SCHEDULING NON-UNIFORM TRAFFIC
IN
HIGH SPEED PACKET SWITCHES AND ROUTERS

A DISSERTATION

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING

AND THE COMMITTEE ON GRADUATE STUDIES

OF STANFORD UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

Adisak Mekkittikul

November 1998

© Copyright 1999 by Adisak Mekkittikul
All Rights Reserved

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

Prof. Nicholas W. McKeown
(Principal Adviser)

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

Prof. Fouad A. Tobagi

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

Prof. Joseph W. Goodman

Approved for the University Committee on Graduate Studies:

Abstract

Until recently, Internet routers and ATM switches were generally built around a central pool of shared memory buffers and a fast, shared-bus backplane. However, limitations in both memory and bus bandwidth have led to the use of input queues and switched backplanes. Input queues relieve the bottleneck by distributing the memory over each switch input; and a switched backplane allows packet transfers to take place simultaneously.

This thesis focuses on the design of switched backplanes with input queues. In particular, we focus on the design of schedulers for switched backplanes. The scheduler decides the order in which packets, or cells, may traverse the backplane. Studies have shown that existing scheduling algorithms are either too complex to operate at high speed or lack the intelligence to perform well over a wide range of traffic patterns.

In this thesis, we present two new algorithms that are fast, simple and efficient. Using the methods of Lyapunov, we prove that both algorithms can achieve 100% throughput for all traffic patterns with independent arrivals. We also demonstrate heuristics that can be implemented in fast and relatively simple hardware.

Our exploratory design work shows that the heuristics can make a scheduling decision within 10-20 nanoseconds when implemented using 0.25 μm CMOS technology. At this scheduling speed, it is possible to design switches or routers with more than one terabit per second of aggregate bandwidth.

Acknowledgments

This thesis would not have been possible without the help of many people. First and foremost, among these are my parents who I would like to thank for their utmost dedication to my education.

Importantly, I would like to thank my adviser, Professor Nick McKeown, for his unparalleled support and guidance in both academic and personal matters. I also want to express my gratitude to my associate adviser, Professor Fouad Tobagi, and to Professor Joseph Goodman, who served on my oral examination and reading committees. In addition, I would like to thank Professor Mary Baker, who served on my oral examination committee,

For their help, constructive discussions and friendship, my special thanks go to the Tiny Tera team: Rolf Muralt, Ritesh Ahuja, Pankaj Gupta, Shang-Tse Chuang (Da), Steve Lin, Pablo Molinero Fernandez, Amr A. Awadallah, and Youngmi Joo. Additionally, I thank Kersten Barney for her administrative help and Murray Warren for his help with proof reading.

Finally, I thank my wonderful wife, Wilawan. Words cannot convey my appreciation of her love, her encouragement, and most especially, for her willingness to put her life on hold while I pursued my Ph.D. Daddy also wants to thank Mark and Marisa for their love and for being so understanding about why Daddy could not play. For their devotion to my study, I dedicate this thesis to my wife and parents.

Contents

Abstract iv

Acknowledgments v

Contents vi

CHAPTER 1
Introduction 1

1 Background 1
1.1 Packet Switch Overview 3
1.2 Input vs. Output Queueing 5
1.3 Overcoming Head-of-Line Blocking 8
1.4 Virtual Output-Queued Switch 9
1.5 Cell Scheduling 11
1.5.1 Scheduling Problem 11
1.5.2 Non-weighted vs. Weighted Scheduling Algorithms 14
2 Previous Scheduling Work 14
2.1 Maxsize 15
2.2 PIM 16
2.3 *i*SLIP 17
2.4 Wave Front Arbiter 18
2.5 Deterministic Slot Allocation 20
2.6 Performance Comparison of Previous Work. 21
3 Motivation of Thesis 22
3.1 Low Throughput Due to Non-uniform Traffic 23

3.2 Scheduling Algorithm Complexity 24
4 Outline of Thesis 25
5 Basic Definitions 27

CHAPTER 2
The LQF and OCF Algorithms 29

1 Introduction 29
2 LQF 30
3 OCF 33
4 Iterative Algorithms: *i*LQF and *i*OCF 35
4.1 Implementation of *i*LQF and *i*OCF 35
4.1.1 Pipeline Technique 38
4.2 Performance of *i*LQF and *i*OCF 41

CHAPTER 3
The LPF Algorithm 43

1 Introduction 43
2 The LPF Algorithm 45
2.1 Request Weighting 45
2.2 An LPF Match: a Maxsize and a Maxweight Match 46
3 Using a Maxsize Algorithm to Find an LPF Match 46
3.1 Input and Output Presorting 48
3.2 Implementation Using a Maxsize Algorithm 50
4 Modified Maxsize Algorithm 50
4.1 The Modified Edmonds-Karp Algorithm 52
4.1.1 The Original Edmonds-Karp Algorithm 53
4.1.2 The Modified Algorithm 55
4.1.3 Largest-unmatched-Port-First Search (LPFS) 56
4.1.4 Running Time 57
4.1.5 Equivalency to LPF. 58
5 Performance Analysis of LPF without the Pipeline Delay 59
5.1 Stability Analysis 59
5.2 Simulation Results 59

5.2.1 Uniform Traffic 60
5.2.2 Non-uniform Traffic 61
6 Performance Analysis of LPF with the Pipeline Delay 64
6.1 Improving Scheduling Time by Pipelining 64
6.2 Stability Analysis 65
6.3 Simulation Results 66
6.3.1 Uniform Traffic 66
6.3.2 Non-uniform Traffic 68
7 Starvation Problem 70

CHAPTER 4
The OPF Algorithm 74

1 Introduction 74
2 The OPF Algorithm 75
3 Using a Maxsize Algorithm to Find an OPF Match 76
4 Performance Analysis of OPF without the Pipeline Delay 78
4.1 Stability Analysis 78
4.2 Simulation Results 78
4.2.1 Uniform Traffic 79
4.2.2 Non-uniform Traffic 81
5 Performance Analysis of OPF with the Pipeline Delay 83
5.1 Stability Analysis 83
5.2 Simulation Results 84
5.2.1 Uniform Traffic 84
5.2.2 Non-uniform Traffic 85
6 Starvation Problem 87

CHAPTER 5
The *i*LPF and
***i*OPF Algorithms** 89

1 Introduction 89
2 Approximating LPF and OPF 90
2.1 Three-step Algorithm 91
2.2 Double For-Loop Algorithm 92

3 Implementations 93
3.1 Sorters and Crossbars 93
3.2 Three-Step Algorithm 96
3.3 Double For-Loop Algorithm 98
4 Comparison with *i*LQF and *i*OCF 99
4.1 Hardware Complexity 99
4.2 Running Time 101
4.3 Performance Comparison using Simulation of the Three-Step and the Double For-Loop Algorithms 103
4.3.1 Uniform Traffic 103
4.3.2 Non-uniform Traffic 107
5 Matching Blocking Problem 107
6 Solution to the Matching Blocking Problem 110

CHAPTER 6
Conclusion 112

1 Summary 112
2 Future Research 114
2.1 QoS Integration 114
2.2 Distributed Algorithms 114
2.3 Output Queueing Emulation 115

REFERENCES 116

APPENDIX 1
Stability of an NxN Switch under OCF with Independent Arrivals 124

1 Definitions 124
2 Main Theorem 127
3 Proof 127

APPENDIX 2
Stability of NxN Switch under LPF with Independent Arrivals 133

1 Definitions 133

2 Stability without the Pipeline Delay 134
2.1 Main Theorem 134
2.2 Proof 134
3 Stability with the Pipeline Delay 141
3.1 Main Theorem 141
3.2 Proof 141

APPENDIX 3
LPF Theorems 145

1 LPF Match is a Maximum Size Match 145
2 Modified Edmonds-Karp Algorithm is Equivalent to LPF 146

APPENDIX 4
Stability of NxN Switch
under OPF with i.d. Arrivals 157

1 Definitions 157
2 Main Theorem 159
3 Proof 159
4 Stability with the Pipeline Delay 168
4.1 Main Theorem 168
4.2 Proof 168

CHAPTER 1

Introduction

1 Background

Congestion in high-speed backbone networks, most notably the Internet, has created a real need for high-speed switches and routers with high aggregate bandwidth. In an attempt to alleviate congestion, the networking research community has focused its effort on the development of high-bandwidth switches [2][6][12][15][50][61]. Advances in data transmission technology, particularly the deployment of optical technology, has provided abundant transmission bandwidth at a relatively low cost. Yet neither switches or routers have kept pace with this development. Hence, the bottlenecks in today's high-bandwidth networks are often switches and routers, not transmission links. The lack of network switching capacity, coupled with an explosive growth in both the number of users and the amount of traffic per user [20][44][55][56], has created excessive congestion on the Internet. As a consequence, real-time applications on the Internet often perform poorly because of excessive delay and packet loss as well as large delay variation [5][43].

High-bandwidth switches can increase network switching capacity in a number of ways. In addition to being used directly to form a fast network, high-bandwidth switches

can be used to improve routers. Increasingly, routers are designed around a central switched backplane [2][6][12][15][50][61]. Aggregate switching bandwidth as high as one Tb/s is achievable using cost-effective CMOS components [50]. But this does not come without a problem: limited memory bandwidth means that high-bandwidth switches must employ input-queueing [12][27][53]. However, as we will see in Section 1.2, input-queueing was once thought to be constrained to a $2 - \sqrt{2} \approx 58\%$ throughput limit due to head of line (HOL) blocking [30]. Fortunately, in recent years, there has been a considerable amount of work showing that a form of input-queueing (called virtual output-queueing[1] (VOQ)) can eliminate HOL blocking [2][51][66]. A switch employing such a queueing discipline is known as a *VOQ switch*.

VOQ switches, however, introduce another set of problems [33]. One of these problems is that VOQ switches require the use of a scheduler to configure the switch, deciding which input to connect to which output in each packet-time. Because the scheduler determines exactly when each packet is transferred across the switch, the scheduler essentially determines the performance of the switch. In this case, performance includes the throughput of the switch, the delay experienced by each packet and the number of packets lost due to buffer overflow. Unfortunately for high-bandwidth VOQ switches, all existing scheduling algorithms are either too complex to run at high speed [49] or only perform well under restricted and unrealistic conditions [54]. This thesis presents two new algorithms to solve both problems.

