)\tau_2 = (n - \frac{a}{b}m)\tau_2$$

We can express m as $m = xb + y$ where x and y are integers such that $x \geq 0$ and $0 \leq y < b$. Therefore,

$$t_2 - t_1 = (n - ax - \frac{a}{b}y)\tau_2$$

We are interested in the case where $t_2 - t_1 \geq 0$, so assume without loss of generality that $n - ax - \frac{a}{b}y \geq 0$.

We distinguish between two cases. If $\frac{a}{b}y$ is an integer, then $t_2 - t_1 \in \{0, \tau_2, 2\tau_2, ...\}$ and the lemma is true since $0 < \epsilon \leq \tau_2$. If $\frac{a}{b}y$ is not an integer (and hence $Y \neq \emptyset$), then

$$t_2 - t_1 = (n - ax - \frac{a}{b}y)\tau_2 \geq (\lceil \frac{a}{b}y \rceil - \frac{a}{b}y)\tau_2 \geq \min_{y \in Y}(\lceil \frac{a}{b}y \rceil - \frac{a}{b}y)\tau_2 = \epsilon$$

and the lemma is true. ■

Lemma 2.5 *Let ϵ be as defined in Lemma 2.4. If event E_1 is delayed by ϵ' time units such that $0 < \epsilon' < \epsilon$, then if event E_1 occurs at time t, event E_2 cannot occur in the interval $[t, t + \epsilon - \epsilon')$.*

Proof: From Lemma 2.4, we know that if event E_1 occurs at time t, then event E_2 cannot occur in the interval $(t, t+\epsilon)$. It is possible, however, for event E_2 to occur at time t. Therefore, delaying event E_1 ϵ' time units such that $0 < \epsilon' < \epsilon$, guarantees that if event E_1 occurs at time t, event E_2 cannot occur in the interval $[t, t + \epsilon - \epsilon')$.

■

Now back to our previous argument. Let E_1 be the event that the first packet arrives and let E_2 be the event that a matching phase begins. From Lemma 2.4, there exists an ϵ such that if the first packet arrives at time t, a matching phase cannot begin in $(t, t+\epsilon)$. Furthermore, the adversary can delay the arrival of packets by $\epsilon' < \epsilon$ time units, and by Lemma 2.5, we get that if the first packet arrives at time t, a matching phase cannot begin in $[t, t+\epsilon-\epsilon')$. We can set the line speed (equivalently, the packet size) such that the last bit of a packet can be written to the memory of the input queued switch in less than $\epsilon - \epsilon'$ time units. Therefore, since the last bit of a packet requires less than $\epsilon - \epsilon'$ time to be written, and both packets are written in parallel, when the first packet arrives at time t, the next packet will arrive before the matching phase begins. As a result, when the matching phase begins, both packets are present. A similar argument can be made to prove that both packets arrive during the same discrete interval of length δ.

Next we prove a lemma similar to Lemma 2.2 for the case of the 3-*symmetric* ϕ-Adversary.

Lemma 2.6 *For any well-behaved input priority scheme π and any total order relation ϕ on the (i,j) pairs, a π^{ϕ}-stable matching policy, under the 3-symmetric ϕ-Adversary, serves at most 2 packets in each matching phase.*

Proof: We will prove that if there are packets at the input side during a matching phase, the π^{ϕ}-*stable* matching policy will choose one of the following matchings: $\{(1,3),(3,1)\}$, $\{(1,2),(2,1)\}$, or $\{(2,3),(3,2)\}$. We will prove this by induction on the number of matching phases:

Base case: The claim in trivially true at a fictitious matching phase before the beginning of the first matching phase.

Inductive step: We assume that the claim is true up to matching phase $m-1$. We need to prove that it remains true for matching phase m. First, we denote by (i,j) and (j,i) two edges belonging to one of the above three matchings. Since the claim is true up to matching phase $m-1$ and the adversary assigns the same traffic to all flows, VOQ_{ij} and VOQ_{ji} will have the same state by the beginning of matching

phase m. Secondly, we can see from Figure 2-3 that if (k,l) is adjacent to (i,j) i.e. either $i=k$ or $j=l$, then (l,k) is adjacent to (j,i); moreover, by the property of the 3-*symmetric* cycle we have: $(i,j) \prec_\phi (k,l)$ iff $(j,i) \prec_\phi (l,k)$.

If there are no packets at the input side during matching phase m, then we are done. Otherwise, let (i,j) be the edge in the graph such that there is no other edge (k,l) in the graph that satisfies $(k,l) \prec_{\pi^\phi} (i,j)$. Therefore, by the property of the π^ϕ-*stable* matching, (i,j) will be in the matching during matching phase m. We will prove that (j,i) is also in the matching.

Consider an edge (l,k) in the graph during matching phase m that is adjacent to (j,i). By equality of VOQ states, we know that (k,l) is in the graph during matching phase m (VOQ_{lk} active implies VOQ_{kl} active). We also know that (k,l) is adjacent to (i,j).

case 1: If $VOQ_{ij} \prec_{\pi_m} VOQ_{kl}$, then by equality of VOQ states, $VOQ_{ji} \prec_{\pi_m} VOQ_{lk}$.

case 2: Otherwise, it must be that $VOQ_{ij} \not\prec_{\pi_m} VOQ_{kl}$ and $(i,j) \prec_\phi (k,l)$ by our choice of (i,j). By equality of VOQ states and the property of the 3-*symmetric* cycle, $VOQ_{ij} \not\prec_{\pi_m} VOQ_{lk}$ and $(j,i) \prec_\phi (l,k)$.

Therefore, in both cases, $VOQ_{ji} \prec_{\pi_m^\phi} VOQ_{lk}$ for any active VOQ_{lk} such that (l,k) is adjacent to (j,i), and hence (j,i) is in the matching during matching phase m by the property of the π^ϕ-*stable* matching. ■

Theorem 2.2 *For any well-behaved input priority scheme π and any total order relation ϕ on the (i,j) pairs, a π^ϕ-stable matching policy cannot achieve weak throughput under an α-shaped traffic unless $S > \frac{3}{2}\alpha$.*

Proof: We will use the 3-*symmetric* ϕ-Adversary. Consider a time t. By Lemma 2.1, we have at most $tS+1$ matching phases by time t, each of which forwards at most 2 packets by Lemma 2.6. Therefore, the number of packets forwarded by time t is at most $2(tS+1)$. By Lemma 2.1, the number of packets arrived by time t to the switch is at least:

$$6(\frac{\alpha}{2}t-1)$$

Therefore, at time t, the number of packets remaining at the inputs is at least:

$$(3\alpha - 2S)t - 8$$

For $S < \frac{3}{2}\alpha$, $(3\alpha - 2S) = \delta > 0$. If weak throughput is to be achieve, then for every $\epsilon > 0$, there must exist a large enough t, say t_0, such that for every VOQ_{ij}, $\frac{X_{ij}(t)}{t} \leq \epsilon$ for any $t \geq t_0$. Assume that weak throughput is achieved and let $\epsilon < \frac{\delta}{6}$ and t_0 be as defined above. Let $t \geq t_0$ be such that $\frac{\delta}{6} - \frac{8}{6t} > \epsilon$. Since at the inputs we have 6 active VOQs, there exists a VOQ_{ij} such that the number of packets remaining in VOQ_{ij} at time t is at least $\frac{\delta t - 8}{6}$. Therefore,

$$\frac{X_{ij}(t)}{t} \geq \frac{\delta}{6} - \frac{8}{6t} > \epsilon$$

Since $t \geq t_0$, we have a contradiction. ■

We have proved that any switching algorithm based on an input priority scheme that breaks ties using the indices of the ports cannot achieve weak throughput under an *α-shaped* traffic unless $S > \frac{3}{2}\alpha$. For instance, the *Central Queue* and the *Oldest Cell First* switching algorithms cannot achieve weak throughput unless $S \geq \frac{3}{2}$, under the assumption that indices of the input and output ports are used to break ties when two VOQs have the same priority (i.e. same VOQ length and same age of HOL packet respectively). We can prove that the *Oldest Cell First* switching algorithm cannot achieve weak throughput even without the above tie breaking assumption, by adding a minor change to the adversary. The adversary will keep the same traffic for flows $\{(1,3),(3,1)\}$; however, it will delay the traffic for flows $\{(1,2),(2,1)\}$ by one time unit, and it will delay the traffic for flows $\{(2,3),(3,2)\}$ by two time units. In that case, the *Oldest Cell First* algorithm will have to choose matchings in a way similar to before, forwarding only two packets per matching phase, because the VOQs that will have the oldest HOL packets are the ones that belong to one of the three matchings listed above.

Theorem 2.2 suggests that a speedup of at least $\frac{3}{2}$ is required for an input priority switching algorithm to provide throughput with a full loading of the switch.

Below we prove a corollary.

Corollary 2.4 *For any input priority scheme π, weak throughput is not reducible to a π-stable matching policy unless $S > \frac{3}{2}$.*

Proof: There exists an input priority scheme π' such that for any matching phase m, π'_m is a total order relation on active VOQs. Hence, π' is a well-behaved input priority scheme. Note that $\pi'^{\phi} = \pi'$ for any total order relation ϕ on the (i, j) pairs. Moreover, $VOQ_{ij} \prec_{\pi_m} VOQ_{kl} \Rightarrow VOQ_{ij} \prec_{\pi'_m} VOQ_{kl}$. Therefore, since a π'-*stable* matching policy is a π-*stable* matching policy, the result is immediate from Theorem 2.2 using $\alpha = 1$. ■

Now we discuss the enhanced version of the *LOOFA* algorithm presented in [18] which uses a combined input-output priority scheme. Although *LOOFA* assumes that only one packet can arrive to an input port per time unit (which is not true with the 3-*symmetric* ϕ-Adversary), we will show that the priority scheme of *LOOFA* does not imply weak throughput under an α-*shaped* traffic if $S < \frac{3}{2}$.

LOOFA computes a matching in the following way: it finds the port with the smallest output queue and selects an input to match it with, then repeats until the matching is maximal. In the deterministic version of *LOOFA*, the input selection criterion can be either the input with the oldest HOL packet, or it can be performed in a round robin fashion. We can show that this combined input-output priority scheme also suffers the same limitations. We will not go into the details, we will just illustrate a sketch of the proof.

Consider the example of Figure 2-3. Let $S < \frac{3}{2}\alpha$, which means that $\frac{2}{\alpha} < \frac{3}{S}$. Since every VOQ accumulates packets at a rate $\frac{\alpha}{2}$, every three matching phases, a VOQ will receive a new packet. This means that a policy can continuously select the following matchings in that order:

$$\{(1, 3), (3, 1)\}, \{(1, 2), (2, 1)\}, \{(2, 3), (3, 2)\}$$

Once can show, irrespective of the speedup of the switch, that this order in choosing the matching satisfies the smallest output queue criterion, assuming that forwarded

packets arrive at the same time to their output queues and that output queues are served as soon as possible (there is no restriction on the order in which packets are delivered at the output). This assumption can be justified by an adversary that controls the timing of the algorithm. Moreover, this order in choosing the matching satisfies three selection criteria: *round robin*, *oldest packet*, and *largest length.* Since a matching of size two is computed in every matching phase, weak throughput cannot be achieved as proved in Theorem 2.2.

2.4 Summary

We proved lower bounds on the speedup required by several classes of switching algorithms to achieve weak throughput. By doing so, we showed that most of the practical switching algorithms suffer the same theoretical limitation, which is the fact that speedup cannot be avoided for throughput to be guaranteed. An algorithm based on a Birkhoff-von Neumann decomposition of the rate matrix that provides a delay guarantee with no speedup under a strong constant burst traffic has been suggested in [4]. This algorithm, however, requires an explicit knowledge of the rates λ_{ij}s and is therefore sensitive to the values of the λ_{ij}s. Moreover, it requires a pre-processing step of $O(N^{4.5})$ time complexity (but it runs after that in $O(\log N)$ time). Therefore, given that speedup cannot be avoided practically, we consider in Chapter 5 the use of multiple switches with no speedup in parallel in order to employ some of the practical switching algorithms, while eliminating the speedup requirement they impose on the switch.

Chapter 3

Two Priority Switching Algorithms

In this chapter, we present two simple priority switching algorithms. The general operation of a priority switching algorithm is depicted in Figure 1-6. In principle, a priority switching algorithm prioritizes the VOQs and computes a matching based on the priority scheme in the following way: it chooses the highest priority non-empty VOQ_{ij} and adds (i, j) to the matching if possible. Then it discards VOQ_{ij} and repeats until a maximal matching is obtained. Therefore, to fully describe a priority switching algorithm, it suffices to determine what the priority scheme is. We are going to present two priority schemes called *Earliest Activation Time* and *Latest Activation Time*. For this, we need to recall the definition of an active VOQ as stated in Definition 2.2. Basically, an active VOQ is a non-empty VOQ. We define the activation time of a VOQ as follows:

Definition 3.1 (activation time) *The activation time of a VOQ is the last time at which the VOQ transitioned from being inactive to active.*

Recall the formal definition of a priority scheme from Chapter 2 (Definition 2.3). We define the two priority schemes that we mentioned above as follows:

Definition 3.2 (earliest activation time) *If π is the Earliest Activation Time priority scheme, then for every matching phase m, $VOQ_{ij} \prec_{\pi_m} VOQ_{kl}$ iff VOQ_{ij} has an earlier activation time than VOQ_{kl}.*

Definition 3.3 (latest activation time) *If π is the Latest Activation Time priority scheme, then for every matching phase m, $VOQ_{ij} \prec_{\pi_m} VOQ_{kl}$ iff VOQ_{ij} has a later activation time then VOQ_{kl}.*

The motivation behind the above priority schemes is to reduce the state information needed by the switching algorithm to compute the priorities. As we have seen in the previous chapter, most of the priority switching algorithms suggested in the literature require a considerable amount of state information. For instance, *Oldest Cell First* and *LOOFA* require packets to be tagged by their arrival times because their priority schemes rely on the age of packets. Moreover, the *Central Queue* algorithm requires to maintain the length of each VOQ. Although this is considerably less than keeping ages of packets, the algorithm requires the traffic to be constantly backlogged in order to achieve a delay guarantee. This means that for every VOQ_{ij}, the number of packets that arrive to VOQ_{ij} by time t, has to satisfy $A_{ij}(t) \geq \lambda_{ij}t - k$, where λ_{ij} is the rate of flow of packets from input i to output j, and k is a constant. The two priority schemes defined above will eliminate the need to maintain the ages of packets as well as the need for the traffic to be constantly backlogged.

3.1 Earliest Activation Time

We start with the *Earliest Activation Time* switching algorithm. We will prove that it provides, under a strong constant burst traffic, strong throughput with a speedup $S = 2$ and a delay guarantee with a speedup $S > 2$. In the results below, we state S as a function of α which is the loading of the switch. Setting $\alpha = 1$ gives us the claims above.

Theorem 3.1 *Under a strong constant burst traffic, the Earliest Activation Time switching algorithm achieves strong throughput with a speedup $S = 2\alpha$, where α is the loading of the switch.*

Proof: We will prove that the length of every VOQ is bounded. For a VOQ_{ij}, if $\lambda_{ij} = 0$, then by the definition of a strong constant burst traffic, $A_{ij}(t) \leq \lambda_{ij}t + B = B$

for any time t. Therefore, the length of VOQ_{ij} cannot exceed B. So let us assume that $\lambda_{ij} \neq 0$. We will prove that VOQ_{ij} cannot remain active for more than a bounded time D. Consider the VOQ_{ij} with $\lambda_{ij} \neq 0$ that is the first to remain active for a time D (if more than one VOQ satisfy the property, we choose one arbitrarily). Therefore, if VOQ_{ij} became active at time t, it will remain active during $[t, t+D]$. Recall that the switching algorithm will compute a π-*stable* matching (Definition 2.6) in each matching phase, where π is the *Earliest Activation Time* priority scheme. As a result, in every matching phase m during $[t, t+D]$, either VOQ_{ij} is served, or an active VOQ_{ik} is served and $VOQ_{ij} \not\prec_{\pi_m} VOQ_{ik}$ for some k, or an active VOQ_{kj} is served and $VOQ_{ij} \not\prec_{\pi_m} VOQ_{kj}$ for some k. In other terms, either a packet from VOQ_{ij} is forwarded, or a packet from VOQ_{ik} with an activation time no later than VOQ_{ij} is forwarded, or a packet from VOQ_{kj} with an activation time no later than VOQ_{ij} is forwarded. Therefore, by Lemma 2.1, at least $SD-1$ packets that satisfy the above criterion were forwarded from input i or to output j during $[t, t+D]$. Moreover, since by the choice of VOQ_{ij}, at time $t+D$, all active VOQs have been active for at most a time D, the number of these packets can be bounded as follows:

$$\lambda_{ij}D + B + \sum_{k \neq j}(\lambda_{ik}D + B) + \sum_{k \neq i}(\lambda_{kj}D + B) \leq (2\alpha - \lambda_{ij})D + (2N-1)B$$

The bound above is obtained by the property of the strong constant burst traffic and by the fact that all VOQs up to time $t+D$ have been active for at most a time D (except possibly for some VOQ_{ik} with $\lambda_{ik} = 0$ or some VOQ_{kj} with $\lambda_{kj} = 0$, which in that case implies that the length of those VOQs is always at most B as argued above). We reach a contradiction if $SD - 1 > (2\alpha - \lambda_{ij})D + (2N-1)B$ or if $D > \frac{(2N-1)B+1}{S-2\alpha+\lambda_{ij}}$. If we define $\lambda_0 = \min_{i,j|\lambda_{ij} \neq 0} \lambda_{ij}$, then $D \leq \frac{(2N-1)B+1}{S-2\alpha+\lambda_0}$. If $S = 2\alpha$, D is at most $\frac{(2N-1)B+1}{\lambda_0}$ and VOQ_{ij} cannot be the first one to remain active for more than D. As a result, the length of VOQ_{ij} cannot exceed $\lambda_{ij}\frac{(2N-1)B+1}{\lambda_0} + B$ by the property of the strong constant burst traffic. ■

Theorem 3.2 *Under a strong constant burst traffic, the Earliest Activation Time switching algorithm achieves a delay bound on every packet with a speedup* $S > 2\alpha$,

where α is the loading of the switch.

Proof: We proved in Theorem 3.1 that VOQ_{ij} cannot remain active for more than a time $D = \frac{(2N-1)B+1}{S-2\alpha}$. Therefore, for any $S > 2\alpha$, D is a well defined bound. As a consequence, a packet cannot remain in its VOQ for more than a time D; otherwise, its VOQ will remain active for more than a time D, a contradiction. We still need to bound the time a packet remains in its output queue (recall that an input queued switch with a speedup $S > 1$ has output queues). Note that for a given output j, no more than $SD + 1$ packets destined to output j can be present at the input side of the switch at any time. The reason behind this fact is that at most $SD + 1$ packets can be forwarded to output j during an interval of time D (Lemma 2.1), and hence if more than $SD + 1$ packets destined to output j are present at the input side at some point in time, at least one packet will remain in its VOQ for more than a time D.

Consider a time interval $[t_1, t_2]$ such that the queue at output j becomes non-empty at time t_1 and remains so during $[t_1, t_2]$. Since output j delivers a packet from its output queue whenever possible (one per time unit at line speed), it delivers at least $t_2 - t_1 - 1$ packets during $[t_1, t_2]$ (again, Lemma 2.1). Therefore, the number of packets in output queue j at time t_2 cannot exceed $SD + 2 + NB$ because at most $\alpha(t_2 - t_1) + NB \leq (t_2 - t_1) + NB$ packets destined to that output can arrive to the switch during $[t_1, t_2]$, by the property of the strong constant burst traffic. Therefore, the number of packets in output queue j cannot exceed $SD + 2 + NB$ at any time. This is true for any output j. Since the packets at the output are delivered in a FIFO manner, a packet cannot remain in its output queue for more than $SD + 2 + NB$ time, resulting in a total delay of $(S + 1)D + 2 + NB$. ■

3.2 Latest Activation Time

The *Earliest Activation Time* switching algorithm achieves strong throughput with a speedup $S = 2$; however, the theoretical bound obtained on the length of a VOQ depends highly on the traffic, namely λ_0 which is the minimum non-zero λ_{ij}. We will eliminate this dependence with the *Latest Activation Time* switching algorithm by

making the length of a VOQ depend on the traffic only through the burst constant B. As before, we express S as a function of the loading of the switch α.