Before discussing the detail of these new algorithms, we first review the background and related issues of VOQ switches as well as introduce necessary definitions used later. Collectively, the following subsections give a comprehensive overview of various aspects of high-bandwidth switch design.

1. Also known as Dynamically-allocated multi-queue (DAMQ) [65].

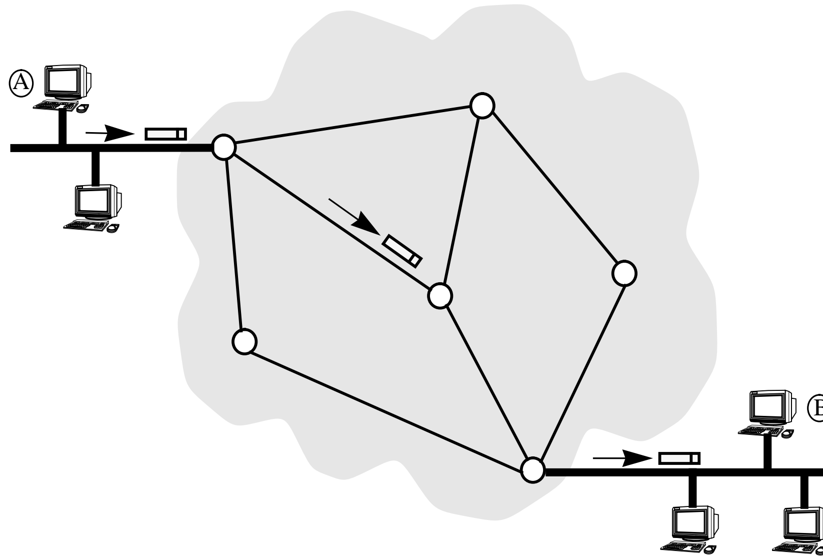


Figure 1.1 An example of a packet switched network. The nodes in the network cloud can be switches or routers. As they arrive at each node, packets sent from host A to host B get switched onto a path which leads them to host B.

1.1 Packet Switch Overview

In a packet switched network, the delivery of packets from point A to point B as illustrated in Figure 1.1 involves two basic tasks: routing and switching. Routing is done to determine a path which the packets must take to reach the destination. Switching, which takes place at every node on the path, is to switch — to place — the packets onto the path determined by a prior routing decision.

As shown by Figure 1.2, packet switching at each node (whether it is an Internet Protocol (IP) router [69] or a switch such as an ATM switch [3]) in a network includes packet buffering and packet forwarding. In this thesis, we often refer to a device which can perform both tasks as a packet switch or simply a switch. Upon its arrival at an input port, each packet is examined by the switch. From the header of a packet,[1] the switch deter-

1. A packet consists a header and a payload (data) [69].

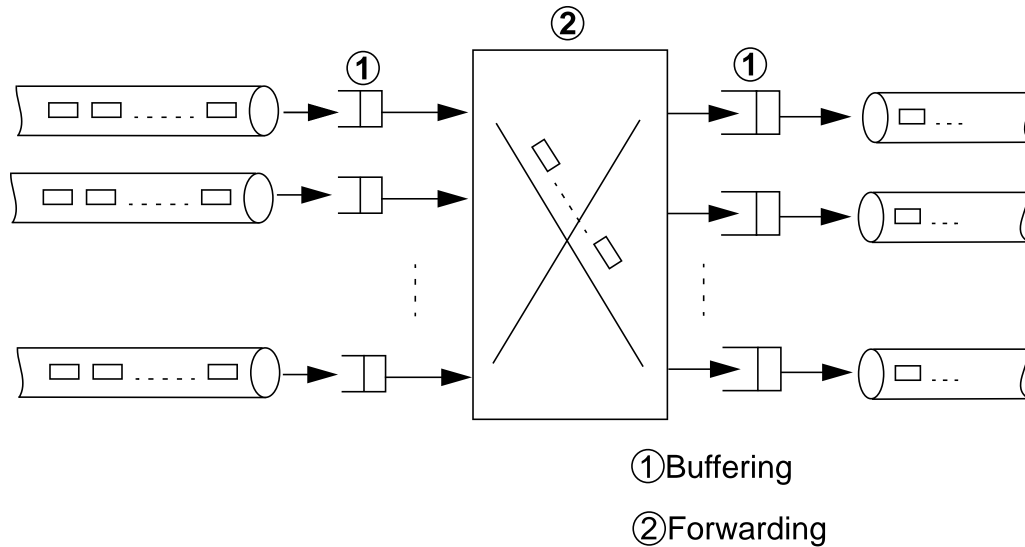


Figure 1.2 A packet switch. As packets arrive via the incoming links, the switch determines which output ports they should leave from and then tries to forward the packets to the outputs. Queues to buffer packets must be placed at every contention point to prevent packet dropping. Packet forwarding can be implemented using a space switch or a shared bus backplane.

mines to which output port the packet should be forwarded. Depending on its architecture, a switch may or may not be able to immediately forward all arriving packets to the outputs. Switches that cannot immediately forward all incoming packets must maintain queues at the inputs to buffer the remaining packets. Similarly, switches need to maintain some queues at the outputs if they cannot immediately place all forwarded packets onto the outgoing links. More detail of queueing considerations is discussed in the following subsection.

Because a packet switch implements the two tasks that a router also needs to perform [12][33][61], a router can take advantage of the high switching bandwidth already offered by a switch. Increasingly, high performance routers use a switch as a backplane to handle packet buffering and packet forwarding [2][12][53][61]. While packets arriving at a router may be of variable length, most high performance routers internally use fixed size packets

[2][12][31][53][61] that we shall refer to as *cells*.[1] Using cells simplifies the system design, allowing switches to run faster. In order to switch variable length packets, routers simply segment all incoming packets into cells as soon as the packets arrive at the input ports, and reassemble cells back into variable length packets at the outputs before the packets are sent out onto the transmission links.

1.2 Input vs. Output Queueing

One of the central concerns in designing a high-bandwidth switch is limited memory bandwidth [3][33][50]. For switches operating at high speed, the speed of the memory, which is a basic building block of queues, becomes a limiting factor. Switch speed is often limited by the rate at which the memory can operate [27][53]. Under this condition, a switch that makes an efficient use of memory bandwidth can run faster than one that does not. The memory bandwidth requirement varies widely across queueing disciplines [3][11]. Depending on switch architecture, queueing can take place at different parts of the switch: at the inputs, at the outputs, at both inputs and outputs, or at a centralized location. The following compares three queueing techniques: output-queueing, centralized shared memory and input-queueing.

Output-queueing is referred to as a queueing technique in which all queues are placed at the outputs as shown in Figure 1.3 (a) [3][64][69]. Switches employing this queueing technique are known as output-queued (OQ) switches. While it is known for its high throughput and ability to guarantee quality-of-service (QoS) [11][15][29][72], output-queueing does not make efficient use of memory bandwidth. Because there are no queues at the inputs, all arriving cells must be immediately delivered to their outputs. Being able to deliver every cell to the output immediately is both desirable and undesirable. In terms of the throughput and the QoS guarantee, it is advantageous because all cells immediately appear at the outputs ready to be considered for transmission, making QoS guarantee pos-

1. The term cell is borrowed from ATM. In this thesis, a cell size is not restricted to 53 bytes as in ATM.

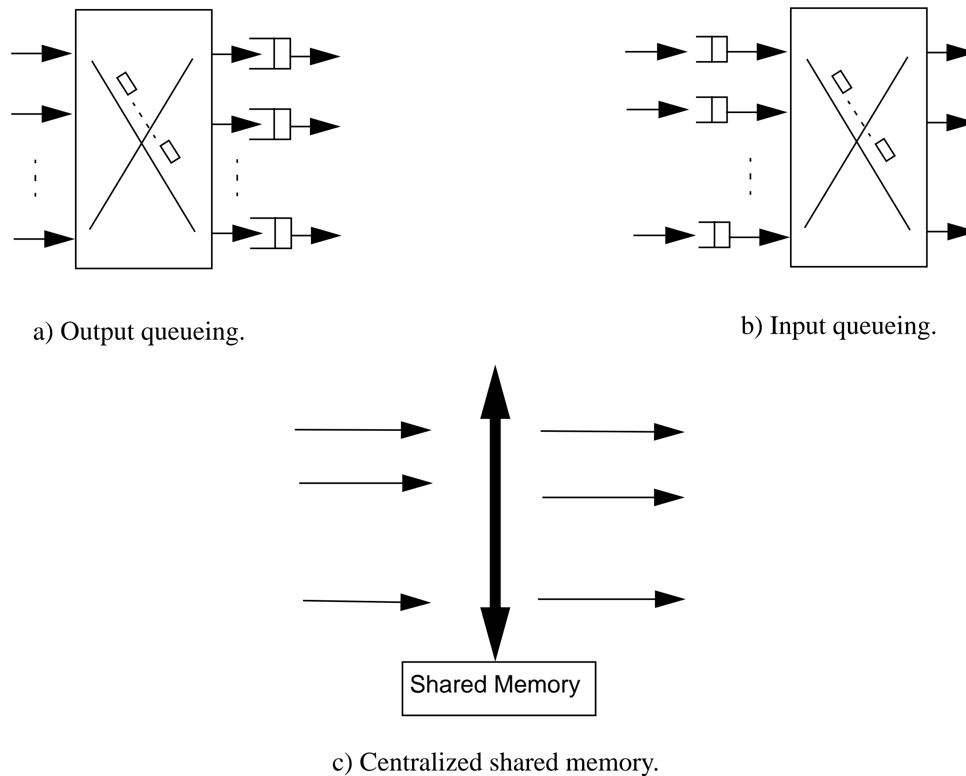


Figure 1.3 Various queueing techniques to buffer packets in a packet switch.

sible [16][72]. A major disadvantage, however, is that simultaneous delivery of all arriving cells to the outputs requires too much internal interconnection bandwidth and memory bandwidth. For an $N \times N$ switch, there can be up N cells, one from every input, arriving for any one output simultaneously. To be able to receive all N cells at one time, a memory implementing an output queue has to support N write accesses in a cell-time, as does the interconnection feeding into the memory. This requirement is known as *internal speedup* of a switch [11]. An output-queued switch, thus, has an internal speedup of N. With one read to send one cell to the outgoing link every cell-time, a memory implementing an output queue must operate $N+1$ times faster than the line rate.

Centralized shared memory is another queueing technique, commonly used in low bandwidth switches [14][19]. As shown in Figure 1.3 (c), the centralized memory is

shared by all inputs and all outputs. The access to the memory is time multiplexed. Each input and each output accesses the memory one at a time via a shared bus. Logically, the memory is partitioned into multiple queues, one for each output.[1] The partition can be static or dynamic [10]. In one cell-time, each input can write one cell into a queue corresponding to the output destination of the cell, and each output can read one cell, the HOL cell, from its queue. A switch employing this type of queueing is often known as a shared bus/shared memory switch [14][19]. This queueing technique has advantages and disadvantages. One of the advantages is that it has the same cell-by-cell behavior of output-queueing: logically, a shared memory scheme can be viewed as a form of output-queueing with all the output queues moved to a central location. Another advantage of a shared memory scheme is lower cell loss probability. With memory sharing, buffers that are unused by other outputs can be given to outputs under heavy load. A major disadvantage, however, is the speed at which a shared memory has to operate. Similar to output-queueing, a shared memory scheme is constrained by an internal speedup requirement. For an $N \times N$ switch, the memory and the bus must be able to support N read accesses by all the outputs and N write accesses by all the inputs in a cell-time, i.e., the bus and memory must operate $2N$ times faster than the line rate.

Input-queueing, on the other hand, has no speedup requirement. Queues at the inputs need not to be able to receive or send more than one cell simultaneously because at most one cell can arrive at and depart from each input in one cell-time. Thus, the memory is only required to operate at twice the line rate, making input-queueing of interest for high-bandwidth switches. Unfortunately, it is previously known that an input-queued (IQ) switch with a single FIFO queue at each input performs poorly due to head-of-line (HOL) blocking [30]. Nonetheless, this problem can be completely eliminated by a simple queueing technique.

1. The memory may be partitioned further: for example, one queue for each flow.

1.3 Overcoming Head-of-Line Blocking

Despite a long-standing practice of avoiding input-queueing because of an HOL blocking problem [3][69], recent work has found a simple technique to completely eliminate HOL blocking [2][49][66]. HOL blocking occurs because cells for different outputs share the same FIFO queue. When cells for different outputs share a FIFO queue, a cell which is destined to a free output can be blocked by a cell in front of it that is destined to a different output but has to remain in the queue because its output is busy. As a result, some inputs and outputs are unnecessarily left idle. Karol et al. showed that under certain conditions, HOL blocking results in a $(2-\sqrt{2}) \approx 58.6\%$ throughput limit [30].

Since then various techniques have been suggested to reduce HOL blocking [7][24][29]. However, one technique called virtual output queueing (VOQ) can completely eliminate HOL blocking. In a VOQ switch, an example of which is shown in Figure 1.4, all inputs maintain a simple queue structure consisting of multiple FIFO queues, one for each output. Now that all cells in each FIFO queue are destined for the same output, no cell can be blocked by a cell in front of it that is destined to a different output; in other words, no HOL blocking can occur. Although VOQ may appear complicated, memory bandwidth required to implement VOQ remains the same as a single FIFO queue because at most one cell can arrive at and depart from each input at a time. Using head and tail pointers, all queues at an input can share the same physical memory. Logical partitioning of the memory can be either static or dynamic [10].

Efficient memory bandwidth utilization, together with HOL blocking elimination, makes VOQ a viable solution for high-bandwidth switch design. However, the scheduling problem in VOQ switches is more complex than the one in single FIFO switches because VOQ switches maintain more queues at each input than single FIFO switches.

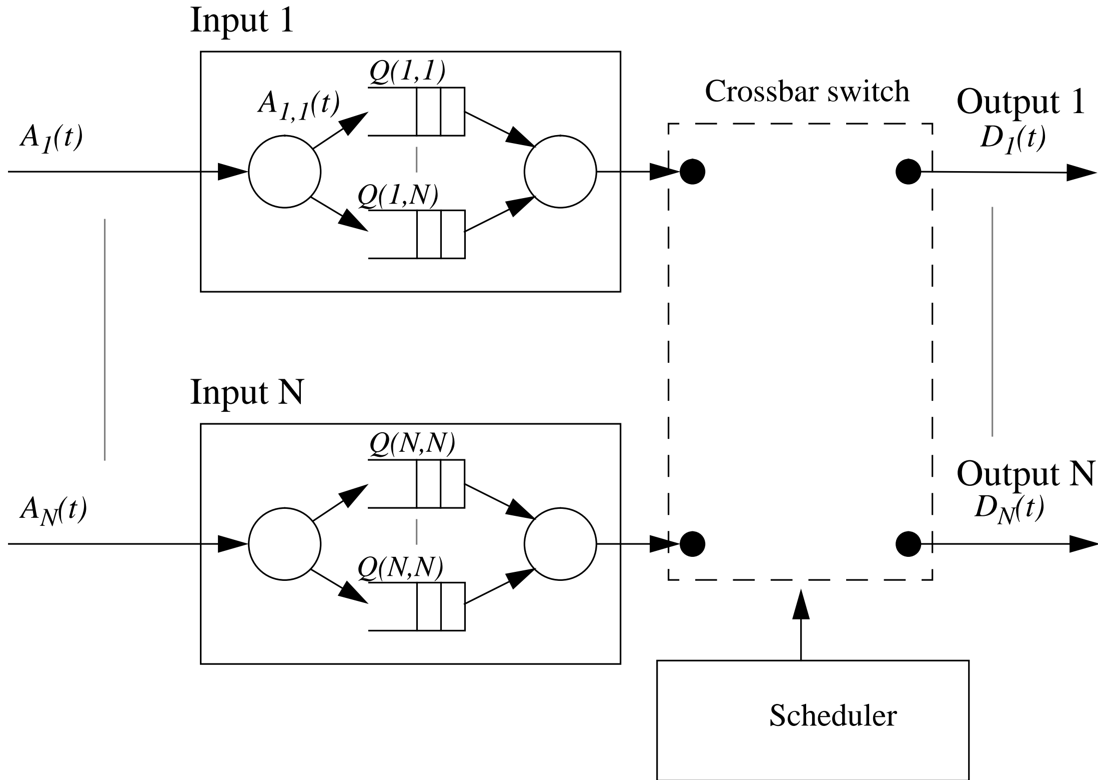


Figure 1.4 A simple model of VOQ switch consisting of three major components: a non-blocking switch fabric (e.g. crossbar) [3][25][69], a centralized scheduler and input-output ports.

1.4 Virtual Output-Queued Switch

A base-line VOQ switch is as shown in Figure 1.4. It consists of three main components: (i) N inputs and N outputs, (ii) a nonblocking switching fabric, and (iii) a scheduler. Input i maintains N FIFOs, $Q_{i,1}, Q_{i,2}, \ldots, Q_{i,N}$, one for each output. $A_{i,j}(n)$ is the discrete-time arrival process of cells at input i for output j while $A_i(n)$ is the aggregate arrival process at the input. Time is slotted into cell-times which we shall, hereafter, refer to as *slots*. During each slot, at most one cell can arrive at each input. Upon arrival, every cell identifies its output destination and thus is queued according to its output destination. $Q_{i,j}$ is a FIFO queue at input i to buffer cells waiting to go to output j. $\lambda_{i,j}$ is the arrival

rate of $A_{i,j}(n)$, and $A(n) = \{A_{i,j}(n); 1 \le i \le N, 1 \le j \le N\}$ is the set of all arrival processes,[1] which throughout this thesis is often referred to as the *arrival traffic*.

Definition 1: *An arrival process is said to be admissible when no input or output is oversubscribed, i.e., when* $\sum_{i=1}^{N} \lambda_{i,j} < 1, \sum_{j=1}^{N} \lambda_{i,j} < 1, \lambda_{i,j} \ge 0$.

Definition 2: *The traffic is uniform if all arrival processes have the same arrival rate, and destinations are uniformly distributed over all outputs, otherwise the traffic is non-uniform.*

Definition 3: *Traffic is called independent and identically distributed (i.i.d.) if and only if:*

1. *Every arrival is independent of all other arrivals both at the same input and at different inputs.*
2. *All arrivals at each input are identically distributed.*

During every slot, the scheduler examines all of the virtual output queues in the system. By considering only the non-empty queues, the scheduler selects a set of conflict-free paths from the set of inputs to the set of outputs. By "conflict-free," we mean that each input in connected to at most one output, and each output is connected to at most one input. The selected queues are then served, which causes them to dequeue their HOL cells and send them along the pre-established paths to the corresponding outputs where the cells can be transmitted on the outgoing link.