Theorem 3.3 *Under a strong constant burst traffic, the Latest Activation Time switching algorithm achieves strong throughput with a speedup $S = 2\alpha$, where α is the loading of the switch.*

Proof: We will prove that the length of every VOQ is bounded. For a VOQ_{ij}, if $\lambda_{ij} = 0$, then by definition of a strong constant burst traffic, $A_{ij}(t) \leq \lambda_{ij}t + B = B$ for any time t. Therefore, the length of VOQ_{ij} cannot exceed B. So let us assume that $\lambda_{ij} \neq 0$. We will prove that VOQ_{ij} cannot remain active for more than a bounded time D. Consider a VOQ_{ij} with $\lambda_{ij} \neq 0$ that remains active for a time D. Hence, if VOQ_{ij} became active at time t, it will remain active during $[t, t + D]$. Recall that the switching algorithm will compute a π-*stable* matching (Definition 2.6) in each matching phase, where π is the *Latest Activation Time* priority scheme. As a result, in every matching phase m during $[t, t + D]$, either VOQ_{ij} is served, or an active VOQ_{ik} is served and $VOQ_{ij} \not\prec_{\pi_m} VOQ_{ik}$ for some k, or an active VOQ_{kj} is served and $VOQ_{ij} \not\prec_{\pi_m} VOQ_{kj}$ for some k. In other terms, either a packet from VOQ_{ij} is forwarded, or a packet from VOQ_{kj} with an activation time no earlier than that of VOQ_{ij} is forwarded, or a packet from VOQ_{kj} with an activation time no earlier than that of VOQ_{ij} is forwarded. Therefore, by Lemma 2.1, at least $SD - 1$ packets that satisfy the above criterion were forwarded from input i or to output j during $[t, t+D]$. Moreover, by definition of the *Latest Activation Time* priority scheme, these packets arrived to the switch no earlier than time t. The number of these packets can be bounded as follows using the property of a strong constant burst traffic:

$$\lambda_{ij}D + B + \sum_{k \neq j}(\lambda_{ik}D + B) + \sum_{k \neq i}(\lambda_{kj}D + B) \leq (2\alpha - \lambda_{ij})D + (2N - 1)B$$

We reach a contradiction if $SD - 1 > (2\alpha - \lambda_{ij})D + (2N - 1)B$ or if $D > \frac{(2N-1)B+1}{S-2\alpha+\lambda_{ij}}$. If $S = 2\alpha$, D is at most $\frac{(2N-1)B+1}{\lambda_{ij}}$. Therefore, VOQ_{ij} cannot remain active for more than $\frac{(2N-1)B+1}{\lambda_{ij}}$. As a result, the length of VOQ_{ij} cannot exceed $\lambda_{ij}\frac{(2N-1)B+1}{\lambda_{ij}} + B =$

$2NB + 1$ by the property of the strong constant burst traffic. ■

As it can be seen from the proof of the above theorem, the bound on the length of a VOQ depends on the traffic but only through its burst constant B. Next we state a delay result similar to Theorem 3.2 but with weaker conditions on the traffic.

Theorem 3.4 *Under a weak constant burst traffic, the Latest Activation Time switching algorithm achieves a delay bound on every packet with a speedup $S > 2\alpha$, where α is the loading of the switch.*

Proof: We will prove that a VOQ cannot remain active for more than a time D, which in turn will imply a delay bound of $(S+1)D + 2 + B$ on every packet for a weak constant burst traffic, as argued in the proof of Theorem 3.2. The proof is identical to that of Theorem 3.3 except that, by the property of a weak constant burst traffic, the number of packets arriving to the switch during $[t, t + D]$, which originate at input i or are destined to output j, can be bounded as follows:

$$\alpha D + B + \alpha D + B = 2\alpha D + 2B$$

As argued in the proof of Theorem 3.3, we reach a contradiction if $SD - 1 > 2\alpha D + 2B$ or if $D > \frac{2B+1}{S-2\alpha}$. Therefore VOQ_{ij} cannot remain active for more than $\frac{2B+1}{S-2\alpha}$. ■

3.3 Implementation Issues

In this section, we look at the implementation details of both algorithms. The nature of the priority schemes used will make the implementation of both algorithms very practical. For instance, in both cases, the algorithm can maintain a queue that holds indices of VOQs. Whenever a VOQ becomes active, its index is added to the tail of the queue. Whenever a VOQ becomes inactive, its index is removed from the queue. In every matching phase, the *Earliest Activation Time* switching algorithm will consider the VOQs starting from the head of the queue. In other terms, the highest priority VOQ will be at the head of the queue. On the other hand, the *Latest Activation Time* switching algorithm will consider the VOQs starting from

the tail of the queue. In other terms, the highest priority VOQ will be at the tail of the queue. In doing so, each algorithm will guarantee that a matching based on its specific priority scheme is computed in every matching phase.

3.3.1 Time and Space Complexity

The time complexity of both algorithms is clearly $O(N^2)$ using the RAM model of computation. The space complexity is the amount of memory needed to maintain the queue of VOQ indices. This is $O(N^2 \log N)$ since there are at most N^2 indices each of which can be represented using $O(\log N^2) = O(\log N)$ space. Note that both algorithms do not require packets to be tagged by their arrival time, nor do they require to keep any information about the length of the VOQs or the output queues.

3.3.2 Communication Complexity

In this section we consider the amount of communication needed between the switching algorithm and the switch. Note that the switching algorithm obtains its input from the switch itself in order to compute a matching. Therefore, the switching algorithm can be considered as being performed on a central scheduler in the switch. The scheduler needs to obtain information about the VOQs in every matching phase to compute a matching. Moreover, it needs to communicate back some information. For instance, the matching itself needs to be communicated back to the switch so that the input and output ports are configured appropriately.

In our case, a VOQ that becomes active needs to be communicated to the scheduler so that our algorithm can add its index to the queue. Moreover, a VOQ that becomes inactive needs to be communicated to the scheduler as well so that our algorithm can drop its index from the queue. By the property of a matching, we know that at most one VOQ can become inactive at an input during a single matching phase, since at most one VOQ can be served at that input during a single matching phase.

If at most one VOQ can become active at a given input during a single matching

phase, then the communication requirement needed from the switch to the scheduler is $O(N \log N)$, since each input will have to communicate at most two indices (the index of the VOQ that becomes active at that input and the index of the VOQ that becomes inactive at that input). Note that this communication complexity is optimal if we consider the $\Omega(N \log N)$ amount of communication needed from the scheduler to the switch to specify a matching for the switch.

Depending on the implementation of the VOQs however, it might be possible that more than one VOQ can become active at a given input during a matching phase. This will bring the communication complexity to $O(N^2)$ since up to N VOQs can become active at a given input. We suggest a modification to the *Earliest Activation Time* and the *Latest Activation Time* switching algorithms in order to reduce the communication complexity back to $O(N \log N)$.

As mentioned above, the high communication complexity comes from the fact that multiple VOQs at an input can become active during the same matching phase. We will restrict every input to communicate at most one active VOQ in the following way: every input will communicate active VOQs in the order they become active, only one VOQ at a time. This means that when a VOQ is declared active, it could have been active for at most $\frac{N}{S}$ time; therefore, it will have at most a bounded number of packets, which can be added to the burst constant of the traffic (whether the weak or the strong constant burst traffic is being used). Hence, a VOQ will not remain active for more than $\frac{N}{S}$ time in addition to the bound obtained with the adjusted burst constants.

3.4 Summary

We presented two priority switching algorithms that provide strong throughput with a speedup $S = 2$ and a delay guarantee with a speedup $S > 2$, under appropriate constant burst traffic models. Both algorithms offer the advantage of not requiring extensive state information like the age of packets (as in the *Oldest Cell First* algorithm [6] and *LOOFA* [18]), the length of the VOQs (as in the *Central Queue*

algorithm [16]), or the length of the output queues (as in *LOOFA* [18]). Moreover, they do not require the traffic to be constantly backlogged as it is the case for the *Central Queue* algorithm [16]. The running time of both algorithms is $O(N^2)$ in the RAM model of computation and their memory requirement is $O(N^2 \log N)$. The communication complexity of both algorithms is $O(N \log N)$ which is optimal if we consider the $\Omega(N \log N)$ amount of communication required to specify a matching for the switch in order to configure the input and output ports appropriately. Therefore, both algorithms offer a better communication requirement compared to the previous algorithms for which more information needs to be communicated, like the age of packets for instance.

Chapter 4

An Iterative Switching Algorithm

We present in this chapter an iterative switching algorithm that we call π-RGA. As described in Chapter 1, an iterative switching algorithm operates in iterations within a single matching phase, where in each iteration some input and output ports are matched. Examples of these algorithms are *PIM* [1] [1], *iSLIP* [22], *iPP* [13], *DRR* [20], and *pDRR* [9]. For a brief description of some of these algorithms, see Section 1.7.3. For a comparison among these different algorithms, see [2], [9], and [13]. In all of these algorithms, the matching computed in one iteration is not necessarily maximal as described in Chapter 1. In other terms, more input and output ports can still be matched. The reason for this is the following. Each iteration is composed of three stages: *Request*, *Grant*, and *Accept* (hence the name of the algorithm presented here). In the *Request* stage, inputs send matching requests to the outputs. In the *Grant* stage, each output grants at most one request. Finally, in the *Accept* stage, each input accepts at most one granted request. Since different inputs might request the same output, and similarly, different outputs might grant the same request, the resulting matching might not be maximal. This situation cannot be avoided in general because there is no direct communication among the input ports themselves or among the output ports themselves, as this would lead to a more complicated hardware.

Nevertheless, with additional iterations in which previously matched inputs and

[1] *PIM* uses randomness and reaches a maximal matching with $O(\log N)$ iterations on average. A variation on *PIM*, also presented in [1] and called statistical matching, achieves theoretically 72% throughput with 2 iterations.

outputs do not participate in the RGA protocol, more inputs and outputs will be matched, thus leading to a larger size matching. A larger size matching will generally imply higher throughput of the switch; however, from the theoretical point of view, the required additional iterations limit the speed of the switch, since more iterations will be performed in one matching phase of the switch. The work on π-*RGA* is motivated by the following observations:

- All proposed iterative algorithms practically achieve 90%-95% throughput with multiple iterations and no speedup [2]. The number of iterations is experimentally found to be $O(\log N)$ iterations.

- Some of the iterative algorithms can be proved to achieve theoretically 100% throughput with one iteration but only when the traffic is uniform, i.e. the rate of packets from an input to an output is the same all over the switch.

Therefore, we would like to limit the number of iterations to one iteration only and still provide high throughput for an arbitrary traffic pattern [3] even with that one iteration.

Limited to one iteration only, the π-*RGA* switching algorithm attempts to maintain parts of the previously computed matching in order to grow the size of the matching with successive matching phase. Therefore, instead of restarting the computation of a matching from scratch in every matching phase, π-*RGA* uses information about the previous matching. In doing so, the π-*RGA* algorithm differentiates between two kinds of requests: *Strong* and *Weak* requests. For instance, requests that were granted and accepted become *Strong* requests in the following matching phase. Precedence is given to the *Strong* requests, and hence the matching will tend to stabilize with successive matching phases towards a matching that grants the *Strong* requests. By

[2] A lower bound on the speedup required to achieve throughput can be proved for a number of iterative algorithms. For instance, using a traffic like the one in Figure 2-3, we can prove that *iSLIP* and *DRR* require a speedup $S > \frac{3}{2}\alpha$ to achieve weak throughput. Therefore, with no speedup, these algorithms cannot achieve more than 66.67% throughput.

[3] The traffic may be other than uniform, unlike the analysis provided in [20] for *DRR*, which is also a one iteration algorithm.

not competing with requests at other inputs, *Weak* requests will help the stabilization process to grow the size of the matching with successive matching phases. A priority scheme π is used in conjunction with the *Strong* and *Weak* modifiers in order to ensure that the stabilization process favors connections with high priority. The priority scheme π, therefore, serves like a parameter to the algorithm as suggested in the name π-RGA. The properties of the priority scheme π will be discussed later in the chapter.

We will show that with an appropriate priority scheme π, the π-RGA switching algorithm achieves strong throughput with a speedup of 2 and a delay guarantee with a speedup $S > 2$, under a strong constant burst traffic.

The π-RGA algorithm was developed initially with a particular theoretical framework in mind (the standard switch model presented in Figure 1-5). Note however, that later in the chapter, we will present adaptations of π-RGA for a burst switch (will be explained later) as well as for a multiple server burst switch, an architecture described in [9] (will be presented later as well).

4.1 The π-RGA Switching Algorithm

As mentioned earlier, the π-RGA arbitration algorithm works in three stages: *Request*, *Grant*, and *Accept*. We will use Definition 2.2 of an active VOQ and define a VOQ transition to be a transition of the VOQ from being inactive to active or vice-versa. We will also use Definition 2.3 of a priority scheme. We will further assume the following: For any matching phase m, if VOQ_{ij} and VOQ_{ik} are active, then either $VOQ_{ij} \prec_{\pi_m} VOQ_{ik}$ or $VOQ_{ik} \prec_{\pi_m} VOQ_{ij}$. Similarly, for every matching phase m, if VOQ_{ij} and VOQ_{kj} are active, then either $VOQ_{ij} \prec_{\pi_m} VOQ_{kj}$ or $VOQ_{kl} \prec_{\pi_m} VOQ_{ij}$. In simpler terms, during matching phase m, active VOQs that share either an input or an output must be ordered by π_m. During a matching phase m, a π-highest VOQ_{ij} for a set of VOQs Q is such that there is no $VOQ_{kl} \in Q$ with $VOQ_{kl} \prec_{\pi_m} VOQ_{ij}$; furthermore, if $VOQ_{ij} \in Q$, we say VOQ_{ij} is π-highest in Q.

Figure 4-1 shows the π-RGA switching algorithm for matching phase m.

Algorithm π-RGA

start with an empty matching $M = \emptyset$
repeat for a number of iterations (possibly only once)

R (at unmatched input i): if no VOQ was served in matching phase $m-1$, send *Strong* requests for all active VOQs; otherwise, if there is an active VOQ_{ij_0} that was served in matching phase $m-1$, send *Strong* requests for all active VOQ_{ij} such that $VOQ_{ij_0} \not\prec_{\pi_m} VOQ_{ij}$, and *Weak* requests for all other active VOQs.

G (at unmatched output): with R being the set of requests received, if there are *Strong* requests in R, grant the π-highest *Strong* request in R; otherwise, grant the π-highest *Weak* request in R if any.

A (at unmatched input): with G being the set of grants received, if there are granted *Strong* requests in G, accept the π-highest granted *Strong* request in G; otherwise, accept the π-highest granted *Weak* request in G if any.

if input i accepts a grant from output j
$M = M \cup (i, j)$

Figure 4-1: The π-*RGA* switching algorithm

As a side remark, the sequence input-output-input where the three stages of an iteration are performed can be alternatively changed to output-input-output. But since information about VOQs is more naturally obtained at the input side, we adopt the sequence shown above.

The most crucial aspect of the π-*RGA* algorithm is the way requests are prepared. Every input sends *Strong* requests for active VOQs that have high priority, where the threshold of high priority is the priority of the previously served VOQ. In other terms, an input attempts to request better service based on the priority scheme π. Moreover, since *Weak* requests, regardless of their priority, are always considered next, an input which has already accepted a high priority granted request will not prohibit other inputs from matching (by sending its low priority requests which are going to be *Weak*).

Therefore, to summarize what has been described so far, this innovative approach behind π-*RGA* can be conceptually visualized as having three different components:

- The presence of *Strong* requests help stabilize the matching by creating requests that will always tend to be granted. Therefore, in the absence of multiple

iterations, the matching needs not be computed from scratch.

- The presence of a priority scheme helps guide the stabilization process of the matching by determining which requests can become *Strong*.
- The presence of *Weak* requests help grow the size of the matching with successive matching phases by making some requests, that are unlikely to be granted, not compete with other requests.

Note that in the absence of multiple iterations, the use of *Weak* requests emulates the process by which a matched input stops sending requests in future iterations. For this reason, when the number of iterations is fairly large, π-*RGA* might not be the best algorithm to use.

We have not yet specified the priority scheme π to be used. It is obvious that if π changes arbitrarily from one matching phase to the other, the *Strong* requests (and hence the grants) will become arbitrary, yielding to an unstable matching. This will make it difficult to realize the main goal of this algorithm, which is to maintain parts of the previously computed matching in order not to require multiple iterations for growing the size of the matching.

In the following sections, we discuss formally some of the properties that π might have and their implications on the performance of the π-*RGA* switching algorithm.

4.2 Stable Priority Scheme π

In this section we define a stable priority scheme:

Definition 4.1 (stable π) *A priority scheme π is a stable priority scheme iff it satisfies the following: if VOQ_{ij} and VOQ_{kl} remain active during a time interval T, and $VOQ_{ij} \prec_{\pi_{m_0}} VOQ_{kl}$ for some matching phase m_0 in T, then $VOQ_{ij} \prec_{\pi_m} VOQ_{kl}$ for every matching phase $m > m_0$ in T.*

When π is stable, the π-*RGA* algorithm will attempt to stabilize a maximal matching that favors higher priority *VOQ*s. In other terms, it will attempt to reach a

π-stable matching. The reason for this is the following: if no VOQ transitions occur, then a highest priority VOQ_{ij} will have a *Strong* request and will start being served (if it was not already being served), and will keep getting served until inactive. The next highest priority VOQ_{kl} that can still be served will start being served next (if it was not already being served), and will keep getting served until inactive; this is guaranteed by the *Request* stage which will issue *Weak* requests for all VOQs except VOQ_{ij} at VOQ_{ij}'s input i. These *Weak* requests will allow the request for VOQ_{kl} to be granted. This continues until the matching is stabilized after at most N matching phases. Note that in this stabilization process, input i will have at most one *Strong* request (VOQ_{ij}), input k will have at most two *Strong* requests (VOQ_{kl} and possibly VOQ_{kj}), and so on. In this resulting matching, a VOQ that is not served is blocked by a higher priority VOQ, and hence the matching is *π-stable.*

The above reasoning assumed that no VOQ transitions occur; however, if VOQ transitions do occur, the matching might be perturbed every time there is such transition. Nevertheless, we can still have a notion of stability for a particular VOQ even in the presence of VOQ transitions. This notion is captured in the following definition.

Definition 4.2 *For a priority scheme π, a matching computed in matching phase m is π-stable with respect to VOQ_{ij} iff it satisfies the following condition: If VOQ_{ij} is active and is not served by the matching, then either an active VOQ_{ik} is served by the matching and $VOQ_{ij} \not\prec_{\pi_m} VOQ_{ik}$, or an active VOQ_{kj} is served by the matching and $VOQ_{ij} \not\prec_{\pi_m} VOQ_{kj}$.*

Note that the above definition is a relaxation of Definition 2.6 in the sense that the matching satisfies the property with respect to VOQ_{ij} only instead of all VOQs. Note also that if no VOQ transitions occur, as argued above, *π-RGA* will reach a *π-stable* matching with respect to all VOQs in at most N matching phases. The interesting observation is that a transition for VOQ_{kl} will not affect the notion of stability in Definition 4.2 with respect to VOQ_{ij} if VOQ_{kl} does not have higher priority than VOQ_{ij}. More precisely, we have Lemma 4.1 below. In Lemma 4.1, S is, as before, the speedup of the switch.

Lemma 4.1 *Given a stable priority scheme π and a VOQ_{ij} that remains active during a time interval $[t, t + \frac{N}{S}]$, if all VOQ transitions during $[t, t + \frac{N}{S}]$ are only for a set of VOQs Q such that VOQ_{ij} is π-highest for Q during $[t, t + \frac{N}{S}]$, then the matching computed by the π-RGA switching algorithm in matching phase $\lceil tS \rceil + N - 1$ is π-stable with respect to VOQ_{ij} and all VOQ_{kl} such that $VOQ_{kl} \prec_{\pi_{\lceil tS \rceil}} VOQ_{ij}$.*

Proof: The proof is similar to the argument that a π-*stable* matching will be reached in at most N matching phases in no VOQ transitions occur. Starting from the first matching phase $\lceil tS \rceil$, we only consider in the argument VOQ_{ij} and the VOQs that have higher priority than VOQ_{ij}.

Let Q_0 be the set containing VOQ_{ij} and all VOQ_{kl} such that $VOQ_{kl} \prec_{\pi_{\lceil tS \rceil}} VOQ_{ij}$. Note that, since π is stable, no transitions occur for VOQs in Q_0 during $[t, t + \frac{N}{S}]$ by the condition of the lemma. Note also that by assumption, during a matching phase m, VOQs that share an input or an output port are ordered by π_m, and hence the highest priority VOQ at an input or an output is uniquely determined during a matching phase.

Regardless of any transitions for VOQs outside Q_0, a π-highest priority VOQ_{kl} in Q_0 will be served in matching phase $\lceil tS \rceil$, since it is the highest priority VOQ at input k and output l and will therefore be issued a *Strong* request that will be granted and accepted. Moreover, VOQ_{kl} will keep getting served until matching phase $\lceil tS \rceil + N - 1$ since no VOQ transitions occur in Q_0 before that time, and hence VOQ_{kl} remains π-highest in Q_0 during $[t, t + \frac{N}{S}]$ by the stability property of π.

Let Q_1 be the set obtained by removing from Q_0 all the VOQs that have input k or output l. Note that all VOQs in $Q_0 - Q_1$ that are not served during this and the next $N - 1$ matching phases are blocked by VOQ_{kl} which is π-highest in $Q_0 - Q_1$ during $[t, t + \frac{N}{S}]$.

A π-highest VOQ_{mn} in Q_1 will be served in matching phase $\lceil tS \rceil + 1$. To see this, observe that a *Strong* request will be issued for VOQ_{mn} at its input m. The reason is the following: If no VOQ at input m was served in matching phase $\lceil tS \rceil$, then input m will issue *Strong* requests for all its VOQs. If on the other hand, a VOQ at input m was served in matching phase $\lceil tS \rceil$, then it was either VOQ_{mn} or a VOQ

with lower priority than VOQ_{mn}, since all VOQs with higher priority than VOQ_{mn} are in $Q_0 - Q_1$, and hence correspond to output l which was matched to input k in matching phase $\lceil tS \rceil$. Therefore, input m will issue a *Strong* request for VOQ_{mn}, which will be granted by output n since all requests in $Q_0 - Q_1$ coming from input i to output n will be *Weak* requests, and all requests in Q_1 to output n have lower priority. Again, VOQ_{mn} will keep getting served until matching phase $\lceil tS \rceil + N - 1$ since no VOQ transitions in $(Q_0 - Q_1) \cup Q_1 = Q_0$ occur before that time, and hence VOQ_{mn} remains π-highest in Q_1 during $[t, t + \frac{N}{S}]$ by the stability property of π.

We define Q_2 to be the set obtained by removing from Q_1 all the VOQs that have input m or output n. As before, all VOQs in $Q_1 - Q_2$ that are not served during this and the next $N - 2$ matching phases are blocked by VOQ_{mn} which is π-highest in $Q_1 - Q_2$ during $[t, t + \frac{N}{S}]$.

The argument is carried forward until matching phase $\lceil tS \rceil + N - 1$ where we define $Q_N = \emptyset$ (since the size of a matching cannot be more than N). Since $Q_0 = (Q_0 - Q_1) \cup (Q_1 - Q_2) \cup \ldots \cup (Q_{N-1} - Q_N)$, in the resulting matching during matching phase $\lceil tS \rceil + N - 1$, if a VOQ in Q_0 is not served, then it must be blocked by another VOQ in Q_0 of higher priority according to π. Therefore, the matching computed in matching phase $\lceil tS \rceil + N - 1$ is *π-stable* with respect to all VOQs in Q_0. ■

Therefore, Lemma 4.1 establishes the property that π-*RGA* with a stable priority scheme π will be able to sustain a stable matching for a set of high priority VOQs, so long as they remain active.

The following section defines another property of the priority scheme π that will be useful for the operation of the π-*RGA* switching algorithm.

4.3 Bounded Bypass Priority Scheme π

In this section we define a bounded bypass priority scheme:

Definition 4.3 (bounded bypass π) *A priority scheme π is a bounded bypass priority scheme iff it satisfies the following: for any time interval T in which VOQ_{ij}*

remains active, VOQ_{kl} *becomes active at some time* t *in* T *with* $VOQ_{kl} \prec_{\pi_{\lfloor tS \rfloor}} VOQ_{ij}$ *(i.e.* VOQ_{kl} *"bypasses"* VOQ_{ij}*) at most a bounded number of time* b *in* T*.*

As mentioned in the previous section, with a stable priority scheme, the matching might be perturbed every time there is a VOQ transition. The bounded bypass property limits the number of times this perturbation occurs with respect to a particular VOQ. By the bounded bypass property, a VOQ_{ij} that remains active will eventually become (and remain thereafter) a π-highest VOQ (after at most a bounded number of transitions for VOQs with higher priority than VOQ_{ij}); and therefore, if π is also stable, the matching will become π-*stable* with respect to VOQ_{ij} and remains so until VOQ_{ij} in inactive.

We can loosely bound the number of matching phases in which the matching is not π-*stable* with respect to VOQ_{ij} as follows:

Lemma 4.2 *Given a stable bounded bypass priority scheme* π *and a* VOQ_{ij} *that remains active during a time interval* T*, the matching computed by the* π*-RGA switching algorithm is* π*-stable with respect to* VOQ_{ij} *for all matching phases in* T *except for at most* $(2b+1)(N^2-1)N$ *matching phases, where* b *is the bound from the bounded bypass property of* π*.*

Proof: Since π is a bounded bypass priority scheme, a VOQ can "bypass" VOQ_{ij} at most a bounded number of times b. More precisely, during a time interval T in which VOQ_{ij} remains active, VOQ_{kl} can become active at some time t in T with $VOQ_{kl} \prec_{\pi_{\lfloor tS \rfloor}} VOQ_{ij}$ only a bounded number of times b. This implies that VOQ_{kl} can become inactive at some time t in T while $VOQ_{kl} \prec_{\pi_{\lfloor tS \rfloor}} VOQ_{ij}$ at most $b+1$ times; since otherwise, VOQ_{kl} bypasses VOQ_{ij} more than b times for the following reasoning: between any two times $t_1 < t_2$ in T at which VOQ_{kl} becomes inactive, there must exist a time $t \in (t_1, t_2)$ at which VOQ_{kl} becomes active, and $VOQ_{kl} \prec_{\pi_{\lfloor tS \rfloor}} VOQ_{ij} \Leftrightarrow VOQ_{kl} \prec_{\pi_{\lfloor t_2 S \rfloor}} VOQ_{ij}$ by the stability property of π.

Therefore, a transition for a VOQ with higher priority than VOQ_{ij}, can occur at most $2b+1$ times while VOQ_{ij} is active. Since we have at most (N^2-1) VOQs other than VOQ_{ij}, a transition for a VOQ with higher priority than VOQ_{ij} can occur at

most $(2b+1)(N^2-1)$ times while VOQ_{ij} is active. Since π is a stable priority scheme, a π-*stable* matching with respect to VOQ_{ij} can be reached in at most N matching phases after a "bypass" as stated in Lemma 4.1. Therefore, the number of matching phases in T for which the matching is not π-*stable* with respect to VOQ_{ij} is at most $(2b+1)(N^2-1)N$. ■

4.4 Theoretical Results

Lemma 4.2 implies that the π-*RGA* switching algorithm satisfies the following *local stability* property with a stable bounded bypass [4] priority scheme π.

Definition 4.4 (local stability) *During any time interval T in which VOQ_{ij} remains active, the matching is π-stable with respect to VOQ_{ij} for every matching phase in T except for at most a bounded number of matching phases.*

The *local stability* property implies some sort of a *local maximality* property. For instance, it implies that, during a time interval T in which VOQ_{ij} remains active, either input i is matched or output j is matched except for a bounded number of matching phases. Note that π-*RGA* might never succeed in computing a maximal matching; however, for a particular active VOQ, the matching will always be "locally" maximal except for a bounded number of matching phases.

Next, we enumerate the guarantees of π-*RGA* under different traffic models.

4.4.1 SLLN Traffic

It has been shown in [8] that a maximal matching policy guarantees weak throughput with probability 1 under any SLLN traffic with a speedup $S \geq 2\alpha$. It can be shown that the result of [8] still holds for any switching algorithm that satisfies the *local*

[4]Note that the stable and bounded bypass properties as presented here are not the most general restrictions on π that ensure the *local stability* property. A generalized form of the bounded bypass property, in which a VOQ_{kl} can acquire a higher priority then VOQ_{ij} only a bounded number of times while VOQ_{ij} is active, is enough by itself to ensure the *local stability* property. However, the stable property is important in practice to stabilize the matching more quickly and achieve high throughput.

stability property. Therefore, we have the following result: For any stable bounded bypass priority scheme π, the π-*RGA* switching algorithm guarantees weak throughput with probability 1 under an SLLN traffic with a speedup $S \geq 2\alpha$.

4.4.2 Weak Constant Burst Traffic

It has been shown in [6] that a maximal matching policy guarantees a delay bound on every packet under a weak constant burst traffic with a speedup $S > 4\alpha$. Again, it can be shown that the result of [6] still holds for any switching algorithm satisfying the *local stability* property. Therefore, we have the following result: For any stable bounded bypass priority scheme π, the π-*RGA* switching algorithm guarantees a delay bound on every packet under a weak constant burst traffic with a speedup $S > 4\alpha$.

By strengthening the burst condition, we can provide guarantees with a less stringent speedup requirement. We strengthen the condition on the traffic by assuming the strong constant burst model.

4.4.3 Strong Constant Burst Traffic

We will prove the following results stated in Theorem 4.1 and Theorem 4.2. Let the priority scheme π_0 be the *Earliest Activation Time* [5] priority scheme defined in Chapter 3 with an embedded tie breaking that uses the indices of the input and output ports (or any other way that ensures π_0 is stable). Note that π_0 is a stable bounded bypass priority scheme.

Theorem 4.1 *With the particular stable bounded bypass priority scheme π_0, the π_0-RGA switching algorithm achieves strong throughput under a strong constant burst traffic with a speedup $S = 2\alpha$, where α is the loading of the switch.*

Proof: The proof is essentially the same as that of Theorem 3.1. In Theorem 3.1, we are computing a π_0-*stable* matching in every matching phase while VOQ_{ij} is

[5] Now that we are using a distributed way of computing the matching, keeping a centralized clock to compute the *Activation Times* might be inefficient in hardware. However, the similar effect of a centralized clock can be obtained if each port keeps a local counter and the values of the counters are communicated in the messages between the ports [19].

active, whereas here, we are computing a matching that is π_0-*stable* with respect to VOQ_{ij} in every matching phase while VOQ_{ij} is active, except for a bounded number of matching phases, say K. Therefore, the term $SD-1$ in Theorem 3.1, which is a lower bound on the number of forwarded packets that satisfy certain criterion (see Theorem 3.1), can be replaced by $SD-1-K$. This will imply a bound of $\frac{(2N-1)B+1+K}{S-2\alpha+\lambda_0}$ on the time VOQ_{ij} can remain active, yielding a bound of $\lambda_{ij}\frac{(2N-1)B+1+K}{S-2\alpha+\lambda_0}+B$ on the length of VOQ_{ij} in case $\lambda_{ij}\neq 0$, where $\lambda_0=\min_{i,j|\lambda_{ij}\neq 0}\lambda_{ij}$, and B is the burst constant. In case $\lambda_{ij}=0$, the length of VOQ_{ij} cannot exceed the burst constant B.

■

Theorem 4.2 *With the particular stable bounded bypass priority scheme π_0, the π_0-RGA switching algorithm achieves a delay bound on every packet under a strong constant burst traffic with a speedup $S>2\alpha$, where α is the loading of the switch.*

Proof: The proof is identical to that of Theorem 3.2 using the bound of $\frac{(2N-1)B+1+K}{S-2\alpha}$ from Theorem 4.1. ■

Note that π_0 is a zero bypass priority scheme, i.e. a VOQ that becomes active will have the lowest priority. Hence, once a matching is maximal, it will remain maximal until a VOQ becomes inactive. This was found experimentally to be useful. The following section provides some of the experimental results that were done to compare the performance of π-*RGA* to the performance of another iterative switching algorithm *pDRR* found in [9].

4.5 Experimental Results

While the results mentioned above hold for a speedup of 2, we simulated the π-*RGA* switching algorithm with no speedup and with the priority scheme $\pi=\pi_0$ defined above. In the rest of this chapter, π-*RGA* actually refers to π_0-*RGA*. The simulations showed that π-*RGA* with no speedup is capable of sustaining fairly high loads with one iteration only. We will show performance comparisons between π-*RGA* and *pDRR* (stands for Prioritized Dual Round Robin) which proved to perform better than *PIM*, *iSLIP*, *iPP*, and pure *DRR*.

Before we describe *pDRR*, we need to introduce the concept of a burst switch. In a burst switch, the size of a packet is less than the transmission unit of the switch. The switch can forward up to B (this is not related to the burst constant B of the traffic) packets from a VOQ. This does not mean that the switch is operating at a speedup B, as it is possible for B packets to arrive at one input per time unit and be distributed among many VOQs at that input. Therefore, when a matching is computed, B packets are forwarded from VOQ_{ij} if input i is matched to output j. If VOQ_{ij} contains less than B packets however, then as many packets as VOQ_{ij} contains will be forwarded. The burst switch model appears in many places in high speed optical networks where the configuration speed of the optical switch is slow compared to its transmission speed; and hence, multiple packets will be forwarded once the switch is configured with a particular matching. We will refer to this grouping of packets as "burstification". Note that due to this burstification, the switch might loose on throughput by forwarding less than B packets from a VOQ. Stated in a different way, at an input, the traffic is arriving in bursts of up to B packets that are distributed among the different VOQs, but the switch cannot distribute its capacity and has to serve one VOQ at a time, thus loosing bandwidth. Therefore, it might be harmful to match input i to output j if VOQ_{ij} contains less than B packets. If $B = 1$, then we get our previous model which does not have the problem mentioned above.

The *pDRR* algorithm holds five priority classes where P_0 is the lowest priority and P_4 is the highest priority. Each VOQ acquires a priority depending on its state, as described in the table below:

P_0	contains less than B packets
P_1	contains at least B packets
P_2	contains at least $5B$ packets
P_3	contains at least $10B$ packets
P_4	contains a packet that has been waiting for more than a TIMEOUT constant

Given these priority classes, an input sends requests for all its VOQs with their

corresponding priorities. At the output, the request with the highest priority will be granted, and ties are broken with a round robin strategy. Similarly, at the input, the highest priority granted request is accepted, and ties are broken with a round robin strategy.

Our comparisons were done using a 16×16 switch with a geometrically polarized SLLN traffic (see [3]). Basically, a geometrically polarized traffic is a traffic in which at an input i, it is possible to order the outputs 1 to N such that $\lambda_{i1} = \rho\lambda_{i2} = \rho^2\lambda_{i3} = ... = \rho^{N-1}\lambda_{iN}$, where $0 < \rho < 1$ is the polarization factor. This is one way of producing a non-uniform traffic.

4.5.1 Standard Switch

With no burstification, *π-RGA* performed better than *pDRR* when the number of iterations was less than three. With three iterations, *pDRR* performed better on high loads. Recall that our motivation in developing *π-RGA* was to reduce the number of iterations to one iteration only and still support a high traffic load. Below is a figure showing the performances of *π-RGA* and *pDRR* with one iteration.

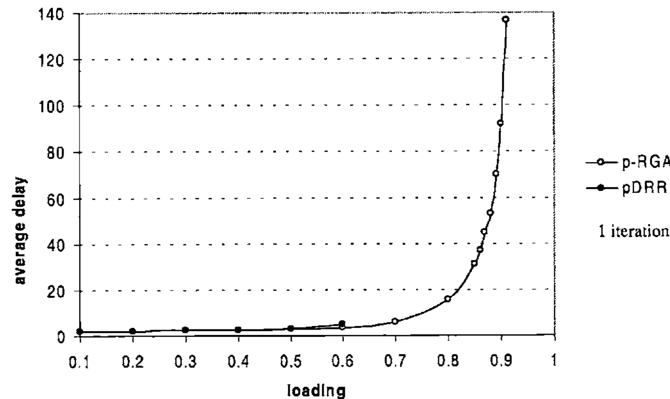


Figure 4-2: *π-RGA* and *pDRR* with one iteration for $B = 1$

As it can be seen from the figure above, *π-RGA* with one iteration is able to support up to 90% loading (93% with two iterations), while *pDRR* fails at 60%. For a burst switch however, *π-RGA* does not perform as well as shown above, as will be explained next.

4.5.2 Burst Switch

The problem that burstification introduces, as discussed earlier, is that serving a VOQ with less than B packets leads to an under-utilization of the switch, since the switch can forward up to B packets from a VOQ. We can refer to this problem as serving non-full bursts. The basic π-RGA does not perform well in this setting because of its greedy nature: it will keep on serving a VOQ until the VOQ becomes inactive (empty), implying that the switch will serve non-full bursts more often (possibly every time it empties a VOQ). Therefore, we modified π-RGA such that the definition of active VOQ is changed to a VOQ that contains at least B packets. We call π-RGA' this modification of π-RGA. Again, when the number of iterations was less than three, π-RGA' performed better than $pDRR$ at high loads. Figure 4-3 shows the performance of π-RGA' compared to that of $pDRR$ for one iteration.

Note that a consequence of the new modification in π-RGA' is that low loads will have higher delays since a VOQ is not considered active until it acquires B packets. Note also that with the burst switch, π-RGA and π-RGA' fail just before reaching 90% loading with one iteration (slightly worse than the case where $B = 1$).

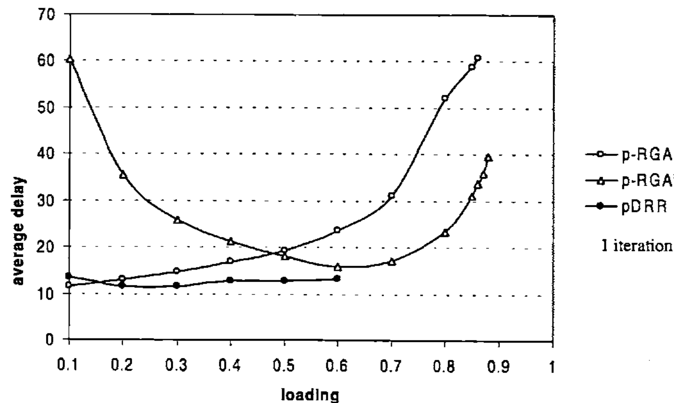


Figure 4-3: π-RGA' and $pDRR$ with one iteration for $B = 256$

The result shown in Figure 4-3 suggests another modification to π-RGA which will enhance its performance for low loads: instead of considering VOQs with less then B packets inactive, they will be given a lowest priority P_0. Figure 4-4 illustrates the result for this modification, for the same burst size $B = 256$ and for one iteration.

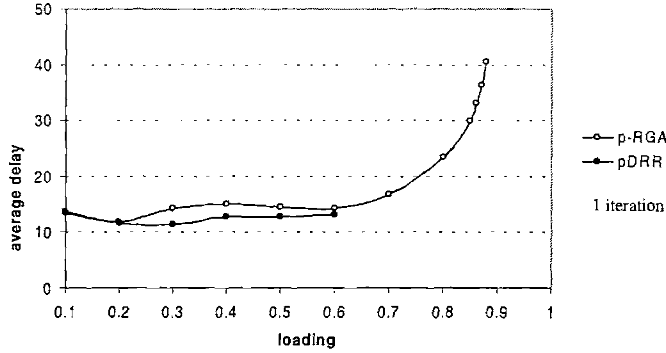


Figure 4-4: Modified π-RGA' and $pDDR$ with one iteration for $B = 256$

With this modification of the π-RGA, the lowest priority P_0 requests were handled by a DRR strategy. Therefore, Figure 4-4 shows that a balance between DRR and π-RGA is likely to lead to a good performance for a burst switch. Next, we change the threshold of *activeness* and assign the lowest priority P_0 to VOQs with less than $3B$ packets (instead of B packets). This leads to a higher load support; however, as expected, increases the average delay. The result is depicted in Figure 4-5.

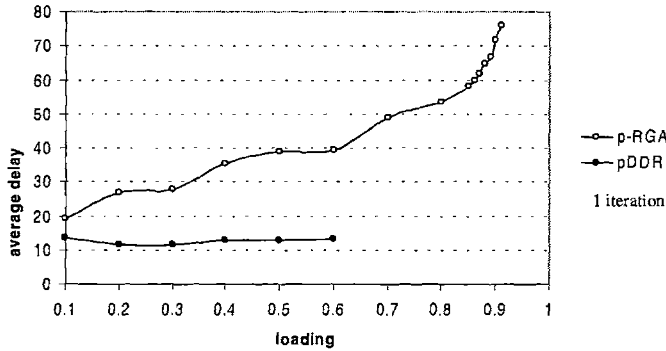


Figure 4-5: VOQ activeness $= 3B$ packets with one iteration for $B = 256$

4.5.3 Multiple Server Switch

A multiple server switch represents an architecture solution for dealing with the problem of burstification. With multiple servers, the capacity of the switch remains the same; however, it is divided into smaller granularity. Instead of being able to forward B packets from a VOQ, H servers, each capable of forwarding $\frac{B}{H}$ packets from an input to an output, will be used. The architecture is described in [9] and is illustrated

in Figure 4-6.

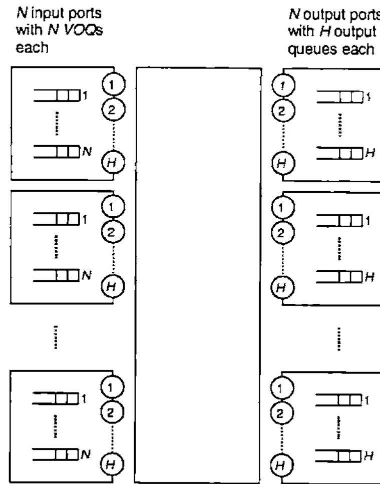


Figure 4-6: Multiple server switch model

As seen from Figure 4-6, instead of computing a matching between input ports and output ports, a matching between servers is computed. This makes it more efficient to handle non-full bursts since we can avoid dedicating the full capacity of the switch to a non-full burst, by assigning the H servers to different VOQs. Of course, more than one server (up to H) can still be assigned to the same VOQ. In fact, one way of assigning servers (at it is done in $pDRR$), is by assigning as many servers as needed for the highest priority request, and as many of the remaining servers as needed for the second highest priority request, and so on.

When multiple servers are used, more than one VOQ can be served at the same input during a single matching phase. For this reason, we need to modify the π-RGA algorithm to deal with this feature, more precisely, the definition of the previously served VOQ needs to adapt to this new setting. Since the idea behind the previously served VOQ is to keep on serving that VOQ, we simply change the definition of the previously served VOQ to the VOQ that was previously fully served by all H servers. In other terms, the previously served VOQ is defined only if there is a unique one. With this modification, we can simulate the π-RGA algorithm to study its performance with multiple servers. It turned out that $pDRR$ deals with multiple servers better than π-RGA. Therefore, we modified π-RGA further to deal with multiple

servers. The following simulation shows the result of a balance between *pDRR* and π-*RGA* as follows: all *VOQ*s with *Weak* requests are given the same low priority P_2, all *VOQ*s with less than $3B$ packets are given the same low priority P_1, and all *VOQ*s with less than B packets are given the same low priority P_0, where $P_2 > P_1 > P_0$. *VOQ*s with equal priority are handled by a *DRR* strategy. This means that this newly modified version of π-*RGA* uses four priority classes, where a *VOQ* that does not fall into the three classes mentioned above, takes the highest priority P_3. The π_0-*RGA* switching algorithm operates as usual on the *VOQ*s with priorities P_2 and P_3, with the exception that *Weak* requests are given the same priority P_2 instead of their original π_0 priority. In some sense, we use π-*RGA* on high priority *VOQ*s and *pDRR* on low priority *VOQ*s. The following figures illustrate the results for one iteration and one, two, and four servers.