1. Unless otherwise stated, all arrival processes are stationary and ergodic.

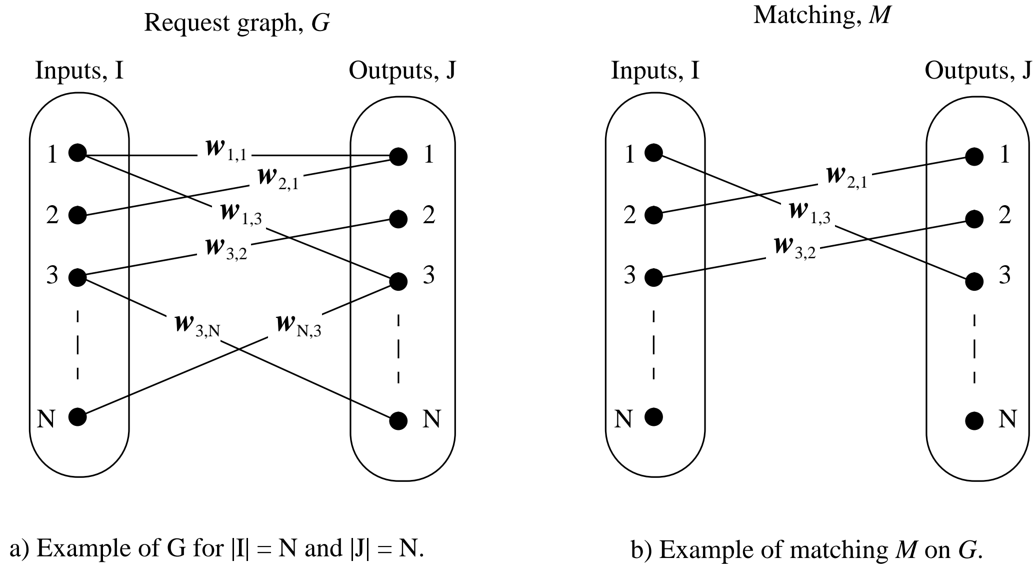


Figure 1.5 A request graph and a matching graph of an $N \times N$ switch. Define $G = [V,E]$ as an undirected graph connecting the set of vertices V with the set of edges E. The edge connecting vertices i, $1 \le i \le N$ and j, $1 \le j \le N$ has an associated weight denoted $w_{i,j}$. Graph G is bipartite if the set of inputs I = {i: $1 \le i \le N$} and outputs J = {i: $1 \le j \le N$} partition V such that every edge has one end in I and one end in J. Matching M on G is any subset of E such that no two edges in M have a common vertex.

1.5 Cell Scheduling

1.5.1 Scheduling Problem

Since in one slot each input can send at most one cell, and each output can receive at most one cell, the goal in cell scheduling is to find a one-to-one match between a non-empty VOQ and a free output. In other words, the scheduler matches an unmatched input to an unmatched output. In this scenario, every unmatched input makes requests to the scheduler telling it which outputs it wants to be matched, and along with every request it can give a weight to indicate its preference. Conceptually, the set of requests can be represented by a bipartite graph, called a *request graph*, illustrated in Figure 1.5 (a). Associated

with every edge is a weight $w_{i,j}$. Consequently, cell scheduling is equivalent to finding a bipartite graph matching [13][67].

Given a request graph G, the scheduling algorithm solves a bipartite graph matching problem to find a match graph M. To satisfy one-to-one matching, every node in M can have at most one edge incident as illustrated in Figure 1.5 (b). In M, a node is matched if it has an edge incident; otherwise it is unmatched.

Let $S_{i,j}(n)$ be a service indicator such that $\sum_{i=1}^{M} S_{i,j}(n) \le 1$ and $\sum_{j=1}^{N} S_{i,j}(n) \le 1$; a value of one indicates that input i is matched to output j, i.e., $Q_{i,j}$ is allowed to forward one cell to its output.

Definition 4: *A maximum size matching algorithm is an algorithm that finds a maximum size match, i.e., maximizes* $\sum_{i,j} S_{i,j}(n)$*, the number of connections.*

Definition 5: *A maximum weight matching algorithm is an algorithm that finds a maximum weight match, i.e., maximizes* $\sum_{i,j} S_{i,j}(n) w_{i,j}(n)$ *, the total weight.*

Definition 6: *For brevity, we shall refer to a maximum size matching algorithm as* ***maxsize algorithm****, and a maximum size match as* ***maxsize match****.*

$$\begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} & \cdots & w_{1,N} \\ w_{2,1} & w_{2,2} & w_{2,3} & \cdots & w_{2,N} \\ w_{3,1} & w_{3,2} & w_{3,3} & \cdots & w_{3,N} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ w_{N,1} & w_{N,2} & w_{N,3} & \cdots & w_{N,N} \end{bmatrix}$$

a) A weight matrix, $\omega(n)$, indicating a weight value assigned to each queue. Empty queues may have non-zero weights.

$$\begin{bmatrix} 1 & 0 & 1 & \cdots & 0 \\ 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 1 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 1 & \cdots & 0 \end{bmatrix}$$

b) A raw request matrix containing only 1's and 0's to denote whether queues are non-empty or empty, i.e., whether they make a request or not.

$$\begin{bmatrix} w_{1,1} & 0 & w_{1,3} & \cdots & 0 \\ w_{2,1} & 0 & 0 & \cdots & 0 \\ 0 & w_{3,2} & 0 & \cdots & w_{3,N} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & w_{N,3} & \cdots & 0 \end{bmatrix}$$

c) A weighted request matrix whose elements are the product of the corresponding elements of a raw request matrix and a weight matrix.

Figure 1.6 Request matrix formation of a $N \times N$ switch.

Definition 7: *For brevity, we shall refer to a maximum weight matching algorithm as **maxweight algorithm**, and a maximum weight match as **maxweight match**.*

Each request can be represented as the i, j th element of a request matrix, examples of which are shown in Figure 1.6 (b) and Figure 1.6 (c). We call a request matrix containing weighted requests a *weighted request matrix,* denoted as $R_{\omega}(n)$, and one containing unweighted requests a *un-weighted request matrix* or *raw request matrix,* denoted as $R(n)$.

1.5.2 Non-weighted vs. Weighted Scheduling Algorithms

Scheduling algorithms can be divided into two classes: weighted and non-weighted. Weighted algorithms consider request weights when making scheduling decisions while non-weighted algorithms do not. An example of a non-weighted algorithm is a maximum size matching algorithm [13][67], and an example for a weighted algorithm is a maximum weight matching algorithm [13][51][67].

For uniform traffic, ignoring request weights does not greatly affect the performance of the switch because all queues are likely to experience the same degree of congestion when traffic is uniform and hence likely to request the same preference. For non-uniform traffic, however, this is not the case. When the traffic is non-uniform, queue occupancies and cell delays can differ greatly from queue to queue. Queues experiencing heavy traffic can overflow while other queues with light traffic are empty most of the time. This circumstance is not only undesirable, but can result in a low throughput [51][54]. In order to prevent this, a scheduling algorithm should consider congestion conditions by giving preference to highly congested queues. But without considering request weights, non-weighted algorithms cannot give such preference. Weighted algorithms, on the other hand, can be made to consider congestion conditions. For instance, each request weight can reflect the occupancy of the corresponding queue or the waiting time of the cell at the head of the queue, and preference is then given to requests with high weights [51][52].

2 Previous Scheduling Work

Since the introduction of VOQ [65], a variety of scheduling algorithms have been proposed for VOQ switches [1][2][8][42][46][49][52][54][60][66]. Among these algorithms, Parallel Iterative matching (PIM) [2], *i*SLIP [49] and wave front arbiter (WFA) [8][66] were demonstrated to be practical for high-bandwidth switches and shown to achieve 100% or close to 100% throughput when the traffic is uniform. Introduced in 1993, PIM is

used in DEC's AN2, a giga-bit switch [2]. However, the use of randomizers, as we will discuss shortly, makes PIM too slow for tera-bit switches [15][50].[1] During the same time frame, Tamir proposed WFA [66]. As discussed in Section 2.4, WFA is relatively simple to implement and fast enough to operate in tera-bit switches. Then in 1995, McKeown proposed another simple, fast, and efficient algorithm called *i*SLIP, which overcomes the complexity problem of PIM and outperforms PIM in several ways [40][49]. Other derivative algorithms were also proposed in an attempt to improve upon these three algorithms [42][46].

Unfortunately, these algorithms were developed under the assumption that traffic is uniform. Thus, they are not equipped to handle non-uniform traffic: they are all non-weighted algorithms. As Section 3 will demonstrate, these algorithms can experience low throughput when traffic becomes non-uniform. Despite the differences in implementation and performance, all of these algorithms attempt to approximate a maxsize algorithm.

2.1 Maxsize

Among non-weighted matching algorithms, a maxsize algorithm is the one that achieves a maximum instantaneous throughput[2] for the same given set of requests. It always finds a match of maximum cardinality [67]. If more than one maximum size match exists, ties are broken randomly. For uniform traffic, a maxsize algorithm offers good performance: as shown by simulation results, it can achieve 100% throughput with the lowest average cell delay as compared to other non-weighted algorithms [1][2][8][46][49][60][66]. However, it is only of theoretical value: it is too slow for any practical use in high-bandwidth switches. For an $N \times N$ switch, the most efficient algorithm takes $O(N^{2.5})$ time to run [23]. Nonetheless, there are a number of algorithms

1. We refer to a tera-bit switch as a switch that have an aggregate bandwidth of close to or more than one Tb/s.
2. Defined as a number of cell forwarded in one slot.

approximating a maxsize algorithm which are fast and simple to implement in hardware. WFA, PIM and *i*SLIP are examples of such algorithms.

As proved in [51], a maxsize algorithm can lead to a low throughput under non-uniform traffic. Hence, all algorithms which attempt to approximate a maxsize algorithm suffer the same fate. In addition, selecting a maximum size match can lead to a starvation problem, in which some queues receive very little or no service [49].

2.2 PIM

The Parallel Iterative Matching (PIM) algorithm attempts to approximate a maximum size matching algorithm by iteratively matching the inputs to the outputs until it finds a *maximal*[1] size match [2][41]. In each iteration, it successively matches additional inputs and outputs until no more matches can be found. The first iteration begins with all inputs and outputs unmatched, and each iteration of PIM consists of the following three steps:

Step 1. *Request*. Every unmatched input sends a request to every output for which it has a queued cell.

Step 2. *Grant*. Among all received requests, an unmatched output chooses one randomly with equal probability.

Step 3. *Accept*. Each unmatched input, then, randomly accepts one granted request with equal probability.

Simulation[2] shows PIM to achieve 100% throughput for uniform traffic [2]. On average, PIM converges in $\log_2 N$ iterations, although in the worst case it may take up to N iterations [2][49]. For high-bandwidth switches, the number of iterations required becomes a deciding factor because there may not be enough time to complete them [40][50]. This is one of the areas in which *i*SLIP outperforms PIM [49]. Nonetheless, the main problem of PIM lies in randomizers required in the grant and accept steps. At high

1. A maximal size match is one which leaves no input unmatched unless all of the outputs it requests are matched.
2. For a 16×16 switch and 16 iterations.

speed, randomizers are relatively expensive and slow compared to the schemes used in *i*SLIP and WFA.