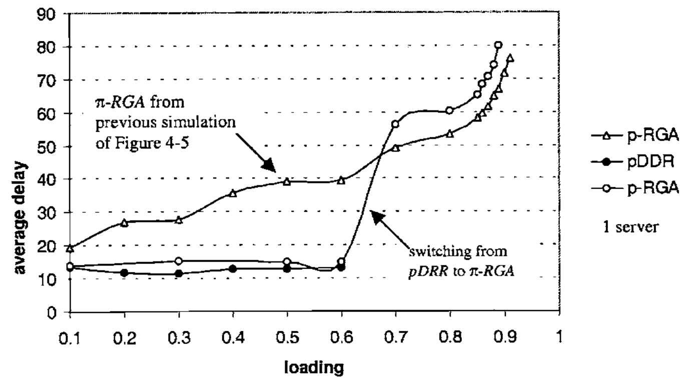


Figure 4-7: One iteration and one server for $B = 256$

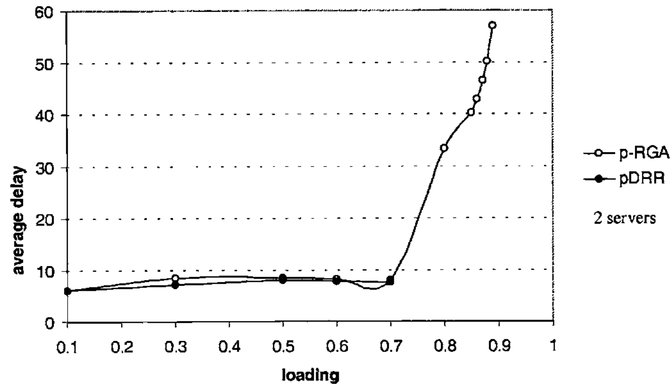


Figure 4-8: One iteration and two servers for $B = 256$

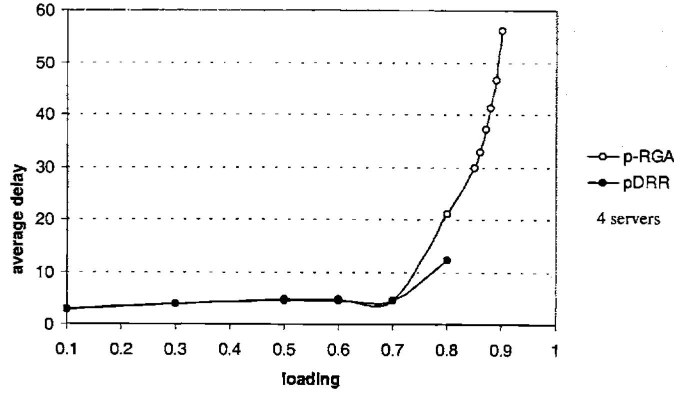


Figure 4-9: One iteration and four servers for $B = 256$

As the number of servers increases, the newly modified π-*RGA* and *pDRR* converge to the same performance.

4.6 Summary

Theoretically speaking, with a particular priority scheme π_0=*Earliest Activation Time*, the π-*RGA* switching algorithm provides strong throughput with a speedup of 2 and a delay guarantee with a speedup $S > 2$. The π-*RGA* switching algorithm requires $O(\log N)$ computational complexity to select the highest priority request with the use of appropriate parallelism at the ports. Since only one RGA iteration is needed in each matching phase (or more generally, a constant number of iterations), the computational complexity of the π-*RGA* switching algorithm will be $O(\log N)$. An algorithm that provides delay guarantees with no speedup and requires $O(\log N)$ computational complexity is described in [4]. As described in the Summary section of Chapter 2, this algorithm requires explicit knowledge of the values of λ_{ij}s, and therefore, the algorithm is sensitive to the traffic pattern. This knowledge requirement can be removed by transforming the traffic into a uniform traffic using two consecutive switches [5]. However, this will cause receiving packets in an out of order fashion and leads to the need to re-sequence packets at the output. The π-*RGA* switching algorithm does not require explicit knowledge of the values of λ_{ij}s.

Practically speaking, the π-*RGA* algorithms supports up to 90% loading with no

speedup, one iteration only, and the *Earliest Activation Time* priority scheme. For burst switches and multiple server switches, modifications to the π-RGA algorithm that establish some sort of balance between plain π-RGA and $pDRR$ proved to have better performance than plain π-RGA and $pDRR$ when the number of iterations is less than three. Such modifications, as indicated in the previous section, allow us to perform $pDRR$ at low loads and π-RGA at high loads, combining the advantages of both: non-greediness of $pDRR$ and stability of π-RGA.

Chapter 5

Switching using Parallel Switches with no Speedup

As we have seen in Chapter 1, output queued switches become increasingly inadequate to meet high speed requirements, because having to account for multiple packet arrivals to the same output requires their queue memories to operate at N times the line speed, where N is the number of inputs. Although input queued switches provide an attractive alternative since their memory and switch fabrics may operate at only the line speed, they present a challenge for providing guarantees comparable to those provided by output queued switches, and require a sophisticated switching algorithm that becomes a critical component of the switch. For instance, traditional switching algorithms that achieve 100% throughput in an input queued switch do not provide strict delay guarantees, and are based on computing a maximum weighted matching that requires a running time of $O(N^3)$ [21], [23], or $O(N^{2.5})$ [24], making them impractical to implement on high speed switches. Some recent work [7] has, therefore, focused on asking whether an input queued switch can be made to emulate an output queued switch, and has demonstrated that this can be achieved by a combination of a speedup (of $2 - \frac{1}{N}$) and a special switching algorithm based on computing a stable marriage matching [12]. Such emulation involves substantial bookkeeping and communication overhead between the switching algorithm and the switch itself, and despite its theoretical significance, is not yet practical at high speeds. Moreover, most

practical switching algorithms for input queued switches (see, for instance, [6], [18]) require a speedup of between 2 and 4 to achieve adequate guarantees.

In this chapter, we propose a parallel switching architecture that requires no speedup and provides a delay guarantee. The architecture consists of k input-output queued switches, with FIFO queues, operating at the line speed in parallel, with k being independent of the number N of inputs and outputs. Arriving traffic is demultiplexed (spread) over the k identical switches, forwarded to the correct output, and multiplexed (combined) before departing from the parallel switch. We show that by using an appropriate demultiplexing strategy at the inputs and by applying the same matching in each of the k parallel switches, we guarantee a way for packets of a flow to be read in order from the output queues of the switches, thus eliminating the need for re-sequencing. Further, by allowing the switching algorithm to examine the state of only the first of the k parallel switches, we reduce considerably the amount of state information required. The switching algorithms that we develop are based on existing practical switching algorithms for input-output queued switches, and will have an additional communication complexity that is optimal up to a constant factor.

5.1 Motivation

As we have seen so far, most practical switching algorithms require a speedup of at least 2. This poses two non-trivial difficulties in moving towards higher speed switches:

- The first is that the memory within the switch must run at a speed faster than that of the external lines. This reduces memory access times, and makes it difficult to build practically useable memories, especially with continuously increasing line speeds.
- The second is that, with speedup, the time available to obtain a matching (by execution of the switching algorithm) is also reduced. This is particularly problematic for some of the more complex switching algorithms needed to provide

guarantees. Specifically, with a speedup of S, a switching algorithm has only $\frac{1}{S}$ time units to compute a matching.

Our approach, therefore, is to eliminate the need for speedup by using input-output queued switches in parallel. It is worth noting here that a previous work that addresses the use of parallel switches appears in [14]. Below we briefly point out some differences between our approach and the latter:

- In [14], the authors use parallel output queued switches, while we use parallel input-output queued switches, thus offering a different theoretical framework for the problem.

- The objective in [14] is to emulate output queuing for a switch operating at a high line speed by using a number of output queued switches operating in parallel at some sub-multiple of the line speed. Our objective is to provide basic guarantees, such as bounded delay on every packet, without requiring any speedup in the system.

- The algorithm in [14] relies on simulating an output queued switch in the background, which requires the maintenance of a large amount of state information. Our switching algorithms, on the other hand, are based on existing switching algorithms for an input-output queued switch that do not require an excessive amount of state information.

- The architecture in [14] naturally requires $2N$ parallel layers (where N is the size of the switch) of output queued switches to fully eliminate memory speedup in the system. This is because the queue memory of each switch is required to operate at a speed equal to $\frac{2RN}{k}$, where R is the line speed and k is the number of parallel switches. This dependence on N can be removed if input-output queued switches are used instead (4 of them). As a consequence, each input-output queued switch will then have to emulate an output queued switch. While such an emulation is possible as demonstrated in [7], it is not yet practical due to the excessive bookkeeping and communication needed between the switching

algorithm and the switches. Moreover, the emulation makes use of non-FIFO queues. Nevertheless, the emulation algorithm provided in [7] is practical at low speeds, suggesting that increasing the number of parallel input-output queued switches renders the algorithm practical. This however implies that the number of parallel switches needed has a dependence on the line speed, even if switches operating at the line speed are available. By contrast, to eliminate speedup, our architecture uses a constant number of layers that is independent of N.

- The bandwidth of the architecture in [14] is $2NR$ where R is the line speed. The bandwidth of our architecture is kNR. Therefore, for $k = 2$, which is sufficient to provide delay guarantees as will be seen later, both architectures have the same bandwidth. A more recent work [15] by the same authors of [14] illustrates an output queuing emulation up to an additive constant factor D using N output queues with no speedup. Hence, they reduce the bandwidth required to NR only. This however, requires re-sequencing of packets at the output. But since there is a bound D on the time a packet will be delayed from its output queuing time, re-sequencing can be eliminated by waiting a time D before delivering any packet at the output. The remaining disadvantage is that $D = 2kN$ where k is the number of switches, and hence the delay is $O(N^2)$.

Our main goal is not to emulate output queuing, as was done in [14] and [15]. Rather, it is to obtain an efficient and practical way of achieving basic guarantees, such as bounded delay on every packet, with a constant number of parallel layers, no speedup, and without the need to re-sequence packets at the output.

5.2 The Parallel Architecture

We use an architecture similar to the architecture described in [14]. The only difference between the architecture presented here and that of [14] is that we use input-output queued switches while the authors in [14] use output queued switches. The architecture is depicted in Figure 5-1.

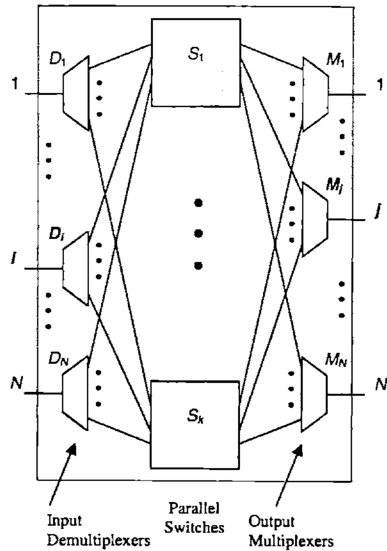


Figure 5-1: The parallel switches

The architecture consists of the N input ports having a demultiplexer each, and the N output ports having a multiplexer each. The middle stage consists of k switches in parallel, with each switch being an input-output queued switch, like the one depicted in Figure 1-5. At each input port i, a demultiplexer sends a packet arriving on that input to one of the k parallel switches. Likewise, at every output port j, a multiplexer accesses the output queue for that port (i.e. the jth output queue) in each of the k switches. Since no speedup is to be used, we define a time slot, as described at the beginnings of Chapter 1, to be the time needed for a packet to be read from or stored into a queue. Therefore, the switches operate in time slots where in each time slot, each switch can forward at most one packet from an input port and at most on packet to an output port. Although we assume that no speedup is being used, the switches of Figure 5-1 are input-output queued switches for the following reason: Since there is no speedup, an output port can deliver at most one packet per time slot; however, multiple packets can be forwarded to that output by multiple switches during a single time slot. Hence, forwarded packets need to be stored.

During each time slot, multiple packets may arrive at an input i provided each is destined to a different output j. The actual arrival pattern, of course, depends on the traffic model and on the specific implementation of the demultiplexers (for

instance each demultiplexer in Figure 5-1 can represent N actual demultiplexers for the different N flows at the input).

To proceed further, we define the following notation:

(i,j)	the flow (of packets) from input i to output j
$P(i,j)$	a packet from input i to output j
$Q(i,j)$	a packet from input i to output j
VOQ_{ij}^{l}	VOQ_{ij} in switch l
OQ_{j}^{l}	Output queue j in switch l

Unless otherwise mentioned, in the proofs that follow, we neither require any synchronization between packet arrivals and the operation of the parallel switches, nor do we require any synchronization between the k switches themselves, except that they all perform a matching by the end of a time slot. Our problem is to find a switching algorithm that provides delay guarantees while being efficient and practical to implement. The architecture in Figure 5-1 suggests the following natural decomposition of the switching algorithm:

- Demultiplexing: At every input i, deciding where to send each incoming packet.
- Switching: For each of the k parallel switches, deciding on a matching, i.e. which packets to forward across the switch.
- Multiplexing: At every output j, deciding which switch to read a packet from.

Before discussing the operation of this architecture, we describe why some simple approaches don't work.

5.2.1 Segmentation

The simplest approach one may consider is to segment each incoming packet into k segments, forward the segments in parallel across the switches, and reassemble the segments at the output. Unlike what one might think, however, this approach does not eliminate the need for speedup. This is because, each segment will now require

$(\frac{1}{k})^{th}$ the time of a complete packet, so a packet will have to be forwarded across the parallel switches in only $(\frac{1}{k})^{th}$ of a time slot. Thus, the time available for the switching algorithm also reduces by a factor of k, and k matchings will have to be computed per time slot.

5.2.2 Rate Splitting

Yet another approach could be to split a flow among the parallel switches to divide its rate equally among them. If the parallel switches are allowed to forward packets independently, however, it is difficult to control the order in which packets of the same flow emerge at the outputs of the switches. This can lead to either deadlock or output overloading with FIFO output queues as describe below. For instance, two packets that arrive at a given input, and are sent to two different switches, may experience different delays depending on the state of each switch, and thus may arrive at the output in the wrong order. Even though it appears that this could be circumvented by controlling the order in which the output queues are read (that is, by determining, at each time slot, the output queue containing the oldest packet of a flow and reading that packet), there could still be situations, such as the one depicted in Figure 5-2, where no output queue can be read without violating the order of packets.

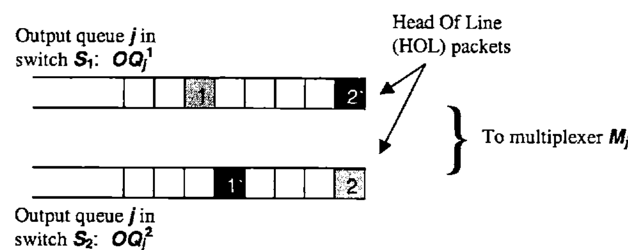


Figure 5-2: Possibility of deadlock at the output

In Figure 5-2, the packets at the head of output queue j in both parallel switches are the second packets of their respective flows. Thus, with FIFO output queues, it is not possible to deliver any packet at output j without violating the order of packets in a flow. Another solution could be to read the Head of Line (HOL) packets and temporarily store them to be delivered later. When the multiplexer has read deep

enough into the output queues to be able to reconstruct the correct order of packets in a flow, the HOL packets stored earlier can be released in the correct order. Clearly, if this happens often, time slots will be wasted without delivering packets at output j, causing the FIFO output queues to become overloaded and to grow indefinitely [25]. Of course, the above statement assumes that the output queues are FIFO and that a multiplexer cannot access more than one packet per time slot. The choice of the FIFO restriction is based on the ease of implementation of FIFO queues. Restricting the multiplexer to at most one access per time slot emanates from the need to have no speedup in any part of the parallel architecture. Both of these restrictions are reinforced by the fact the we do not allow for packet re-sequencing at the output.

Therefore, while on one hand our goal is to enable the switches to operate in a coordinated fashion, on the other it is to avoid excessive bookkeeping of the type needed in [14] to emulate output queuing.

5.2.3 Basic Idea

The key idea is to first avoid the type of deadlock depicted in Figure 5-2. Having achieved that, we focus later on how to provide the delay guarantees. We say that a packet P is older than a packet Q if P arrives before Q. In order to avoid the type of deadlock in Figure 5-2, we consider the following two properties.

Definition 5.1 (***output contention***) *In a single switch, two packets coming from different inputs and destined to the same output cannot be forwarded during the same time slot (by the property of a matching, this is trivial when the switch has no speedup).*

Definition 5.2 (***per-flow order***) *For any two packets P and Q of the same flow, if P is older than Q, then by the end of the time slot during which Q was forwarded, P would have been forwarded.*

We will show that the two properties above are sufficient to ensure that, at an output j, the packets of any flow (i, j) can be read in order. We begin by defining this order more formally. In doing so, we define a partial order relation that we denote

by $\prec_{FIFO}$. The partial order relation $\prec_{FIFO}$ is defined over all the packets that are residing at the output side of the switches. However, as it will be seen later from the definition of $\prec_{FIFO}$, some packets might be left unordered by $\prec_{FIFO}$. These are packets that are destined to different outputs or packets of different flows that are forwarded during the same time slot. We will define the order relation $\prec_{FIFO}$ in such a way that, if the *per-flow order* property is satisfied, it will induce the standard FIFO order on all packets pertaining to a single flow.

Definition 5.3 ($\prec_{FIFO}$) *For any two packets $P(i,j)$ and $Q(k,j)$ at the output side, $P(i,j) \prec_{FIFO} Q(k,j)$ if:*

- *the time slot during which $P(i,j)$ was forwarded precedes the time slot during which $Q(k,j)$ was forwarded, or*
- *$i = k$, $P(i,j)$ is older than $Q(k,j)$, and both were forwarded during the same time slot.*

Note that if $P(i,j) \prec_{FIFO} Q(i,j)$ and the *per-flow order* property is satisfied, then $P(i,j)$ is older than $Q(i,j)$. More precisely, we have the following lemma.

Lemma 5.1 *If the output contention and per-flow order properties are both satisfied, the following is true for every output j: At the end of a time slot, either OQ_j^l is empty for all l or there exists a flow (i,j) such that its oldest packet $P(i,j)$ is at the head of OQ_j^l for some l.*

Proof: If at the end of a time slot, OQ_j^l is empty for all l, the lemma is true. So assume that, at the end of a time slot, there is an l such that OQ_j^l is not empty. Since $\prec_{FIFO}$ is an order relation, there must exist an l and an i such that OQ_j^l contains a packet $P(i,j)$ with the following property: there is no packet $Q(k,j)$ at the output side satisfying $Q(k,j) \prec_{FIFO} P(i,j)$. We will prove that $P(i,j)$ is at the head of OQ_j^l and that $P(i,j)$ is the oldest packet of flow (i,j). We first prove that $P(i,j)$ is at the head of OQ_j^l. If a packet $Q(k,j)$ is ahead of $P(i,j)$ in OQ_j^l, then by the *output contention* property, $Q(k,j)$ was forwarded during a time slot prior to the time slot

during which $P(i,j)$ was forwarded. By the definition of $\prec_{FIFO}$, $Q(k,j) \prec_{FIFO} P(i,j)$ which is a contradiction. Next we prove that $P(i,j)$ is the oldest packet of flow (i,j). If this is not so, note that by the *per-flow order* property, the oldest packet of flow (i,j), $Q(i,j)$, must be at the output side, and in the worst case, must have been forwarded by the end of the time slot during which $P(i,j)$ was forwarded. By the definition of $\prec_{FIFO}$ and since $Q(i,j)$ is older than $P(i,j)$, $Q(i,j) \prec_{FIFO} P(i,j)$ and we reach a contradiction again. ■

The above lemma implies that for every flow (i,j), whenever there are packets in the output queues for output j, a packet can be delivered at output j without violating the order of packets pertaining to flow (i,j). Therefore, this eliminates the deadlock situation described earlier and prevents the output queues from being overloaded. The *output contention* property is trivially satisfied when the switches have no speedup. Therefore, we will design our switching algorithm to satisfy the *per-flow order* property.

5.3 The Approach

To specify our approach, we will describe how we carry out the three steps outlined in Section 5.2 (demultiplexing, forwarding, and multiplexing). As motivated earlier, we will design our switching algorithm to satisfy the *perflow order* property. We state the following definition that we need for the rest of the chapter.

Definition 5.4 (*k-parallel* switching) *k-parallel switching is one where, during each time slot, the switching algorithm computes only one matching, M, and applies it in all k parallel switches.*

We start by describing the demultiplexer operation.

5.3.1 Demultiplexer Operation

To distribute the incoming packets among the k parallel switches, the demultiplexer follows a special demultiplexing strategy, which we call *minimum length* demultiplex-

ing, as defined below:

Definition 5.5 (*minimum length* demultiplexing) *Demultiplexer D_i sends a packet destined for output j to a switch l with a minimum number of packets in VOQ^l_{ij} at the end of the time slot preceding the current time slot.*

We now prove that this strategy together with *k-parallel* switching ensures that the k oldest packets for each flow (i, j) are always in distinct switches. We start with a simple lemma.

Lemma 5.2 *If minimum length demultiplexing and k-parallel switching are used, then at the end of a time slot, the lengths of VOQ^l_{ij} and VOQ^s_{ij} differ by at most 1 for any two switches l and s.*

Proof: The proof is by induction on the number of time slots:

Base case: The lemma is trivially true at a fictitious time slot before the beginning of the first time slot.

Inductive step: Assuming that the lemma is true at the end of time slot T, we will prove that it holds at the end of time slot $T + 1$. We focus on any two VOQs, VOQ^l_{ij} and VOQ^s_{ij}, and we consider two cases:

Case 1: At the end of time slot T, both VOQs were non-empty. *k-parallel* switching during time slot $T+1$ will decrease the length of both VOQs by the same amount (by either 0 or 1). If no packet is sent to either one of the VOQs during time slot $T+1$, then the lemma holds at the end of time slot $T+1$. Otherwise, a packet is sent to one of the VOQs say VOQ^l_{ij}. By the *minimum length* demultiplexing, we know that at the end of time slot T , the length of VOQ^l_{ij} was at most that of VOQ^s_{ij}. Therefore, adding one packet to VOQ^l_{ij}will not violate the lemma.

Case 2: At the end of time slot T, at least one VOQ, say VOQ^l_{ij}, was empty. Then we know by the lemma that VOQ^s_{ij} must contain at at most one packet. If a packet $P(i, j)$ is sent during time slot $T + 1$ to either VOQ^l_{ij} or VOQ^s_{ij}, then by the

minimum length demultiplexing it must be sent to VOQ_{ij}^{l}. Therefore, at the end of time slot $T+1$, the length of both VOQs is at most 1, and the lemma holds. ■

Using Lemma 5.2, we can now prove the following lemma:

Lemma 5.3 *If minimum length demultiplexing and k-parallel switching are used, then for any flow, at the end of a time slot, either all packets at the input side are in distinct switches or the k oldest packets at the input side are in distinct switches.*

Proof: If at the end of a time slot T, there is some VOQ_{ij} that is empty, then by Lemma 5.2, VOQ_{ij}^{l} has length at most 1 for all l, and hence all packets at the input side are in distinct switches. If at the end of a time slot T , no VOQ_{ij} is empty, then for the k oldest packets at the input side not to be in distinct switches, it must be that some VOQ_{ij}, say VOQ_{ij}^{l}, contains two of the k oldest packets P_1 and P_2, and another VOQ_{ij}, say VOQ_{ij}^{s}, contains a packet P_3 that is not among the k oldest packets. Without loss of generality P_3 is the head of VOQ_{ij}^{s} by the end of time slot T . Let T_0 be the time slot during which P_3 arrived to VOQ_{ij}^{s}.

Consider the end of time slot $T_0 - 1$. Since only one packet $P(i,j)$, in this case P_3 , can arrive during time slot T_0 , we know that at the end of time slot $T_0 - 1$, both P_1 and P_2 were in VOQ_{ij}^{l}. Therefore, from the end of time slot $T_0 - 1$ till the end of time slot T, VOQ_{ij}^{l} was non-empty. Therefore, *k-parallel* switching implies that every time VOQ_{ij}^{s} was served by a matching, so was VOQ_{ij}^{l}. Since at the end of time slot T, P_3 is at the head of VOQ_{ij}^{s}, all the packets that were in VOQ_{ij}^{s} at the end of time slot $T_0 - 1$ must have been forwarded by the end of time slot T . This means that at least that many packets, excluding P_1 and P_2 , were also forwarded from VOQ_{ij}^{l}. Therefore, at the end of time slot $T_0 - 1$, the lengths of VOQ_{ij}^{l} and VOQ_{ij}^{s} differed by at least two, which contradicts Lemma 5.2. ■

Using Lemma 5.2 and Lemma 5.3, we prove the main result of this section:

Theorem 5.1 *If minimum length demultiplexing and k-parallel switching are used, then the per-flow order property is satisfied.*

Proof: Consider a flow (i,j) and a time slot T . If no packet $P(i,j)$ is forwarded during time slot T , then the *per-flow order* property for flow (i,j) cannot be violated

during time slot T. Assume a packet $P(i, j)$ is forwarded during time slot T. By Lemma 5.3, at the end of time slot $T-1$, either all packets of flow (i, j) were in distinct switches or the k oldest packets of flow (i, j) were in distinct switches. Therefore, *k-parallel* switching cannot violate the *per-flow order* property during time slot T. ■

As a consequence, we can now prove that using *minimum length* demultiplexing and *k-parallel* switching cannot create the deadlock situation illustrated in Section 5-2.

Corollary 5.1 *If minimum length demultiplexing and k-parallel switching are used, then for every output j, at the end of a time slot, either OQ_j^l is empty for all l or there exists a flow such that its oldest packet is at the head of OQ_j^l for some l.*

Proof: Since the *output contention* property is trivially satisfied, the corollary is immediate from Lemma 5.1 and Theorem 5.1. ■

The demultiplexers do not have to explicitly identify the *VOQ* with the minimum number of packets, as we can prove that each of the following strategies, when combined with *k-parallel* switching, is a *minimum length* demultiplexing.

Round Robin

In this strategy, each demultiplexer keeps N counters, one for each output. Each counter stores the identity of the switch to which a new packet for that output should be sent, and all counters start initially at 0. Every time the demultiplexer sends a packet for a particular output to the switch specified by the corresponding counter, it increments that counter modulo k. This has the nice property of dividing the rate of a flow equally among the k parallel switches. Moreover, as we will illustrate later in Section 5.3.4, this strategy will be useful for building a switch that supports a line speed that is k times the line speed of the individual switches.

Round Robin Reset

This strategy is the *Round Robin* strategy with a slight variation. For every flow (i, j), the system keeps track of the number of packets of that flow that are still residing at the input side of the switches. Whenever this number becomes zero, the counter

at demultiplexer D_i that corresponds to output j is reset to zero. This strategy requires some extra information (to be kept either by the switching algorithm or by the demultiplexers) to correctly reset the counters of the demultiplexers. Moreover, it might require some synchronization between packet arrivals and the time slots of the switch. As will be seen later however, this strategy will actually allow the switching algorithm to keep less information for coordinating the operation of multiplexers at the output ports, and, in some cases, it also helps to reduce the amount of state information that the switching algorithm must consider for computing a matching.

Lemma 5.4 *If k-parallel switching is used, then Round Robin demultiplexing is a minimum length demultiplexing.*

Proof: We will prove that for any flow (i, j), by the end of a time slot T, either VOQ_{ij}s in all switches have the same length, or starting from a switch, we can find a round robin order on the switches, S_1 to S_k, such that there exists $0 < l < k$, such that VOQ_{ij}^l is the last VOQ_{ij} that received a packet $P(i, j)$ by the end of time slot T, the length of any VOQ_{ij}^s for $l < s \leq k$ is L, and the length of any VOQ_{ij}^s for $0 < s \leq l$ is $L + 1$. Note that proving the above claim proves the lemma since the next time slot a packet $P(i, j)$ arrives, it will be sent to a VOQ_{ij} with the minimum number of packets, either because VOQ_{ij}s in all switches had the same length at the end of time slot T, or because the packet is sent to $VOQ_{ij}^{(l+1)}$ by *Round Robin* demultiplexing, which has a minimum number of packets. We prove the above claim by induction on the number of time slots.

Base case: The claim is trivially true at a fictitious time slot before the beginning of the first time slot since VOQ_{ij}s in all switches have the same length.

Inductive step: The claim is true up to time slot T. We will prove that it remains true for time slot $T+1$. We are not going to consider the interleaving in the operations of applying the matching and sending a packet to some VOQ. But one can show that this interleaving has no effect on the reasoning below.

If VOQ_{ij}s in all switches had the same length by the end of time slot T (or after the arrival of a packet during time slot $T+1$), *k-parallel* switching implies that they will have the same length by the end of time slot $T+1$. Moreover, by *k-parallel* switching, if a packet is forwarded from a VOQ_{ij}^s for $s > l$, a packet will be forwarded from a VOQ_{ij}^s for $s \leq l$. Therefore, the above claim will still be true after applying the matching.

If a packet $P(i, j)$ arrives during time slot $T+1$ and VOQ_{ij}s in all switches had the same length by the end of time slot T (or after applying the matching during time slot $T+1$), then if $P(i, j)$ is sent to some VOQ_{ij}^s (which will have the maximum number of packets by the end of time slot $T+1$), we set the order S_1 to S_k such that $S_1 = S_s$ and we make $l = 1$.

If a packet $P(i, j)$ arrives during time slot $T+1$ and by the end of time slot T (or after applying the matching during time slot $T+1$), we had the order S_1 to S_k with some $l < k-1$, then we keep the same order and increment l by one. If $l = k-1$, then by the end of time slot $T+1$, VOQ_{ij}s in all switches will have the same length since $P(i, j)$ will be sent to S_k.

■

A similar proof can be constructed for *Round Robin Reset* since *Round Robin Reset* acts exactly like *Round Robin*, except that it resets the round robin order for a flow whenever all packets of that flow have been forwarded. In the interval between two successive resets, therefore, *Round Robin Reset* behaves exactly like *Round Robin* and hence satisfies *minimum length* demultiplexing.

5.3.2 Switching Operation

We showed how *minimum length* demultiplexing together with *k-parallel* switching can satisfy the *per-flow order* property, which (with the *output contention* property) ensures that, for every output j, it is possible to read a packet (if one is available) without violating the order of packets within a flow. In Section 5.3.3, we explain how, during each time slot, the multiplexer may identify the appropriate queue to read from. Our focus here is to consider how a matching M may be computed to

achieve a bounded delay on every packet. We turn our attention first to a class of switching algorithms for the single switch setting that we call *k-serial* switching.

Definition 5.6 (*k-serial* switching) *In a single switch setting, k-serial switching is one in which the switching algorithm applies a given matching, M, consecutively k times before computing and applying a new matching.*

Our intention is then to show that any *k-serial* switching algorithm with a particular speedup can be emulated by a combination of *minimum length* demultiplexing and some *k'-parallel* switching algorithm, where we define emulation as follows:

Definition 5.7 (emulation) *If, using a k-serial switching algorithm, a packet P is forwarded across the single switch during a time slot T, then using minimum length demultiplexing and some k'-parallel switching algorithm, the same packet would also have been forwarded across one of the k' parallel switches by the end of time slot T.*

In what follows, we will assume that packets arrive only at the beginning of a time slot. This requirement can be realized by delaying an incoming packet until the beginning of the next time slot, which increases the packet delay by at most one time slot.

We first state the following simple lemma:

Lemma 5.5 *For any real number S, if minimum length demultiplexing and $\lceil S \rceil$-parallel switching are used, and packet arrivals occur only at the beginning of a time slot, then if M is the matching computed in time slot T and $(i, j) \in M$, then either all the packets of flow (i, j) or the $\lceil S \rceil$ oldest packets of flow (i, j) are forwarded by the end of time slot T.*

Proof: If at the end of time slot $T - 1$, at least $\lceil S \rceil$ packets of flow (i, j) are at the input side, then the result is true by Lemma 5.3 applied at the end of time slot $T - 1$. If at the end of time slot $T - 1$, less than $\lceil S \rceil$ packets of flow (i, j) are at the input side, then assume, without loss of generality, that a packet $P(i, j)$ arrives at the beginning of time slot T. By Lemma 5.3 applied at the end of time slot $T - 1$, and

by *minimum length* demultiplexing, $P(i,j)$ will be sent to an empty VOQ_{ij}. Thus at the beginning of time slot T, there is at most $\lceil S \rceil$ packets of flow (i,j) at the input side, each being in a separate VOQ by Lemma 5.3. Therefore, $\lceil S \rceil$-*parallel* switching implies that all packets of flow (i,j) will be forwarded by the end of time slot T. ■

Using Lemma 5, we can prove the following theorem:

Theorem 5.2 *If packet arrivals occur only at the beginning of a time slot, then any k-serial switching algorithm under a fractional speedup* $S = \frac{k}{c}$ *can be emulated using minimum length demultiplexing and an* $\lceil S \rceil$*-parallel switching algorithm.*

Proof: Every c time slots, the *k-serial* switching algorithm has exactly k matching phases, during all of which a matching M is kept constant. The $\lceil S \rceil$-*parallel* switching algorithm will run the *k-serial* algorithm in the background, and in doing so, it will compute the same matching M every c time slots. We will prove the theorem by induction on the number of time slots.

Base case: The theorem is trivially true at a fictitious time slot before the beginning of the first time slot.

Inductive step: By the end of time slot T, all packets that were forwarded by the *k-serial* algorithm were also forwarded by the $\lceil S \rceil$-*parallel* algorithm. Consider time slot $T+1$. Since the *k-serial* switching algorithm can have at most $\lceil S \rceil$ matching phases in every time slot (speedup of S), this implies that if $(i,j) \in M$, then the number of packets of flow (i,j) that are going to be forwarded during time slot $T+1$ by the *k-serial* algorithm cannot be more than $\lceil S \rceil$. By Lemma 5.5, if $(i,j) \in M$, then either all the packets of flow (i,j) or the $\lceil S \rceil$ oldest packets of flow (i,j) are forwarded during time slot $T+1$ by the $\lceil S \rceil$-*parallel* algorithm. Therefore, if a packet $P(i,j)$ is forwarded during time slot $T+1$ by the *k-serial* algorithm, and had not been forwarded by the $\lceil S \rceil$-*parallel* algorithm by the end of time slot T, then it must be among the packets that will be forwarded during time slot $T+1$. ■

Note that if S is an integer (which we can always assume to be true), then the $\lceil S \rceil$-*parallel* switching algorithm is an *k-parallel* switching algorithm because $S =$

$\lceil S \rceil = k$. In this case, as a practical consideration, the *k-parallel* switching algorithm does not need to run the *k-serial* switching algorithm in the background. Instead, it reconstructs the state of the single switch from the k parallel switches. This is possible since in every time slot both switching algorithms apply the same matching M an equal number of times k (one in parallel and the other sequentially); therefore, by the end of a time slot, the same packets which are remaining at the input side in the single switch, are also remaining at the input side in the k parallel switches. This reconstruction of the exact state of the single switch requires also that the same packets are being read from the output queues in every time slot by both algorithms (FIFO order). We know from Lemma 5.1 that this is possible (since *k-parallel* switching means that packets that are forwarded to the same output during a single time slot pertain to the same flow, and hence all packets at an output are ordered by the $\prec_{FIFO}$ relation). Reconstructing the single switch from the k parallel switches, however, implies that the switching algorithm has to look at a large amount of state. At the end of this section, we will suggest a way to reduce the amount of state information that the *k-parallel* switching algorithm has to look at; namely, we will consider looking only at the state of the first switch.

Note also that since the $\lceil S \rceil$*-parallel* switching can exactly mimic the *k-serial* ($S = k$) algorithm when S is an integer, it can provide the exact same guarantees as the *k-serial* switching algorithm.

Below, we state some loose delay bounds that the emulation guarantees for every packet under a weak constant burst traffic. We define the arbitration delay of a packet as the time the packet remains in its VOQ. The following theorem states that if each output emulates a global FIFO queue, emulating a *k-serial* switching algorithm that guarantees a packet arbitration delay will also result in guaranteeing a total packet delay.

Theorem 5.3 *If a k-serial switching algorithm under a weak constant burst traffic and a fractional speedup* $S = \frac{k}{c}$ *guarantees a packet arbitration delay* D_A*, then emulating that switching algorithm using minimum length demultiplexing and an* $\lceil S \rceil$*-parallel switching algorithm achieves a bounded delay of* $(\lceil S \rceil + 1)D_A + B$ *on every packet,*

where B is the traffic burst constant, provided that every output reads the packets in the $\prec_{FIFO}$ order.

Proof: By Theorem 5.2, we know that the $\lceil S \rceil$-*parallel* switching algorithm will guarantee a packet arbitration delay D_A. Therefore, at the end of a time slot, the number of packets destined to an output j that are still at the input side cannot exceed D_A in any of the $\lceil S \rceil$ switches. Otherwise, at least one packet will be delayed for more than D_A time slots at its input, implying that its arbitration delay will be greater than D_A. Consequently, at the end of a time slot, the number of packets destined to output j that are still at the input side in all $\lceil S \rceil$ switches is at most $\lceil S \rceil D_A$. By Corollary 1, if there are packets waiting in some output queue OQ_j^l, then it is possible to deliver a packet at output j without violating the packet order of any flow. Therefore, as long as some OQ_j^l is not empty, output j delivers a packet. Consider a time slot T in which some OQ_j^l becomes non-empty. At the end of time slot $T-1$, the number of packets destined to output j that reside at the input side is at most $\lceil S \rceil D_A$ as argued above. If during t time slots starting from time slot T, some OQ_j^s is non-empty, then by the end of the t time slots, output j will have delivered t packets. However, during the t time slots, the total number of packets that could have been forwarded to some output queue of port j is at most $\lceil S \rceil D_A + t + B$; since at most $t + B$ packets destined to output j could have arrived during the t time slots, by the property of the weak constant burst traffic. This means that the total number of packets that remain in the output queues of port j after the t time slots is at most $\lceil S \rceil D_A + B$. This is true for any t; therefore, at the end of a time slot, the number of packets in all output queues of port j is at most $\lceil S \rceil D_A + B$. As a result, since the output emulates a FIFO queue (with an $\lceil S \rceil$-*parallel* switching algorithm, all packets at a particular output are ordered by $\prec_{FIFO}$), once a packet arrives at the output side, it will be delivered within at most $\lceil S \rceil D_A + B$ time slots, hence achieving a bounded delay of $(\lceil S \rceil + 1)D_A + B$ on every packet. ■

If *Round Robin* demultiplexing is used, then to achieve a bounded delay on every packet, we need not restrict the output to read packets in a global FIFO order. We only require that the output read packets of the same flow in order, which is a

requirement we have imposed throughout the chapter.

Theorem 5.4 *If a k-serial switching algorithm under a weak constant burst traffic and a fractional speedup* $S = \frac{k}{c}$ *guarantees a packet arbitration delay* D_A*, then emulating that switching algorithm using Round Robin demultiplexing and an* $\lceil S \rceil$*-parallel switching algorithm achieves a bounded delay of* $(\lceil S \rceil + 1)D_A + B + (\lceil S \rceil - 1)(N - 1)$ *on every packet, where* B *is the traffic burst constant and* N *is the size of the switch, provided every output reads packets of the same flow in order.*

Proof: The proof is similar to the proof of Theorem 5.3. We use the fact that at the end of a time slot, the number of packets in all output queues of port j is at most $\lceil S \rceil D_A + B$. Assume a packet $P(i, j)$ remains in OQ_j^l for at least $\lceil S \rceil D_A + B + (\lceil S \rceil - 1)(N - 1) + 1$ time slots. By the end of the time slot during which $P(i, j)$ was forwarded, OQ_j^l contained at most $\lceil S \rceil D_A + B$ packets including $P(i, j)$. Therefore, at least $(\lceil S \rceil - 1)(N - 1) + 1$ packets, that were forwarded after $P(i, j)$, were delivered at output j before $P(i, j)$. These packets cannot pertain to flow (i, j) since packets of a flow are delivered in order. Therefore at most $N - 1$ flows can contribute to these packets. As a consequence, there exists a flow for which at least $\lceil S \rceil$ packets were forwarded after $P(i, j)$ and delivered at output j before $P(i, j)$. Since *Round Robin* demultiplexing is used and the output reads packets of the same flow in order, it must be that one of these packets, say Q, was in OQ_j^l. But OQ_j^l is a FIFO queue and Q was forwarded to it after $P(i, j)$. Therefore, Q could not have been delivered at output j while $P(i, j)$ remains in OQ_j^l. ■

It remains for us to show the existence of *k-serial* switching algorithms that guarantee a packet arbitration delay under some speedup $S = \frac{k}{c}$. We will modify some existing switching algorithms that guarantee packet arbitration delay under some speedup S to make them *k-serial* switching algorithms, for any integer k.

Some *k-serial* Switching Algorithms

In order to obtain *k-serial* switching algorithms, we convert existing switching algorithms for a single switch into *k-serial* switching algorithms, by simply modifying the

existing algorithms to hold the matching that they compute constant for k times [1]. Our motivation is that the state of the switch cannot change substantially within a constant time $\frac{k}{S}$. Thus, holding the same matching for k times should possibly still be able to guarantee a packet arbitration delay.

We were able to prove this fact for several existing switching algorithms, such as the *Earliest Activation Time* switching algorithm described in Chapter 3, the *Oldest Cell First* algorithm due to Charny et al [6], the *Central Queue* algorithm due to Kam et al [16], and the *Delayed Maximal Matching* algorithm, an algorithm that we describe here in order to illustrate the point further.

Earliest Activation Time

This algorithm was presented in Chapter 3 and is a priority switching algorithm with the *Earliest Activation Time* priority scheme. Recall that the *Earliest Activation Time* priority scheme assigns higher priority to active (see Definition 2.2) VOQs with earlier activation times (see Definition 3.2). Therefore during a matching phase, if an active VOQ_{ij} is not served by the matching, either some active VOQ_{ik} with an activation time no later than that of VOQ_{ij} is served, or some active VOQ_{kj} with an activation time no later than that of VOQ_{ij} is served. Hence, holding a matching M for k times starts to violate the above property for VOQ_{ij} only when some VOQ_{ik} (or some VOQ_{kj}) becomes inactive while being served by the matching M. The above property will be violated at most $k-1$ times by a VOQ_{ik} (or a VOQ_{kj}) while VOQ_{ij} remains active, since once a VOQ becomes inactive, its activation time will be more recent than that of VOQ_{ij} when it becomes active again; and this will be reflected by the priority scheme when the matching is recomputed after k time slots. Therefore, holding the matching constant for k times will violate the above property for VOQ_{ij} at most $2(k-1)(N-1)$ times while VOQ_{ij} is active (there are $2(N-1)$ VOQs that share either an input or an output with VOQ_{ij}). With a speedup $S = 2$, we can prove that this modified algorithm still guarantees a bounded delay on every packet

[1]A similar idea was suggested in [29] where a random matching is computed in every matching phase but the matching is used only if it is better (in some sense) than the last used matching. Therefore, a matching might be held for a while before applying another matching.

when $\alpha < 1$ under a strong constant burst traffic (see Theorem 4.2), where α is the loading of the switch. We can show that the additional delay to the original delay is $\frac{2(k-1)(N-1)}{S-2\alpha}$.