2.3 *i*SLIP

Similar to PIM, *i*SLIP approximates a maxsize algorithm by finding a maximal size match. It has been shown to achieve 100% throughput for uniform traffic with independent arrivals [49], yet it can be implemented using very simple hardware. Furthermore, *i*SLIP can achieve maximum throughput in just a single iteration, making it of interest for high-bandwidth switches. Additional iterations help reduce cell delay. Simulation results suggest that $\log_2 N$ iterations are sufficient for an $N \times N$ switch: more iterations beyond $\log_2 N$ do not substantially reduce cell delay [49].

*i*SLIP replaces the randomizers of PIM with round robin arbiters. Each input and each output has one round robin arbiter, and each arbiter uses a pointer to point to the highest priority output or input [49]. The use of round robin arbiters allows *i*SLIP to be simpler in hardware implementation and faster than PIM [2][40]. *i*SLIP's ability to achieve 100% throughput in one iteration, however, is based entirely on the way in which each independent round robin pointer is updated. McKeown observed that pointer synchronization — pointers point to the same input or output — can result in $1 - \frac{1}{e} \approx 63\,\%$ throughput limit for a single iteration [49]. *i*SLIP resolves the pointer synchronization while maintaining fairness by causing the pointers to *slip* with respect to each other [49].

Similar to PIM, *i*SLIP is an iterative algorithm and can be briefly described as follows. In each iteration, the arbitration is carried out in three steps:

Step 1. *Request*. Every unmatched input sends a request to every output for which it has a queued cell.

Step 2. *Grant*. An unmatched output, starting from the highest priority input, searches in round-robin fashion and chooses the first requesting input. The output, then, notifies each input whether or not its request was granted.

Step 3. *Accept*. An unmatched input, similarly, starting from the highest priority output, searches in round-robin fashion and accepts the first granting output. At this step, the pointer updating takes place. Only if this step is part of the first iteration, then the pointer at every accepting input is moved to one location beyond the output it accepted and the pointer at every accepted output is also moved to one location beyond its accepting input.

Because each output can grant to only one input and each input can only accept one output, the new location of each pointer is unique with respect to other newly updated pointers. This means that the larger the size of a match is, the more pointers are updated to unique locations, and the less synchronized the pointers become. Less pointer synchronization leads to a larger match size in the next cell-time.

Similar to a maxsize algorithm, which it attempts to approximate, *i*SLIP can achieve a low throughput when traffic is non-uniform. In an attempt to prevent a throughput loss under non-uniform traffic, threshold *i*SLIP and weighted *i*SLIP have been introduced [49]. However, no analysis has been carried out to suggest their performance under non-uniform traffic.

2.4 Wave Front Arbiter

Wave front arbiter (WFA) [8][9][66] is another algorithm to achieve 100% throughput for uniform traffic. Similar to *i*SLIP, WFA also approximates a maxsize algorithm and is simple to implement. For an $N \times N$ switch, the arbiter is made up of a two dimensional, $N \times N$ interconnected array of simple cells. An example for a 4×4 switch is illustrated by Figure 1.7 (a). The simplest implementation of WFA is as follows. Each cell contains a register holding a request indicator. Cell i, j is for VOQ $Q_{i,j}$. Priority is given strictly based on cell location. Cells above have higher priority than cells below, and cells to the left have higher priority than cells to the right. As shown in Figure 1.7 (a), each cell has two inputs: the top input indicating that some cell above has been matched, thereby capturing the output, and the left input indicating that some cell to the left has been matched,

thereby capturing the input. If none of its inputs is asserted and the cell has a request, the cell is then matched, i.e., the input and the output to which the cell belonged are matched. Each cell also has two outputs: the bottom output and the right output. The bottom output is asserted when there is a matched cell above, including itself, to signal to all cells below. The right output is asserted when there is a matched cell to the left, including itself, to signal to all cells to the right.

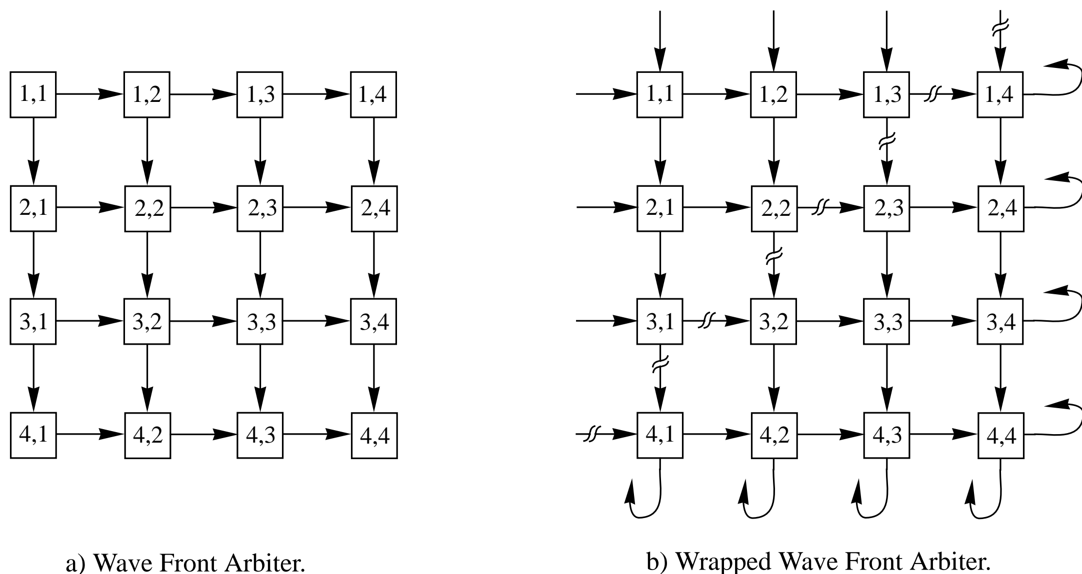


Figure 1.7 A simple diagram of Wave Front Arbiter and Wrapped Wave Front Arbiter for a 4×4 switch.

Because of the cell dependency described above, the arbitration time is $(2N–1)T$ time units, where T is the time for each cell to assert its outputs after receiving the inputs. Exploratory design-work suggests that a cell delay is approximately twice a two-input-NAND gate delay. Implemented in 0.25 μm CMOS technology with approximately 100-200 ps propagation delay for a two-input-NAND gate, the scheduling time for a 32×32 switch is approximately 12-24 ns.

Furthermore, the scheduling time can be reduced by half. By wrapping around the outputs of the bottom most cells and feeding them back to the inputs of the top most cells, and similarly wrapping and feeding the rightmost cells and leftmost cells as shown in Figure 1.7 (b), the wrapped wave front arbiter (WWFA) [8] is able to reduce the scheduling time to NT time units. The loops created by the wrapping described above are broken diagonally, allowing cell arbitration to progress in parallel like a wave front sweeping across the cell array. The broken points can be shifted every cell-time to give equal preference to all cells [8].

The performance analysis in [8][66] shows WWFA to experience higher cell delay than WFA because parallel cell arbitration sometimes makes a premature decision. WFA and WWFA, however, unlike *i*SLIP, lack fairness for some non-uniform patterns [41]. Like *i*SLIP, WFA and WWFA can experience a low throughput under non-uniform traffic.

2.5 Deterministic Slot Allocation

The Deterministic slot allocation (DSA) algorithm is an example of how simple it is to achieve 100% throughput for uniform traffic. For an $N \times N$ switch, DSA services each queue deterministically once every N cell-times. One possible implementation is as follows. At cell-time n, input i is matched to output $((i+n) \bmod N)$ *regardless of whether the input has any cell for the output or not.*

For an i.i.d. arrival process, each queue can be thought of as an M/D/1 system with a service rate of $\frac{1}{N}$ of the line rate. For uniform traffic, the maximum arrival rate at any queue is always less than $\frac{1}{N}$ of the line rate, and therefore because the service rate is greater than the arrival rate, the system is stable [34].

2.6 Performance Comparison of Previous Work.

Figure 1.8 illustrates the average latency as a function of the offered load to a 16×16 switch, and compares the scheduling algorithms described above for uniform traffic.

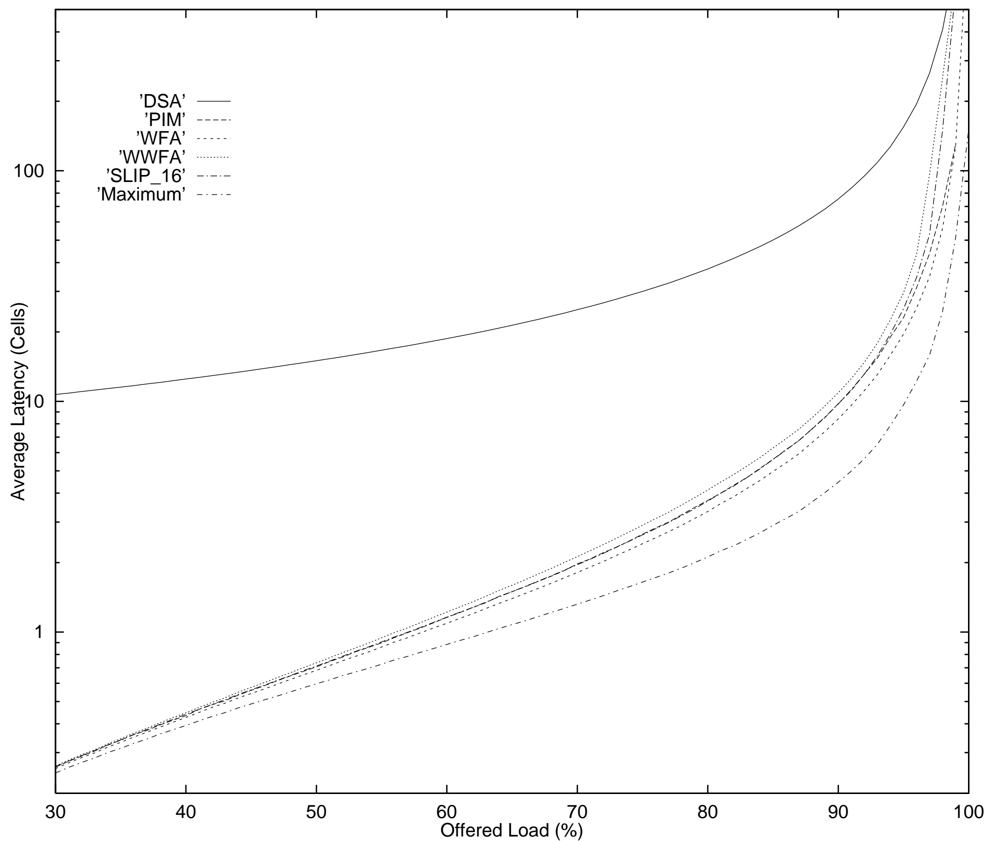


Figure 1.8 A graph of latency as a function of offered load of a 16×16 switch using various scheduling algorithms under uniform traffic. Arrivals at each input are Bernoulli i.i.d.