Oldest Cell First

This algorithm is due to Charny et al. [6] and is also a priority switching algorithm. The priority scheme used by this algorithm assigns higher priority to the VOQs holding older packets. Therefore, in every matching phase, the oldest packet that can still be forwarded is chosen. This is repeated until a maximal matching is obtained. This algorithm guarantees a bounded delay on every packet with $S = 2$ when $\alpha < 1$ under a weak constant burst traffic (see [6]), where α is the loading of the switch. The priority scheme of this algorithms guarantees that if a packet $P(i, j)$ is not forwarded, then either a older packet $Q(i, k)$ is forwarded, or an older packet $Q(k, j)$ is forwarded. With an argument similar to the one made above for the *Earliest Activation Time* switching algorithm, we can prove that holding the matching k times can violate the above property for packet $P(i, j)$ only a bounded number of times, $2(k-1)(N-1)$. This is enough for the algorithm to still provide a delay guarantee (see [6]). The additional delay to the original delay will be $\frac{2(k-1)(N-1)}{S-2\alpha}$.

Central Queue

This algorithm is due to Kam et al. [16]. We will now describe the algorithm as it was originally presented in [16] (which is different than how it was presented in Chapters 1 and 2). The algorithm works by assigning credits to each VOQ_{ij} based of the rate of flow (i, j). A packet is admitted if it has credit. The credit is decremented by 1 whenever a packet is forwarded. The credit of a non-empty VOQ_{ij} represent the weight of (i, j). In every matching phase, the algorithm computes a $\frac{1}{2}$-approximation of the maximum weighted matching, by repeatedly picking (i, j) with the largest weight until a maximal matching is obtained. This was proved to guarantee strong throughput under no speedup when the credit rate at each input and output is less then $\frac{1}{2}$. As argued in [16], when a flow (i, j) is constantly backlogged, a bounded VOQ length L implies a bounded packet arbitration delay $\frac{L}{g_{ij}}$, where g_{ij} is the credit

rate of flow (i,j). Using the same techniques in [16], one can prove that, with a speedup of 2 and a credit rate less than one, this algorithm also guarantees strong throughput.

During a constant time, the change in the credit assigned to a VOQ is bounded. Therefore, the change in the total weight of the maximum weighted matching is also bounded. Our matching, being a $\frac{1}{2}$-approximation of the maximum weighted matching when first computed, when held for k times, cannot differ from the half weight of the maximum weighted matching by more than a certain bound. A problem arises, however, if a VOQ_{ij} with a large credit suddenly becomes empty. In that case, (i,j) is still considered part of the matching, while the matching is being held constant, and is contributing a large weight to the matching. However, that weight should not be counted in the matching because flow (i,j) is idle and no packets of flow (i,j) are being forwarded. Therefore, the weight of the real maximum weighted matching at that time might differ from the weight of the maximum weighted matching when our matching was computed, by as much as the credit of VOQ_{ij}. If flow (i,j) is constantly backlogged however, when it becomes idle, the credit of VOQ_{ij} can be bounded. As a consequence, when all flows are constantly backlogged, the difference between the weight of the matching and the half weight of the maximum weighted matching is bounded at all times (the bound is $O(kN)$), and this is all what we need to keep the proof working (see [16]). Therefore, the VOQ length will still be bounded and a delay guarantee will be achieved when all flows are constantly backlogged. Note that in order to satisfy the requirement of Theorem 5.2, namely that a packet arrival occurs only at the beginning of a time slot, delaying an incoming packet until the next time slot does not violate the condition that a flow is constantly backlogged.

Delayed Maximal Matching

This is a simple algorithm that we present to illustrate further the idea of holding the matching for a constant number of times. The switching algorithm waits for a time T until enough packets have accumulated in the VOQs. Then it forwards those packets in an interval of time T using successive arbitrary maximal matchings. During that

interval of time, another set of packets would have accumulated, and the algorithm repeats. Therefore, the arbitration delay is T. One way of achieving this with a weak constant burst traffic is the following. The switching algorithm builds an $N \times N$ matrix A where a_{ij} represents the number of packets that arrived from input i to output j. The algorithm waits a time $T \geq \frac{B}{1-\alpha}$ where α is the loading of the switch and B is the traffic burst constant. Since the number of packets that arrive from an input or to an output during an interval of time T is at most $\alpha T + B$ (property of the weak constant burst traffic), the sum of entries of any row and any colon in the matrix A will be at most T. In that case, it can be shown that the switching algorithm can forward those packets in at most $2T$ maximal matchings. Therefore, with a speedup of $S = 2$, this is done in at most T time slots. By that time, another matrix would have been computed and the same process is repeated again.

If we hold the matching for k times, every VOQ_{ij} will be served at most $k-1$ times while its a_{ij} is zero. We can show that this implies that the algorithm will need an extra $2(k-1)(N-1)$ matchings (or equivalently $(k-1)(N-1)$ time slots with a speedup $S = 2$) to forward the packets during the interval of time T. For the process to work as before, we require that $\alpha T + B \leq T - (k-1)(N-1)$ or $T \geq \frac{B+(k-1)(N-1)}{1-\alpha}$, which adds an extra delay of $\frac{(k-1)(N-1)}{1-\alpha}$.

Birkhoff-von Neumann Decomposition: A *k-parallel* Switching Algorithm

Chang et al. [4] (see also [5]) have proposed an algorithm that is capable of providing delay guarantees for input queued switches with no speedup. The algorithm consists of taking a static rate matrix and computing only once a static schedule in time $O(N^{4.5})$, based on a decomposition result of Birkhoff and von Neumann. The schedule is a static list of matchings, corresponding to permutation matrices obtained from the decomposition of the rate matrix, and applied according to certain weights. In our context, we may utilize this algorithm in conjunction with *Round Robin* demultiplexing which ensures identical rate matrices for all switches. Using this algorithm, k static schedules can be obtained based on the individual rate matrices. These static schedules will be identical since all switches will have the same rate matrix. Thus, as

a natural consequence of this approach, the same matching will be applied in every time slot in all of the parallel switches. For each individual switch, this provides comparable arbitration delay guarantees as the original algorithm of Chang, with the added advantage, that we can sustain a line speed that is now k times the speed at which the parallel switches operate. Note however, that since each switch is now running at a slower speed, it is not possible to transmit packets at the line speed between the inputs and the switches and between the switches and the outputs. However, the technique described in [15] of buffering packets at the demultiplexers and multiplexers can be utilized, causing only a small additive delay. This will be discussed with more detail in Section 5.3.4.

Reducing State Information

It can be shown that when the speedup $S = k$ is an integer, the *k-parallel* switching algorithm can reconstruct, from the state of all the k parallel switches, the state of the single switch running the *k-serial* switching algorithm. This requires, however, that the scheduler examine the state of each of the k parallel switches, and maintain a global state. It turns out that this global state requirement can actually be relaxed. For the single switch switching algorithms discussed above, only three kinds of state information are used: the activation time for the *Earliest Activation Time* switching algorithm, the oldest packet of each *VOQ* for *Oldest Cell First*, and the length of each *VOQ* for *Central Queue* [2] and *Delayed Maximal Matching.*

For the *Earliest Activation Time* switching algorithm, the only state needed is whether a *VOQ* is active or not (see Chapter 3). This can be done by communicating active and inactive *VOQ*s from the input ports to the switching algorithm as described in Chapter 3, and no extra state information will be needed.

By using *Round Robin Reset* demultiplexing, the amount of state information needed can be greatly reduced for the *Oldest Cell First.* For instance, it ensures that

[2]Here we say the length of a *VOQ* instead of its credit because when a strong constant burst traffic is constantly backlogged, the length of a *VOQ* differs from its credit by at most a constant. Alternatively, the *Central Queue* algorithm can use the credit of a *VOQ* and no other state information will be needed. But then, explicit knowledge of the rates is required.

the oldest packet of every flow is always in the first switch. Thus, when using *Oldest Cell First* and *Round Robin Reset*, the algorithm needs only look at the state of the first switch to compute a matching.

For the *Central Queue* algorithm, the use of any *minimum length* demultiplexing ensures that, for every flow (i, j), the number L of all the packets at the input side is related to the number of packets L_1 in VOQ^1_{ij} in the following way:

$$kL_1 - k < L < kL_1 + k$$

Thus, if kL_1 is used as an approximation to L, the computation of the $\frac{1}{2}$-approximation of the maximum weighted matching will be affected by at most a certain bound, which, as argued previously, will not hurt the delay guarantees for the *Central Queue* algorithm. Note that we are now using lengths of VOQs as the weights and not the credits (see footnote 2).

For the *Delayed Maximal Matching* algorithm, defining similarly a_{ij1} and using the upper bound $ka_{ij1} + k - 1$ as an approximation for a_{ij}, will result in serving a VOQ at most an additional bounded number of times while it is empty; a phenomenon that can be accommodated for in the same way described earlier for the *k-serial* version of the *Delayed Maximal Matching* algorithm, i.e. by increasing the delay after which the algorithm obtains a new matrix. The upper bound $ka_{ij1} + k - 1$ is used here because the algorithm needs to make sure that it is emptying all the matrix as described earlier.

Observe that state reduction is not an issue for the Birkhoff-von Neumann decomposition algorithm, because it only stores a precomputed schedule and so does not require any state information from the switches for its operation.

5.3.3 Multiplexer Operation

We have already shown that when using *minimum length* demultiplexing and *k-parallel* switching, it is possible for the multiplexer at an output port to always deliver a packet from the output queues of the k parallel switches in a way not to violate the order of

packets pertaining to the same flow. The only question that remains is how a multiplexer M_j determines which output queue OQ_j^l to read the next packet from? This can be done in different ways. One way is to use a standard re-sequencing technique. Packets are tagged upon arrival to the switch with their arrival times. At the output side, the multiplexer incrementally sorts the tags of the head of line (HOL) packets and chooses to read the one with the smallest tag. This requires additional access to the output queues which we assume not possible given that no speedup is available; especially since the tag value itself can grow as large as the total delay of a packet.

An alternative is for the switching algorithm to store this information and sort the head of line packets of all the queues. This, however, requires the communication of tags between the demultiplexers and the switching algorithm every time packets arrive. In addition, to avoid the use of unbounded tags, both of these approaches must address the issue of tag reuse.

We would like to avoid the use of the above re-sequencing techniques. A more efficient approach that uses *Round Robin* or *Round Robin Reset* demultiplexing is the following. For each output j, the switching algorithm maintains a FIFO list L_j of tuples of the form (p, s) pertaining to successive time slots during which a packet was forwarded to output j. Hence, for every such time slot, p is the number of packets switched to output j during that time slot, and s is the index of the switch that forwarded the oldest packet to output j during that time slot (note that all packets switched to output j during that time slot pertain to the same flow).

Therefore, during each time slot for which some (i, j) belongs to the matching, the algorithm adds a (p, s) to L_j. The algorithm may easily obtain the information to do so from the demultiplexers. Each demultiplexer D_i stores the number of packets for a particular output that have arrived up to the current time slot and are still remaining at the input side.

Upon applying a matching M, the switching algorithm communicates to demultiplexer D_i the index j of the output for which $(i, j) \in M$. The demultiplexer responds with the number of packets that will be forwarded to output j as a result of applying M (this is easy to determine since it is either all the packets or k packets by Lemma

5.5), and the index of the switch that contains the oldest such packet (also easy to determine with any of the two round robin demultiplexing strategies described earlier). The total communication required between the demultiplexers and the switching algorithm is therefore $O(N \log N + 2N \log k)$.

Following this, demultiplexer D_i updates for every output j the number of packets of flow (i, j) remaining on the input side as well as the index of the switch that now contains the oldest packet of flow (i, j). At the output side, each multiplexer M_j periodically retrieves from the switching algorithm a tuple (p, s) from which it learns the number of packets that must be read and the identity s of the switch from whose output queue the multiplexer must start reading the first packet of this round, and continues in a round robin fashion (as a consequence of the round robin demultiplexing). Therefore, the communication between the switching algorithm and the multiplexers is $O(2N \log k)$. Hence, the total communication with the switching algorithm is $O(N \log kN)$, which is within a constant factor of the $\Omega(N \log N)$ amount of communication needed for the switching algorithm to specify a matching in a single switch.

If we use *Round Robin Reset* demultiplexing, then we know that the oldest packet of a flow is always in the first switch and therefore s is not needed.

Instead of requiring additional memory for the switching algorithm, we can use the memory of the switch itself, i.e. the output queues, in order to store the required information. This works for the case of *Round Robin Reset* demultiplexing in the following way: Since the oldest packet of a flow is in the first switch, we only need to tag a packet $P(i, j)$ that is forwarded across the first switch with the number p of packets of flow (i, j) that are going to be forwarded during the current time slot. At the output, the multiplexer retrieves this number when reading the packet in the first switch, and hence it knows how many packets to read before coming back to the first switch. This is efficient in terms of space since the tag length is $O(\log k)$ and only packets in the first switch need to be tagged. A difficulty with this approach is that we must tag packets upon forwarding them, which might not be straight forward to realize.

5.3.4 Supporting Higher Line Speeds

We now briefly describe how we can use parallel switches that run at a speed slower than the line speed. For this purpose, we assume that the line speed is some integer multiple, m, of the speed of a single switch.

The first thing to note is that each time slot of a switch is now m times the original time slot of the traffic (since the switches are m times slower). Thus, we will refer to the time slots of the traffic by external time slots, reserving the term time slots to denote the internal time slots of the switches.

The second thing to note is that now, a demultiplexer will not be able to send packets to a single switch in successive external time slots, since each link can be accessed only once every time slot, i.e. once every m external time slots. Similarly, a multiplexer can access output queue OQ_j^l for output j once every m external time slots.

We will assume the use of the *Round Robin* demultiplexing strategy. Assume also that the number of parallel switches is m. We will describe how we can use the links that are running at $\frac{1}{m}$ the original line speed.

We will use m FIFO buffers running at the line speed in each demultiplexer and multiplexer. Each of the m FIFO buffers corresponds to one of the m switches. When a packet needs to be sent by demultiplexer D_i from input i to a switch, it is stored in a buffer of D_i corresponding to that switch. The packet at the head of the buffer is sent to the switch when the link is available. When only one packet can arrive at an input during a single external time slot, an analysis of this technique appears in [15] and illustrates that a buffer size of N is enough for each buffer of the demultiplexer. Moreover, each packet will be delayed at most N time slots (i.e. mN external time slots) at the input. Therefore, we can consider a new arrival pattern of packets at the input, where at a given time slot, only packets that arrived N time slots prior to the current time slot are considered present. This produces the original arrival pattern of packets delayed by mN external time slots.

Similarly, when a packet needs to be delivered at output j by multiplexer M_j, it

is stored, when the link is available, in a buffer in M_j corresponding to the switch being used to deliver that packet. The packet remains in that buffer until it can be delivered. Therefore, the m buffers of the multiplexer act as a re-sequencing buffer. As before, the analysis described in [15] yields a buffer size of N for each of the m buffers of M_j . Moreover, each packet will be delayed at most N time slots (i.e. mN external time slots) at the output. Therefore, by waiting mN external time slots at the output, the same techniques for delivering packets described in the previous section are still valid, hence making re-sequencing a simple operation.

In general, for a weak constant burst traffic with burst constant B, the buffer size will be $N + \lceil \frac{B}{m} \rceil$ and the the delay of mN external time slots will be replaced by $mN + B$.

The above buffering technique solves the problem of slow links with an additive delay of $2(mN + B)$. We now illustrate that with these m slow switches, we can still somehow emulate an *m-serial* switching algorithm running at the original line speed.

We consider the new arrival pattern at the input, which is the exact original arrival pattern delayed by $mN + B$ external time slots.

The idea is similar to what Lemma 5.5 and Theorem 5.2 achieve. As before, the *m-serial* switching algorithm holds a matching M for m external time slots, which is equal to one time slot of the *m-parallel* switching algorithm. First we note that *minimum length* demultiplexing operates in every external time slot now as opposed to every internal time slot. Therefore, the number of packets in a VOQ at the end of an external time slot might not be accurately defined since a matching requires m external time slots (one time slot) to complete. Conceptually however, we can think of the matching taking effect only during the last external time slot of a time slot. Hence, *minimum length* demultiplexing reflects the correct number of packets in the VOQs as viewed by the demultiplexers in each external time slot. Since in our setting, *Round Robin* demultiplexing is a *minimum length* demultiplexing as proved earlier, and since *Round Robin* demultiplexing does not rely on the number of packets in the VOQs, regardless of how the matching is carried during a time slot, we will still have the same results as before. Namely, Lemma 5.5 will still be true, and hence if (i, j)

is in the matching M , then either all packets of flow (i,j) or the m oldest packets of flow (i,j) will be forwarded by the end of a time slot. With a proof similar to the one for Theorem 5.2, and since the *m-serial* algorithm can forward at most m packets every m external time slots, we conclude that the *m-parallel* algorithm emulates the *m-serial* algorithm up to an additive constant $m-1$; the reason being that a packet that is forwarded with the *m-parallel* algorithm during a time slot T might have been forwarded by the *m-serial* algorithm during any of the m external time slots that correspond to time slot T. Therefore, if the *m-serial* switching algorithm guarantees an arbitration delay D_A external time slots, the *m-parallel* switching algorithm will guarantee an arbitration delay of $D_A + m - 1 + mN + B = D_A + m(N+1) + B - 1$ external time slots.

5.4 Summary

We suggested a scheme that eliminates the need for speedup by using $k = \lceil S \rceil$ parallel input-output queued switches with no speedup, where S is the speedup of the original switch. The key to our approach was to apply the same matching in all the parallel switches. By adapting existing switching algorithms for the single switch setting to hold their matching constant for a number of times, we were able to apply the same matching in all switches, and guarantee a bounded delay on every packet. In addition, both demultiplexing and multiplexing at the inputs and outputs, respectively, could be done using $O(N \log kN)$ amount of communication between the switching algorithm and the parallel switches. This is to be compared to the $\Omega(N \log N)$ amount of communication needed in a single switch for the algorithm to specify a matching. We also suggested some heuristics that reduce the amount of state information that the switching algorithm needs to look at in order to compute a matching, resulting in the algorithm looking only at the state of the first switch. Our approach offers the advantage of using a constant number of parallel layers. This was not the case in [14] and [15], which emulate output queuing for a high line speed using $O(N)$ output queued switches running at lower speed with no memory speedup. While this dependence

on N can be eliminated by replacing the output queued switches with input-output queued switches [7], the algorithm for emulating an output queued switch becomes more complicated and much less practical to implement. Moreover, our approach makes use of FIFO queues only, whereas the approach outlined in [7] requires the use of non-FIFO queues. The bandwidth requirement of the architecture proposed here is kNR where R is the line speed. The authors of [15] succeeded in reducing this bandwidth requirement to NR only at the expense of allowing packets to arrive in an out-of-order fashion with a bounded delay of $O(N^2)$.

Chapter 6

Scheduling Unsplittable Flows Using Parallel Switches

In this chapter, we address the problem of scheduling unsplittable flows using a number of switches in parallel. This has applications in optical switching and eliminates the need for re-sequencing in traditional packet switching. As we have seen in the previous chapter, the use of parallel switches provides a way of building a high speed switch while overcoming the speedup requirement imposed on the switch. Unlike packet switching however, we will assume that flows cannot be split across switches. This constraint adds a new dimension to the problem: various questions, such as obtaining the best schedule, i.e. the schedule with the maximum throughput possible, become NP-hard.

Our problem here is a special case of the general unsplittable flow problem (see [10] for references), where in a directed capacitated graph containing a number of commodities with demands, the goal is to obtain a flow that does not violate capacity and in which all demands are satisfied and every commodity flows along a single path. In this chapter, we are not going to address the general problem. Rather, we will study the special case of scheduling unsplittable flows using parallel switches, and present some simple approximation algorithms to various aspects of the problem. An approximation algorithm is a polynomial time algorithm that produces a solution within a constant factor of the optimal solution (which is NP-hard and hence no

efficient algorithm for obtaining it is known). For instance, an ϵ-approximation algorithm for maximizing the throughput is a polynomial time algorithm that produces a throughput that is at least an ϵ fraction of the maximum throughput possible. We also define a speedup version of the problem and discuss under what speedup we can schedule all the flow.

Before we proceed to the description of the architecture and the problem in more detail, we first motivate the approach and list a number of assumptions. Three main concerns motivate our decision for not to split the flows:

- Per-flow guarantees: We would be able to achieve per-flow guarantees since each flow will have a dedicated path and bandwidth.
- Re-sequencing: Since packets cannot be out-of-order at the output port anymore, we will eliminate the need for re-sequencing, which is a major drawback at the output.
- Optical flows: We would accommodate for optical switches where the optical flows are naturally unsplittable.

In the ideal situation, we would like our scheduling algorithm to be an *online* algorithm (as opposed to *offline*). An online algorithm schedules flows as soon as they arrive without any knowledge about future flows. For our approximation algorithms, however, we consider offline algorithms. We prove that with no speedup (which will be defined later for our case of unsplittable flows), a fairly general notion of an online algorithm, that we call *greedy*, cannot schedule a subset of the flows in a way to obtain a throughput that is a positive fraction of the maximum throughput possible, even if the flows are admissible (i.e. can be scheduled if splitting is allowed). In other words, there is no *greedy* approximation algorithm for the problem of maximizing throughput.

We also would like our algorithm to be *oblivious* in the sense that it would be able to schedule the flows without any knowledge of the remaining capacities on the links connecting the switches to the output ports, since this knowledge is probably hard to

obtain without direct communication between the input and output ports or between the input ports themselves. Throughout this chapter, however, we will assume that the algorithm is not oblivious and has exact knowledge of the remaining capacities on the links connecting the switches to the output ports.

6.1 The Problem

The switching architecture that we are going to use is depicted in Figure 6-1. This is similar to the architecture in Figure 5-1, except that we do not restrict the switches to be input-output queued switches and we do not explicitly requires the use of demultiplexers and multiplexers at the input and output ports respectively.

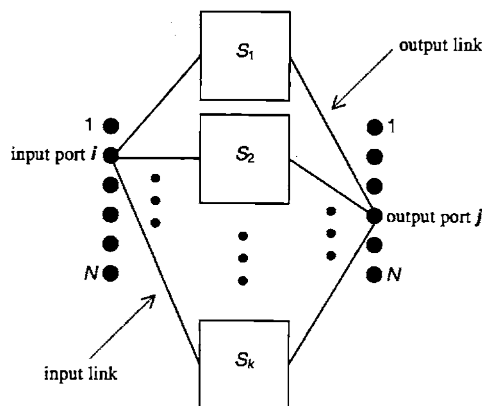


Figure 6-1: The unsplittable flow parallel architecture

Each input and output is connected by links to all k switches. Each link has capacity 1 and, as a consequence, each switch will be able to handle N units of bandwidth, where N is the number of input and output ports. Each flow is at most 1 unit of bandwidth. In the speedup version of the problem, each link has capacity S (where $S > 1$), and therefore, each switch handles SN units of bandwidth. We choose not to affect the demand (individual flow size) with speedup and therefore, each flow will still be at most 1 unit of bandwidth in the speedup version of the problem.

Therefore, given a set of flows each of which is at most 1, we would like to unsplittably schedule the flows, which means to assign flows to the switches, such that all link capacities are not exceeded i.e. each link handles at most 1 unit of bandwidth (or

S units of bandwidth in the speedup version). A set of flows is admissible iff the sum of flows at every input and every output is at most k. This means that all the flows can be scheduled if flow splitting is allowed. We are going to address the following questions which are generally addressed in the literature:

- Maximization: Given a set of flows, can we unsplittably schedule a subset whose throughput is the maximum throughput possible?
- Number of rounds: Given a set of flows, what is the minimum number of rounds that are needed to unsplittably schedule all the flows?
- Speedup: Given an admissible set of flows, what is the minimum speedup needed to unsplittably schedule all the flows in one round? This is known as the congestion factor in the literature [10].

Note that all the problems stated above are NP-hard. With regard to the maximization problem, we will present some simple approximation algorithms (offline) that guarantee a constant fraction of the maximum throughput possible. As for the number of rounds, Du et al. show in [11] that $\lceil \frac{17k-5}{6} \rceil$ switches (instead of k) with no speedup are enough to unsplittably schedule an admissible set of flows. Therefore, 3 rounds are enough to unsplittably schedule an admissible set of flows. We will provide a 4-approximation algorithm for the number of rounds when the set of flows is not admissible. Du et al. show also in [11] that it is possible to unsplittably schedule an admissible set of flows with a speedup $S \geq 1 + \frac{k-1}{k}B$, where k is the number of switches and B is an upper bound on the size of any flow. Therefore, in our case, a speedup of 2 will be enough. We will address the same speedup question for the case of online algorithms.

6.2 Theoretical Framework

In this section, we define a term that we call *blocking factor*. Before we go to the main definition, we need some preliminary definitions. Assume that we have a set of

flows F and that we schedule a subset of flows $G \subseteq F$. In other words, G is the set of flows in F that are passing through the parallel switch architecture. Let $in_{s,f}$ be the input link connecting the input port of flow f to switch s. Similarly, let $out_{s,f}$ be the output link connecting the output port of flow f to switch s. Let $u(e)$ be the amount of flow passing through link e. An input link e in switch s is *blocking* if there exists a flow $f \notin G$ such that $e = in_{s,f}$ and $u(e) \geq u(out_{s,f})$. Similarly, an output link e in switch s is *blocking* if there exists a flow $f \notin G$ such that $e = out_{s,f}$ and $u(e) \geq u(in_{s,f})$.

Definition 6.1 (blocking factor) *The blocking factor β is defined as follows:*

$$\beta = \begin{cases} \min_{e \in B} u(e) & \text{if } B \neq \emptyset \\ \infty & \text{if } B = \emptyset \end{cases}$$

where B is the set of all blocking links.

Note that for any switch s and every flow $f \notin G$, at least one of $in_{s,f}$ and $out_{s,f}$ is a blocking link, and hence f has to use at least one blocking link in s in order to be routed through s. Therefore, every flow $f \notin G$ has to use at least one blocking link to be routed through the parallel switch architecture.

The blocking factor is a measure of how large the throughput of G is, since for every flow $f \notin G$, we look at how much flow in G is going through the switches, either from the input port of f or to the output port of f. A large blocking factor is therefore an indication of a high utilization of the parallel switch architecture and hence of a high approximability of the maximization problem.

6.3 Maximization

In this section, we establish a loose connection between the blocking factor and the approximability of the maximization problem in scheduling unsplittable flows. More precisely, we show that a blocking factor β implies a $\frac{\beta}{2+\beta}$-approximation. We first

prove that a fairly general notion of an online algorithm, that we call *greedy*, cannot achieve any positive approximation factor.

Definition 6.2 (*greedy*) *A greedy scheduling algorithm is an online algorithm that satisfies the following conditions:*

- *When a flow arrives, the algorithm has to decide immediately whether to accept or reject the flow without waiting for any future flows.*
- *If a flow can be scheduled without violating any link capacity, then the algorithm has to accept the flow and assign it to one of the parallel switches.*
- *once a flow is accepted and assigned to a switch, it cannot be re-routed.*

It is worth mentioning that Tsai et al. proved in [30] that a multirate clos network is widesense nonblocking only if the number of switches is at least $3k - 2$. Since a multirate clos network is a special case of our architecture, the result still applies. This is equivalent to the statement that any *greedy* algorithm requires at least $3k - 2$ switches in order to unsplittably schedule all flows.

Theorem 6.1 *For any $0 < \epsilon \leq 1$, there is no greedy ϵ-approximation algorithm for the problem of maximizing the throughput in scheduling unsplittable flows, even with an admissible set of flows.*

Proof: The proof is by construction of a particular instance of the problem where, for a given ϵ, $N = (\lfloor \frac{1}{\epsilon} \rfloor + 1)k$. We divide the input ports into two sets I_1 and I_2, where I_1 contains the first k input ports. Similarly, we divide the output ports into two sets O_1 and O_2, where O_1 contains the first k output ports. Assume that for every input port $i \in I_1$, we schedule an amount of flow equal to 1 from input port i to output port i in all k switches, except for switch i where we schedule a flow of size $1 - \epsilon'$ from input port i to output port i, where $0 < \epsilon' < \frac{1}{2}$.

Given the above setting, any future flow for output $j \in O_1$ has to be assigned to switch j, since all links to output j in all other switches are fully utilized. We will later describe how we can force such a setting.

Next, from every input port $i \in I_2$, we receive k flows of size ϵ'' for all the output ports in O_1. Note that for the admissibility condition to hold at an output port $j \in O_1$, we require that $\lfloor \frac{1}{\epsilon} \rfloor k\epsilon'' \leq \epsilon'$; we can assume equality. The *greedy* algorithm will have to schedule these flows and, as argued above, will assign flows for output j to switch j. For all $i \in I_2$, this will make all k links from input port i partially utilized.

Next, for each $i \in I_2$, we receive $k-1$ flows of size 1 and one flow of size $1 - k\epsilon''$, all from input port i to output port i. Note that the admissibility condition at input ports $i \in I_2$ still holds since $\epsilon'' \times (k) + 1 \times (k-1) + 1 - k\epsilon'' = k$. For each $i \in I_2$, the *greedy* algorithm can only schedule the last flow of size $1 - k\epsilon''$ from input port i to output port i, since every link from input port i is now partially utilized.

So far, the sum of all flows is $kN - k\epsilon'$. So we can still have $k\epsilon'$ additional amount of flow (ϵ' from each input port $i \in I_1$), and without loss of generality, we can assume that the *greedy* algorithm is able to schedule them.

Note that the maximum throughput possible is at least $k^2 - k\epsilon' + \lfloor \frac{1}{\epsilon} \rfloor k(k-1)$ which consists of the initial setting of the switches in addition to assigning to each of the first $k-1$ switches a flow of size 1 from input port i to output port i, for all $i \in I_2$. Therefore, the maximum throughput possible is $O((1 + \lfloor \frac{1}{\epsilon} \rfloor)k^2)$. The throughput achieved by the *greedy* algorithm is $k^2 + (1 - k\epsilon'')\lfloor \frac{1}{\epsilon} \rfloor k$, which is $O(k^2)$. Therefore, the approximation factor is at most

$$\frac{O(k^2)}{O((1 + \lfloor \frac{1}{\epsilon} \rfloor)k^2)}$$

which goes to $\frac{1}{\lfloor \frac{1}{\epsilon} \rfloor + 1} < \epsilon$ when k grows large enough.

Now we illustrate how we can force the setting described at the beginning of this proof, namely, for a given i, scheduling an amount of flow equal to 1 from input port i to output port i in all k switches, except for switch i where we schedule a flow of size $1 - \epsilon'$. We begin by receiving flows, all of which are greater than $\frac{1}{2}$, from input port i to output port i while increasing the size of the flow in every time by a small amount. In every time, the *greedy* algorithm will choose a different switch to schedule

the flow, since two flows cannot be assigned to the same switch without exceeding the link capacities. After the *greedy* algorithm chooses switch i, which has to happen at some point, we start fully utilizing links from input port i to all the switches that have not been chosen yet by the *greedy* algorithm, by receiving a sufficient number of flows of size 1 from input port i to output port i. After that, among all the switches which can take more flows from input port i to output port i, switch i will have the largest amount of flow going through it from input port i to output port i. We start receiving flows in such a way to cause the *greedy* algorithm to choose switches in the same order it had before, by receiving the flows in decreasing size this time and fully utilizing every link from input port i, except the one going to switch i. For instance, if the amount of flow going from input port i to output port i through switch s is b, we receive a flow of size $1-b$ from input port i to output port i, which has to be assigned to switch s (since we start with the smaller values of b). We stop just after fully utilizing all the links from input port i except the one going to switch i. Hence switch i will have $1-\epsilon'$ amount of flow going from input port i to output port i, where $0 < \epsilon' < \frac{1}{2}$. ■

Note that it is possible to prove Theorem 6.1 using a simpler instance with two switches. In that case however, the total amount of flow presented to the switches will be less than the total capacity of the two switches. The instance used in the proof above has the special property that the total amount of flow received is $C = kN$, where C is the full capacity of all the switches. This implies that the low approximation factor is not due to the absence of flows, and in other words, adding more flows cannot enhance the approximation factor. So even with enough flow equal to C, for any ϵ, a *greedy* algorithm will not be able to achieve a total throughput greater than or equal to ϵC.

We now prove that a blocking factor β implies an approximation factor $\frac{\beta}{2+\beta}$.

Lemma 6.1 *If the blocking factor is β, then the throughput is at least a fraction $\frac{\beta}{2+\beta}$ of the maximum throughput possible.*

Proof: If $\beta = \infty$, then there are no blocking links and hence no flows are left out. Therefore, a fraction equal to $\frac{\infty}{2+\infty} = 1$ of the maximum throughput possible is scheduled. If β is finite, then let L be the number of blocking links. Let F be the set of all flows, and $G \subset F$ be the set of flows that are scheduled. Let G' be the set of flows in $F - G$ that are scheduled in the optimal solution. We know that every flow in $F - G$, and hence in G', has to use at least one blocking link in order to be scheduled in one of the switches. This means that the sum of flows in G' cannot be more than L since every blocking link has capacity 1. Moreover, for every blocking link e, $u(e) \geq \beta$ by definition. This means that the sum of flows in G that are passing through blocking links is at least $\frac{\beta L}{2}$, since a flow can pass through at most two blocking links. This implies that the sum of flows in G is at least $\frac{\beta}{2}$ that of G'. The throughput of the optimal solution cannot be more than the sum of flows in $G \cup G'$ by definition of G and G' . Therefore, the sum of flows in G is at least $\frac{\beta L/2}{L+\beta L/2} = \frac{\beta}{2+\beta}$ of the maximum throughput possible. ∎

Note that the fraction stated in Lemma 6.1 is also true if we consider the maximum amount of flow going through an ideal switch, where splitting of flows is permissible. In the proof above, we did not rely on the fact that the optimal solution does not allow for flow splitting. Therefore, the same analysis applies if we compare our throughput to any other throughput even when splitting occurs.

Using the result above, we can obtain a $\frac{1}{5}$-approximation algorithm for the problem of maximizing the throughput. The algorithm is described below:

Algorithm A:
We divide the flows into two groups: large flows and small flows. Flows that are greater than $\frac{1}{2}$ are considered large, all other flows are considered small. The algorithm starts by scheduling large flows first, in an arbitrary way, until no more large flows can be assigned to the switches. Then it schedules small flows, in an arbitrary way, until no more small flows can be assigned to the switches.

Lemma 6.2 *Algorithm A guarantees a blocking factor $\beta > \frac{1}{2}$.*

Proof: To prove that the blocking factor $\beta > \frac{1}{2}$, assume the opposite. This implies that there is a blocking link e in some switch s such that $u(e) \leq \frac{1}{2}$. By the definition of a blocking link, there exists a flow f that is left out, such that either $e = in_{s,f}$ or $e = out_{s,f}$. As a consequence, $u(in_{s,f}) \leq \frac{1}{2}$ and $u(out_{s,f}) \leq \frac{1}{2}$. This means that flow f cannot be a small flow, since otherwise, it could have been assigned to switch s before the algorithm had stopped. So f must be a large flow. But since $u(in_{s,f}) \leq \frac{1}{2}$ and $u(out_{s,f}) \leq \frac{1}{2}$, only small flows are passing through $in_{s,f}$ and $out_{s,f}$, which contradicts the way the algorithm favors large flows first. Therefore, $\beta > \frac{1}{2}$. ■

Theorem 6.2 *There exists a $\frac{1}{5}$-approximation algorithm for the problem of maximizing the throughput in scheduling unsplittable flows.*

Proof: *Algorithm A* is a polynomial time algorithm. By Lemma 6.2, *Algorithm A* guarantees a blocking factor $\beta > \frac{1}{2}$, which by Lemma 6.1, implies a $\frac{1}{5}$-approximation for the problem of maximizing throughput. ■

In the following section, we will see that a $\frac{1}{3}$-approximation algorithm exists for the problem of maximizing throughput when the set of flows is admissible.

6.4 Number of Rounds

The fact that it might be unfeasible to schedule all flows unsplittably, even when the admissibility condition holds, motivates the idea of rounds. In this section, we ask how many rounds are needed to unsplittably schedule all the flows. The authors in [10] provide an algorithm that schedules all flows unsplittably in a general graph with a single source in 5 rounds, given that the cut condition holds. The cut condition is a generalized admissibility condition for graphs. They also show that this leads to a 5-approximation algorithm for the problem of minimizing the number of rounds when the cut condition is not satisfied.

In our case, we provide a 4-approximation algorithm to the minimum number of rounds needed to schedule all flows unsplittably. When the set of flows is admissible, Du et al. proved in [11] that $\lceil \frac{17k-5}{6} \rceil$ switches are sufficient to unsplittably schedule

all flows. This implies that 3 rounds are also sufficient since 3 rounds are equivalent to $3k$ switches. For the sake of completeness, we provide a simple polynomial time algorithm that unsplittably schedules an admissible set of flows using $3k$ switches. Note that when the set of flows is admissible, a polynomial time algorithm that schedules all flows in r rounds implies a $\frac{1}{r}$-approximation algorithm for the problem of maximizing throughput, simply by choosing the round with the maximum throughput, which contains at least $\frac{1}{r}$ of the sum of all flows. As a consequence, we have a $\frac{1}{3}$-approximation algorithm for the problem of maximizing throughput when the set of flows is admissible.

We first describe an offline 4-approximation algorithm to the minimum number of rounds needed to schedule any set of flows:

Algorithm B:

This algorithm consists of a number of rounds. In each round we run *Algorithm A* on the remaining flows. We stop when all flows have been scheduled.

First we prove a simple lemma.

Lemma 6.3 *If a and b are two integers greater than 0 then* $\lfloor \frac{a-1}{b} \rfloor + 1 \geq \frac{a}{b}$.

Proof: $a - 1$ can be written as $q \times b + r$ where both q and r are non-negative integers and $r < b$. Then $\lfloor \frac{a-1}{b} \rfloor = q$. Finally, $\frac{a}{b} = \frac{q \times b + r + 1}{b} = q + \frac{r+1}{b} \leq q + 1$. ■

Theorem 6.3 *There exists a 4-approximation algorithm for the problem of minimizing the number of rounds in scheduling unsplittable flows.*

Proof: Assume that *Algorithm B* stops after r rounds. Let f be a flow that is scheduled in the rth round. Then we know that in the first $r - 1$ rounds, flow f cannot be scheduled, and as a consequence, it has a blocking link in every switch during all $r - 1$ rounds. Since *Algorithm B* runs *Algorithm A* in every round, the blocking factor β must be greater than $\frac{1}{2}$ by Lemma 6.2. This implies that the total amount of flow coming from the input port of f or going to the output port of f is

more than $\frac{1}{2} \times k \times (r-1) = (\frac{r-1}{4}) \times 2k$ during the first $r-1$ rounds. In one round however, we cannot schedule more than $2k$ amount of flow for any pair of input and output ports. This means that we cannot optimally have less than $\lfloor \frac{r-1}{4} \rfloor + 1$ rounds to schedule all the flows including flow f. By Lemma 6.3, this is at least $\frac{r}{4}$. Therefore, we have the result since *Algorithm B* is a polynomial time algorithm. ■

Next, we present an algorithm that unsplittably schedules an admissible set of flows in 3 rounds. As in the previous algorithms, this algorithm relies on the idea of dividing the flows into two groups.

Algorithm C:
We divide the flows into two groups: large flows and small flows. Flows that are greater than $\frac{1}{3}$ are considered large, all other flows are considered small. The algorithm starts by scheduling large flows first using the rearrangeability property of a clos network (Slepian-Duguid theorem [26]): Since at most $3k-1$ large flows can exist at any port (and each is at most 1), we can unsplittably schedule the large flows using at most $3k-1$ switches, or alternatively $3k$ switches. Then the small flows are scheduled in an arbitrary way. The $3k$ switches correspond to the 3 rounds.

Lemma 6.4 *Let F be a set of flows. If no flow $f \in F$ can be scheduled and each flow $f \in F$ is at most B, then the blocking factor β satisfies $\beta > S - B$, where S is the speedup.*

Proof: Assume the opposite. By definition of the blocking factor, there exists a flow $f \in F$ and a switch s such that $u(in_{s,f}) \leq S - B$ and $u(out_{s,f}) \leq S - B$. Therefore, we can assign f to switch s without violating any link capacity (recall that any flow in F, in particular flow f, is at most B). This is clearly a contradiction since flow f cannot be scheduled. Therefore, the blocking factor β satisfies $\beta > S - B$. ■

Theorem 6.4 *Algorithm C unsplittably schedules any admissible set of flows in at most 3 rounds.*

Proof: It is enough to show that with *Algorithm C*, no small flows can be left out. To prove this fact, let F be the set of small flows that cannot be scheduled with *Algorithm C*. Applying Lemma 6.4 to F, $B = \frac{1}{3}$, and $S = 1$, we obtain that the blocking factor β satisfies $\beta > \frac{2}{3}$ in all 3 rounds. Consider a flow $f \in F$. Since f has a blocking link in every switch in all 3 rounds, and the blocking factor is more than $\frac{2}{3}$, the amount of flow coming from the input port of f or going to the output port of f is more than $\frac{2}{3} \times k \times 3 = 2k$. From the admissibility condition however, we know that at most $2k$ amount of flow can exist for any two ports. This is a contradiction. Therefore, the set F has to be empty. ■

Corollary 6.1 *There exists a $\frac{1}{3}$-approximation algorithm for the problem of maximizing the throughput in scheduling unsplittable flows when the set of flows is admissible.*

Proof: By Theorem 6.4, *Algorithm C* schedules an admissible set of flows in 3 rounds. The Corollary is true since *Algorithm C* is a polynomial time algorithm and the round with the maximum throughput among all rounds has to contain at least $\frac{1}{3}$ of the total amount of flows in the admissible set. ■

In comparison with the work in [10], a $\frac{1}{4.43}$-approximation algorithm is obtained for the problem of maximizing throughput in a general graph with a single source, when the cut condition is satisfied.

Theorem 4 also implies that a speedup of 3 is sufficient to unsplittably schedule an admissible set of flows. This can be achieved by superposing all 3 rounds together to get the effect of one round where each link has capacity 3. In the following section, we prove a stronger result, namely that any *greedy* algorithm can unsplittably schedule an admissible set of flows with a speedup of 3.