Among them, a maxsize algorithm leads to lower average latency. WFA achieves slightly lower latency than *i*SLIP. Nevertheless, the latency of WFA, WWFA, and *i*SLIP do not differ greatly. A switch using DSA algorithm, on the other hand, experiences a higher average delay at low offered load.

Apparently, this result shows this set of algorithms to achieve 100% throughput for uniform traffic. For non-uniform traffic, however, this is not the case: all of these algorithms can achieve less than a maximum throughput when traffic is not uniform. More detailed discussion is given in Section 3.

3 Motivation of Thesis

As VOQ switches have gained interest because of their ability to improve switching capacity by an order of magnitude [2][12][50][61], two problems with their existing scheduling algorithms have become known [49][51][52]. All existing algorithms either fail to maintain high throughput when the traffic becomes non-uniform [54] or are too complex to operate at high speed, becoming a speed bottleneck in the switch [33][40]. Shown in [51], a maxsize algorithm can experience low throughput when the traffic is non-uniform, so can algorithms approximating it as illustrated below. This problem is of concern because real traffic is always non-uniform.

Although there exist two algorithms that can avoid the low throughput problem [51][54], both, unfortunately, are too complex and too slow for high speed switches. As described in chapter 2, both algorithms are proved to achieve 100% throughput for all traffic patterns (uniform or non-uniform) in part because they select a maxweight match. But a maxweight algorithm needed to find a maxweight match is too complex for high speed switches.[1]

As speed and efficiency go hand in hand in designing a high bandwidth switch, this thesis aims to address and provide solutions to both problems. The following subsections outline the two problems.

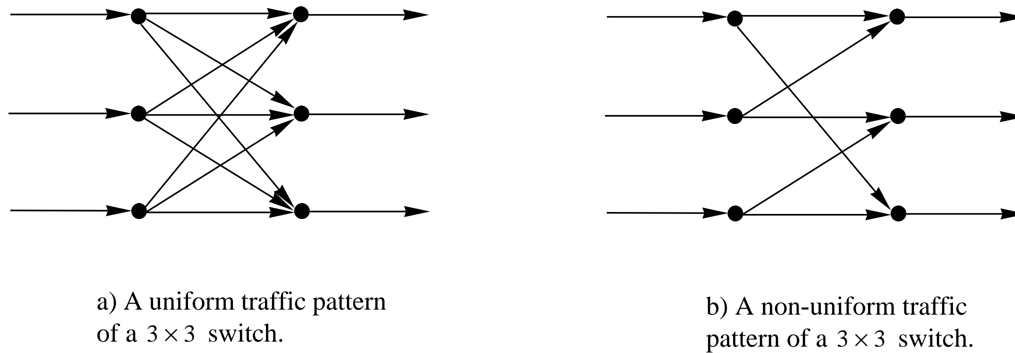
1. The fastest maxweight algorithm takes $O(N^3 \log_2 N)$ times to complete as compared to $O(N^{5/2})$ for the fastest maxsize algorithm.

3.1 Low Throughput Due to Non-uniform Traffic

For a large switch, the problem of using computer simulation to identify all non-uniform traffic patterns that would cause low throughput is extremely difficult: there are just too many patterns to simulate. For instance, consider a 32×32 switch with i.i.d. arrivals and with a restriction that $\sum_{i=1}^{N} \lambda_{i,j} = 1, \sum_{j=1}^{N} \lambda_{i,j} = 1, \lambda_{i,j} = 1, 0$. There are $32! > 10^{35}$ traffic patterns to consider, and without the restriction the number of possible traffic patterns would be much larger.

Nonetheless, for a switch with a few ports, it is possible to find a simple non-uniform traffic pattern which will cause low throughput. For PIM, *i*SLIP, WFA, DSA, we found a non-uniform traffic pattern — not necessarily the worst pattern — that causes low throughput in a 3×3 switch. Without the ability to see the artifacts of traffic such as the number of cells waiting in each queue or the queueing times of cells, these algorithms cannot tell which queues are likely to have traffic and which are not. So the 3×3 non-uniform traffic in Figure 1.9 (b) is created to confuse these algorithms by removing some flows from normally uniform traffic.

As expected and as indicated by the performance graph in Figure 1.10, these algorithms perform poorly under this traffic pattern. WWFA achieves only 67% of maximum throughput while *i*SLIP performs only slightly better with throughput limited to 71%. PIM and WFA perform significantly better than WWFA and *i*SLIP, but still suffer a throughput loss of more than 20%. For DSA, since it always services each flow one third of the time without consideration for requests, the maximum throughput therefore is two thirds of the maximum achievable throughput, 66.6% — almost the same as that of WWFA. Overall, these non-weighted algorithms exhibit various degrees of difficulty in handling the traffic, which at a glance seems to be easy to service. However, the outcome is different for



a) A uniform traffic pattern of a 3×3 switch.

b) A non-uniform traffic pattern of a 3×3 switch.

Figure 1.9 A construction of a non-uniform traffic pattern of a 3×3 switch. The pattern on the right is constructed by removing three flows, one from each input from the uniform pattern on the left. All flows between the inputs and outputs as indicated by the arrows are of an equal rate. Arrivals are Bernoulli i.i.d.

weighted algorithms. As will be shown later, iterative longest queue first[1] (*i*LQF) [49]and iterative longest port first[2] (*i*LPF) algorithms, both designed to handle non-uniform traffic, achieve 100% throughput for this traffic.

3.2 Scheduling Algorithm Complexity

In most cases, complexity is not of concern for non-weighted algorithms [8][9][53]. Complexity problems can arise in weighted algorithms due mainly to the need to consider request weights. Without careful consideration, weighted algorithms can be overly complex, which is the case for existing algorithms [49].

To illustrate the problem, we evaluate the amount of hardware in terms of the silicon area and the speed of two existing algorithms: *i*SLIP, a non-weighted algorithm, and *i*LQF, a weighted algorithm. Particularly, we compare a component called an arbiter which is used in both algorithms. *i*SLIP's arbiter does not consider or compare request weights while the other arbiter in *i*LQF does [40][49]. A logic synthesis result indicates that

1. Described in detail in Chapter 2.
2. Described in detail in Chapter 4.

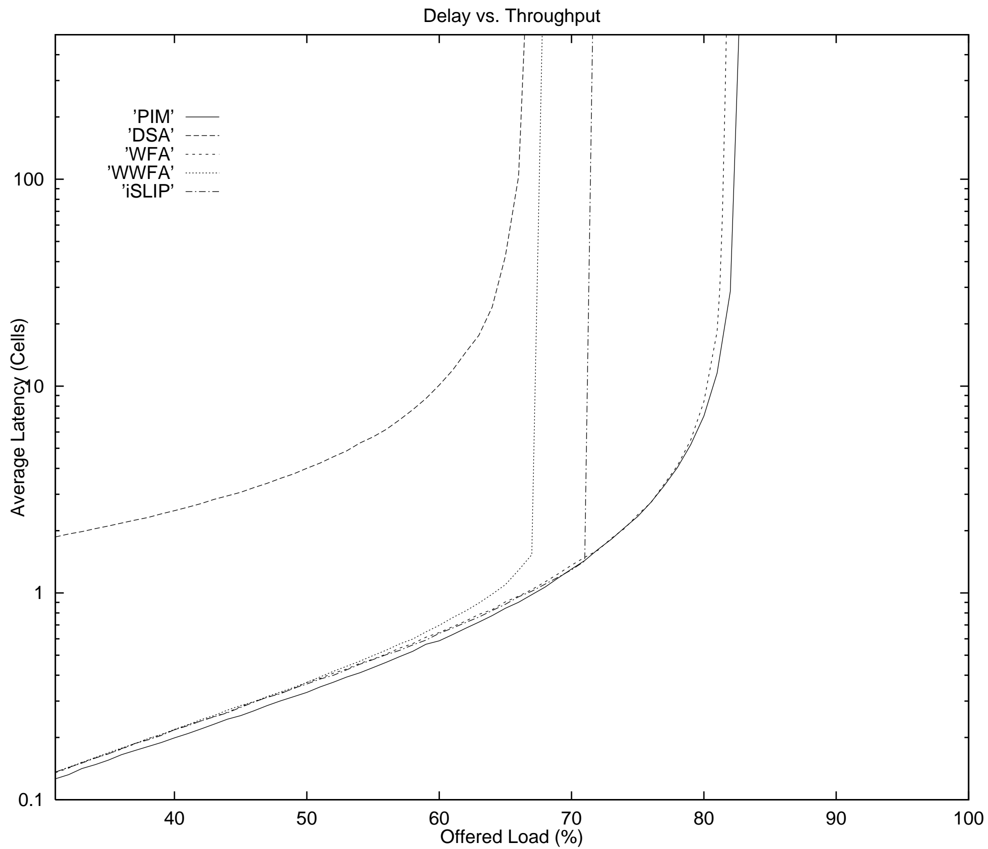


Figure 1.10 A graph of latency as a function of offered load of a 3×3 switch using various scheduling algorithms under non-uniform traffic pattern shown in Figure 1.9. Arrivals at each input are Bernoulli i.i.d.

*i*LQF's arbiter is approximately five times larger and four times slower than *i*SLIP's arbiter. *i*SLIP's arbiter takes 1.74 ns to run while *i*LQF's arbiter takes 7.38 ns. In term of silicon area, the size of *i*SLIP's arbiter is 603 units,[1] and the size of *i*LQF's arbiter is 3332.5 units.

4 Outline of Thesis

This thesis focuses mainly on cell scheduling for non-uniform traffic. Four algorithms are described in this thesis, and all are designed for non-uniform traffic. Our analysis shows that they all perform well under such traffic. In order to keep the chapters focused

1. One unit equals to the area of a two-input NAND gate.

on the algorithms and the concepts, only the related theorems are presented in the chapters, while the proofs are deferred to the appendices. An outline of each chapter is as follows.

Chapter 2 describes two maximum weight matching algorithms: the longest queue first (LQF) and the oldest cell first (OCF), which both achieve 100% throughput for all non-uniform traffic patterns with independent arrivals. They are the first algorithms proved to achieve such a throughput. LQF and OCF were initially proposed by McKeown [49]. McKeown proved that LQF can achieve 100% throughput under independent traffic both uniform and non-uniform [51], and conjectured that OCF can also achieve similar performance [49]. In this chapter, we prove that OCF can indeed achieve 100% throughput. Unlike LQF, OCF can never starve any input queue. But LQF and OCF are of only theoretical value: because of their high complexity, LQF and OCF are not suitable for high-speed implementation.

Chapter 3 presents a new algorithm called the longest port first (LPF), which is designed to overcome the complexity of LQF. By carefully selecting a weighting function, LPF always finds a match that is both a *maxsize* match and a *maxweight* match. Because of that, it is possible to implement LPF using a *maxsize* algorithm, which makes LPF less complex than LQF. Existing maxsize algorithms, however, cannot find an LPF match. In this chapter, we introduce a modified maxsize algorithm and prove that the algorithm, in fact, finds an LPF match. This chapter also looks at a pipelining technique to further reduce the running time of LPF. We find that it is possible to pipeline LPF and that the delay due to the number of pipeline stages does not lower the throughput. Using simulation results, we compare the performance of LPF with LQF and OCF, and finally, we outline the starvation problem of LPF.

Chapter 4 presents another new algorithm called the oldest port first (OPF), which is designed to lessen the starvation problem of LPF while maintaining the simplicity advantage of LPF. Similar to OCF, OPF avoids starvation by considering how long cells have been waiting in the queues. Just as LPF is simpler than LQF, OPF is also much simpler than OCF. Since OPF inherits all of the techniques used in LPF, this chapter focuses only on the differences between the two algorithms such as the weighting mechanism, starvation prevention, and performance comparison. Theorems concerning the stability of OPF are presented with the proofs deferred to the appendices.

Chapter 5 discusses implementable heuristic approximations to LPF and OPF. They are iterative algorithms called *i*LPF and *i*OPF, respectively. Accordingly, this chapter concentrates on design issues such as silicon area and speed. All design aspects from high-level architecture down to gate-level building blocks are presented in detail along with various design trade-offs.

5 Basic Definitions

The following definitions are used throughout this thesis.

1. The occupancy vector, representing the occupancy of each queue at time n:

$$\underline{L}(n) \equiv (L_{1,1}(n), \ldots, L_{1,N}(n), \ldots, L_{N,1}(n), \ldots, L_{N,N}(n))^T. \quad (1.1)$$

2. The waiting time vector, representing the waiting time of an head-of-line cell at each queue at time n:

$$\underline{W}(n) \equiv (W_{1,1}(n), \ldots, W_{1,N}(n), \ldots, W_{N,1}(n), \ldots, W_{N,N}(n))^T \quad (1.2)$$

3. The (constant) arrival rate matrix:

$$\Lambda \equiv [\lambda_{i,j}], \quad \text{where: } \sum_{i=1}^{N} \lambda_{i,j} \le 1, \sum_{j=1}^{N} \lambda_{i,j} \le 1, \lambda_{i,j} \ge 0 \quad (1.3)$$

and the associated rate vector:

$$\underline{\lambda} \equiv (\lambda_{1,1}, \ldots, \lambda_{1,N}, \ldots, \lambda_{N,1}, \ldots, \lambda_{N,N})^T. \quad (1.4)$$

4. The arrival matrix, representing the sequence of arrivals into each queue:

$$\mathbf{A}(n) \equiv [A_{i,j}(n)], \quad \text{where:} \quad A_{i,j}(n) \equiv \begin{cases} 1 & \text{if arrival occurs at } Q(i,j) \text{ at time } n \\ 0 & \text{else} \end{cases} \quad (1.5)$$

and the associated arrival vector:

$$\underline{A}(n) \equiv (A_{1,1}(n), \ldots, A_{1,N}(n), \ldots, A_{N,1}(n), \ldots, A_{N,N}(n))^T. \quad (1.6)$$

5. The inter-arrive time vector whose each element represents the inter-arrival between the current HOL cell and the cell behind.

$$\underline{\tau}(n) \equiv (\tau_{1,1}(n), \ldots, \tau_{1,N}(n), \ldots, \tau_{N,1}(n), \ldots, \tau_{N,N}(n))^T. \quad (1.7)$$

6. The service matrix, indicating which queues are served at time n:

$$\mathbf{S}(n) \equiv [S_{i,j}(n)], \quad \text{where:} \quad S_{i,j}(n) = \begin{cases} 1 & \text{if } Q(i,j) \text{ is served at time } n \\ 0 & \text{else} \end{cases} \quad (1.8)$$

and $\mathbf{S}(n) \in \mathbf{S}$, the set of service matrices.

Note that: $\sum_{i=1}^{N} S_{i,j}(n) = \sum_{j=1}^{N} S_{i,j}(n) = 1$

and hence $\mathbf{S}(n) \in \mathbf{S}$ is a *permutation matrix*.

We define the associated service vector as:

$$\underline{S}(n) \equiv (S_{1,1}(n), \ldots, S_{1,N}(n), \ldots, S_{N,1}(n), \ldots, S_{N,N}(n))^T, \quad (1.9)$$

hence $\|\underline{S}(n)\|^2 = N$.

CHAPTER 2

The LQF and OCF Algorithms

1 Introduction

The Longest Queue First (LQF) algorithm was the first algorithm that directly addressed the problem of low throughput due to non-uniform traffic, and is the first scheduling algorithm proven to enable input-queued switches to achieve the same throughput as output-queued switches [29][51]. LQF achieves 100% throughput for all independent arrival processes by giving preferential service to queues with high occupancies [51]. Giving preference to longer virtual output queues allows LQF to work well in keeping the occupancies relatively well-balanced, and reducing the possibility of overflow. Unfortunately, the consequence of giving preference to high occupancy queues is that queues with low occupancies can receive very little or no service, i.e., they can be starved [49][52].

The Oldest Cell First (OCF) algorithm is also designed to provide high throughput when traffic is non-uniform and, like LQF, can achieve 100% throughput for all independent arrival processes. Designed to overcome the starvation problem of LQF, OCF gives preferential service based on the waiting times of cells rather than queue occupancies. The reason for this is simple: no cell can wait for service indefinitely because all unserved cells

will eventually become old enough to be served [52]. Despite the difference in the approaches, both algorithms share a common strategy: they maintain a maximum throughput throughout a full range of traffic patterns by considering congestion indicators in the switch [51][52].

Both algorithms were first introduced by McKeown in [49] where it was proved that LQF can achieve 100% throughput for any traffic pattern with independent arrivals [51]. Simulation results in [49] suggest that OCF can achieve 100% throughput. In this chapter, we prove that OCF can achieve 100% throughput for all independent arrival processes.

Although the practicality of LQF and OCF in a high-bandwidth switch is limited by their complexity [49][54], this chapter describes both LQF and OCF because of their importance in demonstrating the concept of scheduling non-uniform traffic.

2 LQF

LQF is a weighted algorithm: LQF uses a maxweight algorithm to give preference to more heavily occupied virtual output queues. Each request weight is set to the corresponding queue occupancy and is defined as follows:

$$w_{i,j}(n) = L_{i,j}(n), \tag{2.1}$$

where $w_{i,j}(n)$ is the current weight of a request from input i to output j. During every slot, LQF selects a match M that maximizes the total weight of matched requests, $\sum_{i,j \in M} w_{i,j}(n)$.

It is intuitive, perhaps obvious, that LQF will help alleviate congestion in a switch and hence reduce overflow by giving preference to longer queues. It is more difficult, however, to prove that LQF can maximize the throughput of the switch for all traffic patterns. As

described in Chapter 1, this cannot be verified by exhaustively simulating all traffic patterns; there are too many. The situation, thus, requires an analytical approach. Using the method of Lyapunov functions [22][32], it was proved in [49] that LQF can achieve 100% throughput for all traffic patterns if arrivals are independent.

How LQF achieves high throughput for non-uniform traffic can be intuitively explained as follows. A switch is more able to transfer a large number of cells (i.e. connect a large number of inputs and outputs) in any one slot[1] if:

1. The scheduler can find a large-sized match for the current request graph. This, in turn is aided by:
2. The presence of queued cells at as many inputs and for as many outputs as possible, creating a highly populated request graph. This makes it easier for the scheduler to match multiple input-output pairs (to transfer many cells simultaneously).

Figure 2.1 illustrates how a highly populated request graph (i.e. having many non-empty queues) can lead to a high throughput.

Therefore, to sustain high throughput (measured over a long period of time), it is desirable that a scheduler does not simply maximize the number of cells transferred in each slot. If it did, it may unnecessarily cause some queues to become empty while others contain many cells. This has the effect of creating a sparse request graph like Figure 2.1 (b), hence making throughput lower in subsequent slots. By giving preference to more occupied queues, LQF avoids this problem.