6.5 Speedup

As mentioned before, Du et al. proved in [11] that a speedup of 2 is enough to unsplittably schedule an admissible set of flows. Note that 2 is also a lower bound on the speedup required to unsplittably schedule an admissible set of flows. To see this,

consider $k+1$ flows from input port i to output port j, each of size $\frac{k}{k+1}$. Since there are k switches only, at least 2 flows must be assigned to the same switch. Therefore, the minimum speedup required is $\frac{2k}{k+1}$. For any $\epsilon > 0$, by choosing a large enough k, we can make $\frac{2k}{k+1} = 2 - \epsilon$. Therefore, 2 is a tight lower bound on the speedup required to unsplittably schedule an admissible set of flows. In this section, we concentrate on *greedy* algorithms. We prove that any *greedy* algorithm can schedule an admissible set of flows when the speedup is at least 3.

Theorem 6.5 *Any greedy algorithm can unsplittably schedule an admissible set of flows with a speedup* $S \geq 3$.

Proof: Let F be the set of flows that cannot be scheduled using the *greedy* algorithm. Applying Lemma 6.4 to F, $B = 1$, and $S = 3$, we obtain that the blocking factor β satisfies $\beta > 2$. Consider a flow $f \in F$. Since f has a blocking link in every switch, and the blocking factor is more than 2, the amount of flow coming from the input port of f or going to the output port of f is more than $2k$. From the admissibility condition however, we know that at most $2k$ amount of flow can exist for any two ports. This is a contradiction. Therefore, the set F has to be empty. ■

The implication of Theorem 6.5 is that, with flows appearing and disappearing, a simple online algorithm can continue to schedule all flows, provided that at any time, the set of existing flows is admissible. Note that the online algorithm has to be non-oblivious (see Section 6.1 for the definition of oblivious). We can prove that $S = 3$ is actually a lower bound for two natural classes of *greedy* algorithms. We call these two classes *packing* and *load balancing*. We begin by defining a *packing* algorithm:

Definition 6.3 (***packing***) *A packing algorithm is a greedy algorithm by which, whenever possible, a new flow* f *is assigned to a switch* s *such that either* $u(in_{s,f}) \neq 0$ *or* $u(out_{s,f}) \neq 0$.

For instance, a *greedy* algorithm that, whenever possible, does not utilize a link that is so far unutilized, is a *packing* algorithm. Similarly, the *greedy* algorithm that

assigns a flow f to a switch s that maximizes $\max(u(in_{s,f}), u(out_{s,f}))$ is a *packing* algorithm.

Next, we define a *load balancing* algorithm:

Definition 6.4 (***load balancing***) *A load balancing algorithm is a greedy algorithm by which, whenever possible, a new flow f is assigned to a switch s such that $u(in_{s,f}) = 0$ and $u(out_{s,f}) = 0$. If this is not possible, then f is scheduled in such a way to keep the maximum used link capacity at a minimum.*

For instance, the *greedy* algorithm that assigns a flow f to a switch s that minimizes $\max(u(in_{s,f}), u(out_{s,f}))$ is a *load balancing* algorithm.

We have the following results:

Theorem 6.6 *There is no packing algorithm that can schedule every admissible set of flows with a speedup $S < 3$.*

Proof: The proof is by choosing a speedup $S = 3 - \epsilon$ for any $\epsilon > 0$ and constructing an admissible set of flows that will cause the *packing* algorithm to fail in scheduling all the flows. We will assume that k is even and that $\frac{3k}{k+1} > S = 3 - \epsilon$, which can be obtained with a large enough k. We also require a large enough number N of input and output ports such that $N \geq k \times C^{k}_{\frac{k}{2}} + 2$. Let i_1, i_2, ..., i_N denote the input ports. Similarly let j_1, j_2, ..., j_N denote the output ports. For each $l = 1...N-1$, we receive k flows of size $\frac{k}{k+1}$ from input port i_l to output port j_l. Since $\frac{3k}{k+1} > 3 - \epsilon$, at most 2 flows from i_l to j_l can be assigned to a single switch. Moreover, since the algorithm is a *packing* algorithm, once a flow from i_l to j_l is assigned to a switch s, the next scheduled flow from i_l to j_l will be assigned to switch s as well. Therefore, exactly $\frac{k}{2}$ switches will be used to schedule the k flows from i_l to j_l for $l = 1...N-1$. $C^{k}_{\frac{k}{2}}$ represents all possible ways of choosing $\frac{k}{2}$ switches among k switches. Since we have $N - 1 \geq k \times C^{k}_{\frac{k}{2}} + 1$, at least $k + 1$ (i_l, j_l) pairs will utilize the same $\frac{k}{2}$ switches s_1, s_2, ..., $s_{\frac{k}{2}}$. Let p_1, p_2, ...,p_{k+1} be the input ports of these $k + 1$ pairs. Now for $l = 1...k+1$, we receive a flow of size $\frac{k}{k+1}$ from p_l to j_N. Note that the admissibility condition still holds because $(k+1) \times \frac{k}{k+1} = k$. Since $\frac{3k}{k+1} > 3 - \epsilon$, switches s_1, s_2, ...,

$s_{\frac{k}{2}}$ cannot be used to schedule any of the new $k+1$ flows. Hence, exactly $\frac{k}{2}$ switches are available to schedule the $k+1$ flows. Each of the available $\frac{k}{2}$ switches can hold at most 2 of the $k+1$ flows since $\frac{3k}{k+1} > 3 - \epsilon$. Therefore, one of the $k+1$ flows cannot be scheduled. ■

Theorem 6.7 *There is no load balancing algorithm that can schedule every admissible set of flows with a speedup $S < 3$.*

Proof: The proof is similar to the one for *packing* algorithms. We assume that the number of ports is this time $N = 2N_0 + 1$, where $N_0 \geq k \times C^k_{\frac{k}{2}} + 1$. So for each $l = 1 \ldots N_0$, we can define the input ports i_l and i_{l+N_0}, and the output ports j_l and j_{l+N_0}. The idea is to make the *load balancing* algorithm utilize $\frac{k}{2}$ switches in the same way presented in the previous proof. We will illustrate how this can be done for one input-output pair (i_{l_0}, j_{l_0}). We first receive $\frac{k}{2}$ flows of size $\frac{k}{k+1}$ from i_{l_0} to j_{l_0}. By the definition of a *load balancing* algorithm, these flows will be assigned to $\frac{k}{2}$ different switches, say s_1, s_2 ,..., $s_{\frac{k}{2}}$. Next we receive $\frac{k}{2}$ flows of size $\epsilon < \frac{1}{k+1}$ from input port $i_{l_0+N_0}$ to output port j_{l_0}. The *load balancing* algorithm will schedule these flows using the other $\frac{k}{2}$ switches, say $s_{\frac{k}{2}+1}$, $s_{\frac{k}{2}+2}$, ...,s_k. Next we receive $\frac{k}{2}$ flows of size $\frac{1}{k+1}$ from input port $i_{l_0+N_0}$ to output port $j_{l_0+N_0}$. The *load balancing* algorithm will schedule the new flows using the switches s_1, s_2 ,..., $s_{\frac{k}{2}}$. Next, we receive $\frac{k}{2}$ flows of size 1 from input port $i_{l_0+N_0}$ to output port $j_{l_0+N_0}$. If the *load balancing* algorithm assigns any of these new flows to any of the switches s_1, s_2 ,..., $s_{\frac{k}{2}}$, then the maximum used link capacity will be $1 + \frac{1}{k+1}$. Therefore, the *load balancing* algorithm will schedule the new flows using switches $s_{\frac{k}{2}+1}$, $s_{\frac{k}{2}+2}$, ...,s_k, making the maximum used link capacity $1 + \epsilon < 1 + \frac{1}{k+1}$. Finally, we receive $\frac{k}{2}$ flows of size $\frac{k}{k+1}$ from input port i_{l_0} to output port $j_{l_0+N_0}$. The *load balancing* algorithm will schedule these flows using the switches s_1, s_2 ,..., $s_{\frac{k}{2}}$, making the maximum used link capacity $2\frac{k}{k+1}$ as opposed to $\frac{k}{k+1} + 1$ in case it assigns any of these flows to any of the switches $s_{\frac{k}{2}+1}$, $s_{\frac{k}{2}+2}$, ...,s_k. We can repeat this process for all $l = 1 \ldots N_0$ yielding to a situation similar to the one described in the previous proof, where at least $k+1$ pairs (i_l, j_l), with a $2\frac{k}{k+1}$ amount of flow from i_l to j_l, utilize the same $\frac{k}{2}$ switches. Note that the admissibility condition

holds everywhere. ■

6.6 Summary

We addressed a setting of parallel switches in which flows cannot be split across multiple switches. This offers the advantage of eliminating re-sequencing at the output in traditional packet switching and provides a framework for optical switches where flows are naturally unsplittable. Most of the questions regarding scheduling unsplittable flows are NP-hard. We looked at some approximation algorithms for different aspects of the problem of scheduling unsplittable flows. We proved that a general notion of an online algorithm cannot achieve any fraction of the maximum throughput possible, and presented some simple offline approximation algorithms for maximizing throughput. We also looked at how to approximate the number of rounds needed to schedule all the flows using an offline algorithm. Finally, we showed that any online algorithm can schedule an admissible set of flows with a speedup of 3, and that $S = 3$ is actually a lower bound on some natural classes of online algorithms.

Chapter 7

Conclusion

A major drawback of the traditional output queuing technique is that it requires a switch speedup of N, where N is the size of the switch. This dependence on N makes the switch non-scalable at high speeds. Input queuing has been suggested instead. The introduction of input queuing creates the necessity for developing switching algorithms to decide which packets to keep waiting at the input, and which packets to forward across the switch. Input-output queuing is a more general model where queues are used at both input and output ports. Moreover, input-output queuing allows the possibility to have a speedup S that is not necessarily dependent on N. Switching in an input-output queued switch has been abstracted in literature as a computation of matchings in which input and output ports are matched together (see chapter 1). Some switching algorithms still require the switch to operate at a speed higher than the line speed to achieve basic guarantees. In other words, they require a switch speedup $S > 1$ such that the switch computes successive matchings every $\frac{1}{S}$ time units. In this thesis, we abstracted many of these algorithms as families of algorithms, and established some lower bounds on the speedup required by these families of algorithms to guarantee throughput.

A practical family of algorithms, priority switching algorithms, have been proposed in literature [6], [16], [18] to overcome the high computational complexity of traditional switching algorithms [21], [23], [24] (these are usually based on computing a maximum weighted matching). A priority switching algorithm computes a matching

by assigning priorities to the different input-output pairs and adding the pairs to the matching in order of their priorities. We proposed two priority switching algorithms that provide throughput with a speedup $S = 2$ and a delay guarantee with a speedup $S > 2$ under appropriate traffic models. They offer the advantage of requiring smaller amount of state information than other priority switching algorithms.

An even more practical family of algorithms, iterative switching algorithms, have been also suggested in literature. These algorithms, due to their distributed nature, do not require global computation of matchings; however, they require multiple iterations to provide high throughput. We proposed an iterative switching algorithm that provides high throughput in practice with one iteration only. The algorithm will also provide, with only one iteration, throughput with a speedup $S = 2$ as well as a delay guarantee with a speedup $S > 2$ under an appropriate traffic model. The property of requiring one iteration only makes it possible to scale the switch at higher speeds.

We investigated the use of multiple input-output queued switches with no speedup in parallel in order to achieve a delay guarantee while eliminating the speedup requirement imposed on the switch. We pushed further the idea of using parallel switches (not necessarily input-output queued) to exploit a setting in which flows cannot be split across multiple switches.

7.1 Some Lower Bounds on Speedup

We proved lower bounds on the speedup required by several classes of switching algorithm to achieve a weak notion of throughput (definition appears in Chapter 1).

- A class of priority switching algorithms that uses a priority scheme based on the output queues of the switch requires a speedup of 2 (a tight bound).
- An algorithm that computes a maximum size matching requires a speedup of 2 (a tight bound).
- A class of priority switching algorithms that uses a priority scheme based on the sate of the input queues of the switch requires a speedup of 1.5 (not known

to be a tight bound).

The above results motivated us to consider the use of parallel switches to accommodate for speedup.

7.2 Two Priority Switching Algorithms

We presented two priority switching algorithms that provide strong throughput (definition appears in Chapter 1) with a speedup $S = 2$ and a delay guarantee with a speedup $S > 2$, under appropriate constant burst traffic models. Both algorithms offer the advantage of not requiring extensive state information like the age of packets (as in the [6] and [18]), the length of the input queues (as in [16]), or the length of the output queues (as in [18]). Moreover, they do not require the traffic to be constantly backlogged as it is the case for the algorithm in [16]. The running time of both algorithms is $O(N^2)$ in the RAM model of computation and their memory requirement is $O(N^2 \log N)$. The communication complexity of both algorithms is $O(N \log N)$ which is optimal if we consider the $\Omega(N \log N)$ amount of communication required to specify a matching for the switch in order to configure the input and output ports appropriately. Therefore, both algorithms offer a better communication requirement compared to the previous algorithms in which more information needs to be communicated, like the age of packets for instance.

7.3 An Iterative Switching Algorithm

We developed an iterative switching algorithms that, with a particular priority scheme, provides strong throughput (definition appears in Chapter 1) with a speedup of 2 and a delay guarantee with a speedup $S > 2$ under an appropriate constant burst traffic model. The switching algorithm requires $O(\log N)$ computational complexity with appropriate parallelism. This algorithm offers the advantage of not requiring more than one iteration to provide high throughput, and outperforms other iterative algorithms when the number of iterations is limited to one. Moreover, it provides

theoretical guarantees (with a speedup of 2) under non-uniform traffic pattern, unlike existing iterative switching algorithms which provide theoretical guarantees under uniform traffic patterns only.

7.4 Switching Using Parallel Switches

We suggested a scheme that eliminates the need for speedup by using $k = \lceil S \rceil$ parallel input-output queued switches with no speedup, where S is the speedup of the original switch. The key to our approach was to apply the same matching in all the parallel switches. By adapting existing switching algorithms for the single switch setting to hold their matching constant for a number of times, we were able to apply the same matching is all switches, and guarantee a bounded delay on every packet. The additional communication cost between the switching algorithm and the parallel switches is $O(N \log kN)$. This is to be compared to the $\Omega(N \log N)$ amount of communication needed in a single switch for the algorithm to specify a matching. Our approach offers the advantage of using a constant number of parallel layers independent of N, the size of the switch. This was not the case in [14] and [15], which emulate output queuing for a high line speed using $O(N)$ output queued switches running at a lower speed with no memory speedup. The bandwidth requirement of the architecture proposed here is kNR where R is the line speed. The authors of [15] succeeded in reducing this bandwidth requirement to NR only at the expense of allowing packets to arrive in an out-of-order fashion with a bounded delay of $O(N^2)$.

7.5 Unsplittable Flows

We addressed a setting of parallel switches in which flows cannot be split across multiple switches. This offers the advantage of eliminating re-sequencing at the output in traditional packet switching and provides a framework for optical switches where flows are naturally unsplittable. Most of the questions regarding scheduling unsplittable flows are NP-hard. We looked at some approximation algorithms for different

aspects of the problem of scheduling unsplittable flows. We proved that a general notion of an online algorithm cannot achieve any fraction of the maximum throughput possible, and presented some simple offline approximation algorithms for maximizing throughput. We also looked at how to approximate the number of rounds (the number of times the switches are used) needed to schedule all the flows using an offline algorithm. Finally, we showed that any online algorithm can schedule an admissible (defined in Chapter 6) set of flows with a speedup of 3, and that $S = 3$ is actually a lower bound on some natural classes of online algorithms.

Bibliography

[1] T.E. Anderson, S. Owicki, J. Saxes, C. Thacker, *High Speed Switch Scheduling for Local Area Networks.* ACM Transactions on Computer Systems 11(4), pp. 319-52, November 1993.

[2] J. Blanton, H. Badt, G. Damm, P. Golla, *Iterative Scheduling Algorithms for Optical Packet Switches.* IEEE International Conference on Communications ICC 2001, Helsinki, June 2001.

[3] J. Blanton, H. Badt, G. Damm, P. Golla, *Impact of Polarized Traffic on Scheduling Algorithms for High Speed Optical Switches.* ITCom 2001, Denver, August 2001.

[4] C.-S. Chang, W.-J. Chen, H.-Y. Huang, *On Service Guarantees for Input-buffered Crossbar Switches: A Capacity Decomposition Approach by Birkhoff and von Neumann.* IEEE IWQOS 99, pp. 79-86.

[5] C.-S. Chang, D.-S. Lee, Y.-S. Jou, *Load Balanced Birkhoff-von Neumann Switches.* IEEE Workshop on High Performance Switching and Routing, HPSR 2001, pp. 276-80.

[6] A. Charny, P. Krishna, N. Patel, R. Simcoe, *Algorithms for Providing Bandwidth and Delay Guarantees in Input-buffered Crossbars With Speedup.* Proceedings of 6th International Workshop on Quality of Service, IWQOS 98, pp. 235-44, May 1998.

[7] S.-T. Chuang, A. Goel, N. McKeown, B. Prabhakar, *Matching Output Queuing with Combined Input-Output Queued Switches.* IEEE Journal of Selected Areas in Communication 17(6), pp. 1030-39, June 1999.

[8] J. G. Dai, B. Prabhakar, *The Throughput of Data Switches With and Without Speedup.* IEEE/ACM INFOCOM 2000, pp. 556-64.

[9] G. Damm, J. Blanton, P. Golla, D. Verchere, M. Yang, *Fast Scheduler Solutions to the Problem of Priorities for Polarized Data Traffic.* Alcatel USA Technical Report.

[10] Y. Dinitz, N. Garg, M. Goemans, *On The Single Source Unsplittable Flow Problem.* Combinatorica 19(1) 1999, pp. 17-41.

[11] D. Z. Du, B. Gao, F. K. Hwang, J. H. Kim, *On Multirate Rearrangeable Clos Networks.* Siam Journal of Computing 28(2), pp. 463-70, 1998.

[12] D. Gale, L. S. Shapley, *College Admissions and the Stability of Marriage.* American Mathematical Monthly, vol. 69, pp. 9-15, 1962.

[13] P. Golla, J. Blanton, G. Damm, *Improved Iterative Ping Pong Scheduling Algorithm for Fast Switches.* Alcatel USA Technical Report.

[14] S. Iyer, A. Awadallah, and N. McKeown, *Analysis of a Packet Switch with Memories Running Slower then the Line Rate.* Proceedings of IEEE INFOCOM 2000, vol. 2, pp. 529-37, March 2000.

[15] S. Iyer, N. McKeown, *Making Parallel Packet Switches Practical.* Proceedings of IEEE INFOCOM 2001, pp. 1680-87, March 2001.

[16] A. Kam, K.-Y. Siu, R. Barry, *A Cell Switching WDM Broadcast LAN with Bandwidth Guarantee and Fair Access.* IEEE/OSA Journal of Lightwave Technology 16(2), pp. 2265-80, December 1998.

[17] M. Karol, M. Hluchyj, S. Morgan, *Input versus Output Queuing on a Space Division Switch.* IEEE Transaction on Communications 35(12), pp. 1347-56, 1987.

[18] P. Krishna, N. S. Patel, and A. Charny, *On the Speedup Requirement for Work-Conserving Crossbar Switches.* IEEE Journal of Selected Areas in Communication 17(6), pp. 1057-69, June 1999.

[19] L. Lamport, *Time, Clocks, and the Ordering of Events in a Distributed System.* CACM 21(7), pp. 558-65, 1978.

[20] Y. Li, S. Panwar, H. J. Chao, *Saturn: A Terabit Packet Switch Using Dual Round-Robin (DRR).* IEEE Communication Magazine, pp. 78-84, December 2000.

[21] N. McKeown, V. Anantharam, J. Walrand, *Achieving 100% Throughput in an Input-Queued Switch.* IEEE INFOCOM 96, vol. 1, pp. 296-302, March 1996.

[22] N. Mckeown, *The iSLIP Scheduling Algorithm for Input Queues Switches.* IEEE/ACM Transactions on Networking, 7(2), pp. 188-201, April 1999.

[23] A. Mekkittikul and N. McKeown, *A Starvation-Free Algorithm for Achieving 100% Throughput in an Input Queued Switch.* Proceeding of the International Conference on Computer Communication and Networking, ICCCN 96, pp. 226-231, October 1996.

[24] A. Mekkittikul, N. McKeown, *A Practical Scheduling Algorithm to Achieve 100% Throughput in Input-Queued Switches.* IEEE INFOCOM 1998, vol 2, pp. 792-99, March-April 1998.

[25] S. Mneimneh, V. Sharma, K.-Y. Siu, Switching Using Parallel Input-Output Queued Switches. To appear in ACM/IEEE Transactions on Networking.

[26] Slepian, *unpublished manuscript.* 1958. Result can be found in literature about Clos networks.

[27] R. E. Tarjan, *Data Structures and Network Algorithms.* Society of Industrial and Applied Mathematics, 1983.

[28] L. Tassiulas, A. Ephremides, *Stability Properties of Constrained Queuing Systems and Scheduling Policies for Maximum Throughput in Multihop Radio Networks.* IEEE Transactions on Automatic Control, 37(12), pp. 1936-49, December 1992.

[29] L. Tassiulas, *Linear Complexity Algorithms for Maximum Throughput in Radio Networks and Input Queued Switches.* IEEE INFOCOM 1998, vol. 2, pp. 533-39.

[30] K.-H. Tsai, D.-W. Wang, F. Hwang, *Lower Bounds For Wide-sense Nonblocking Clos Network.* Theoretical Computer Science 261, pp. 323-28, 2001.