We may look at the problem of achieving high throughput another way. While it is tempting to assume that if we select the largest size match (i.e. the highest instantaneous throughput) in each slot, doing so would have a sparse request graph for future slots. Con-

1. As mentioned in Chapter 1, an instantaneous throughput is a number of cells transferred in the current slot.

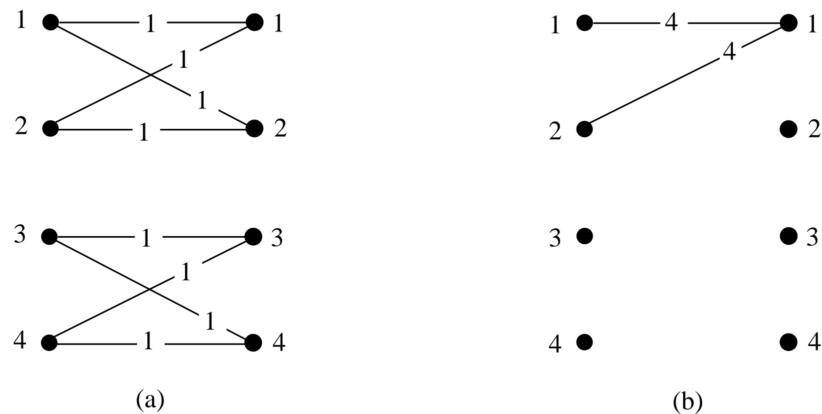


Figure 2.1 Two request graphs for a 4×4 switch with the numbers (on the edges) indicated the queue occupancies. Although both graphs have the same number of total queued cells, the one on the left is more populated and hence permits a match size of four while the one on the right is sparsely populated and allows only a match size of one. Therefore, in the best case, it will take two slots to forward all the cells for the graph on the left, and eight slots for the graph on the right.

sider the bar graph in Figure 2.2 (a) that shows the queue occupancies of the switch in Figure 2.2 (b). If the scheduler chooses a maxsize match of size four, the VOQs (1,1), (2,2), (3,3), (4,4) will become empty. making the graph more sparse, and reducing the size of subsequent matches. If, instead, the scheduler maximizes the weight of the match, the "tall" VOQs are "pushed" downwards, while "shorter" VOQs are allowed to grow. Over time, this tends to balance the occupancies of the (active) VOQs, making large matches possible. In fact, when the occupancies are precisely balanced, the maximum weight match equals the maximum size match.

Unfortunately, LQF is impractical. Using a maxweight algorithm makes LQF too complex for high speed implementation in hardware. The fastest maxweight algorithm known to date has a running time complexity of $O(N^3 \log N)$ [17]. Moreover, a maxweight algorithm needs multi-bit integer comparators in order to compare request weights [67]. As we will see later, the comparators are the primary source of the complexity problem.

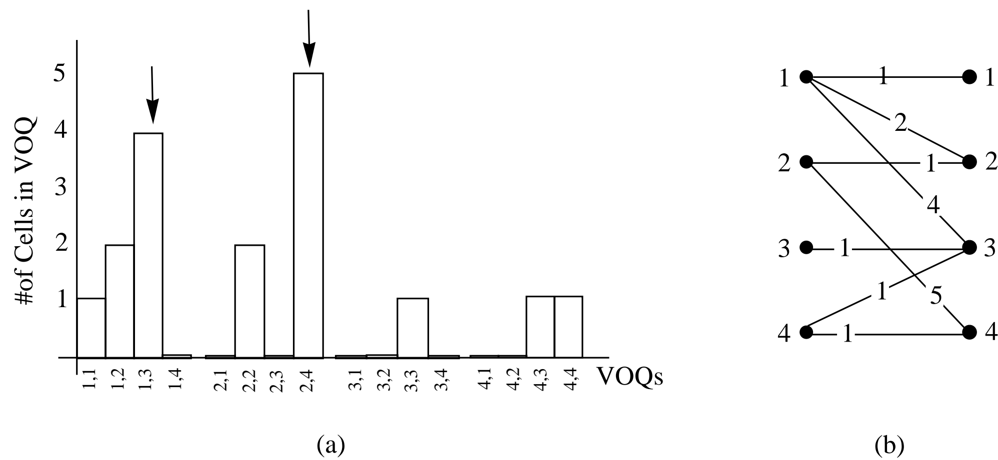


Figure 2.2 (a) A bar graph showing VOQ occupancies of a 4×4 switch; (b) The corresponding request graph. The arrows indicate that the corresponding VOQs are pushed downwards (serviced) if a maxweight match is chosen.

Another shortfall of LQF is its potential to cause starvation [49][52]. Starvation is highly undesirable because cells can wait indefinitely in their queues and never reach their destinations. Starvation can be easily demonstrated using a simple example of a 2×2 switch as shown in Figure 2.3.

3 OCF

Queue occupancies are not the only artifacts of congestion in a switch; other indicators such as the waiting times of cells in the queues also indicate the degree of congestion [34]. This is the principle on which OCF operates. Instead of giving preference to flows based on their queue occupancies, OCF gives preference by using the waiting times of HOL cells as request weights as defined below:

$$w_{i,j}(n) = W_{i,j}(n), \tag{2.2}$$

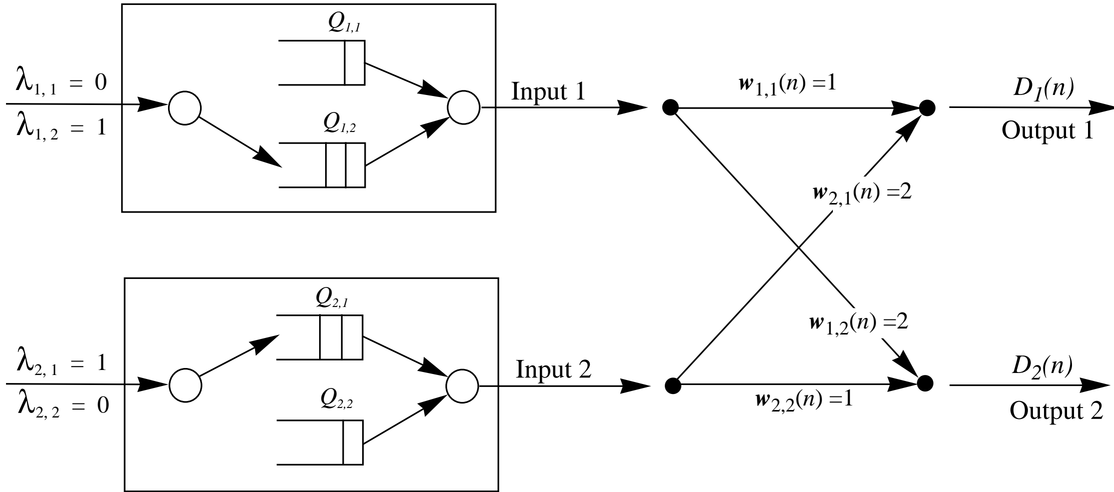


Figure 2.3 An example of starvation in a 2×2 switch for which, using LQF, input queues may be starved. Begin with one cell in $Q_{1,1}$ and $Q_{2,2}$, two cells in $Q_{1,2}$ and $Q_{2,1}$. Assuming, no arrival to $Q_{1,1}$ and $Q_{2,2}$, and one arrival to $Q_{1,2}$ and $Q_{2,1}$ in every slot, the occupancies of $Q_{1,1}$ and $Q_{2,2}$ will never be more than one while the occupancies of $Q_{1,2}$ and $Q_{2,1}$ will never be less than two. As a consequence, LQF always serves $Q_{1,2}$ and $Q_{2,1}$ because they give the largest total weight. The cells waiting in $Q_{1,1}$ and $Q_{2,2}$ are therefore starved.

where $W_{i,j}(n)$ is the waiting time of the HOL cell of $Q_{i,j}$. Like LQF, OCF uses a max-weight algorithm to find a match of a maxweight.

Giving preference based on the waiting times offers OCF a great benefit: no cell can remain unserved indefinitely because eventually every cell will become sufficiently old to warrant the service. Hence, by design, OCF can never starve a flow. To some degree, this makes OCF fairer than LQF: the variation of the waiting times of cells among different flows under OCF is smaller than that under LQF [49].

Although the service policy of OCF does not directly attempt to balance the occupancies, OCF indirectly achieves the same goal by balancing the waiting times. As stated by

Property 2 in Appendix 1, the waiting time of an HOL cell is always greater than or equal to the queue occupancy. Based on this principle, Appendix 1 shows that OCF can achieve 100% throughput.

Theorem 1: *Under the OCF algorithm, the queue occupancies are stable for all admissible and independent arrival processes, i.e.,* $E[\|\underline{L}(n)\|] \le C < \infty$.

Proof: *The proof is given in Appendix 1.*

As in the case of LQF, using a maxweight algorithm makes OCF too complex to implement in fast and simple hardware, and hence unsuitable for use in high-bandwidth switches. As we will see, even an iterative algorithm that approximates LQF and OCF is very complex.

4 Iterative Algorithms: *i*LQF and *i*OCF

Like most theoretical scheduling algorithms, LQF and OCF can be adapted to operate at higher speed. In the same ways that existing algorithms such as PIM and *i*SLIP approximate a maxsize algorithm, a maxweight algorithm used in LQF and OCF can be approximated by an iterative algorithm. In our iterative versions (called *i*LQF and *i*OCF), each iteration consists of three steps: request, grant and accept. Figure 2.4 outlines the algorithms, which were first introduced in [49]. Initially, they look similar to *i*SLIP and PIM. However, the weight comparisons in step 2 and 3 make them more complex than *i*SLIP [49].

4.1 Implementation of *i*LQF and *i*OCF

A direct implementation of the three-step algorithm in Figure 2.4 is shown in Figure 2.5. Since speed is often the main concern for high-bandwidth switches and since the study of gate-count is already covered extensively in [49], the following discussion concentrates on the running time of the algorithm. For all iterative algorithms, the running

***i*LQF and *i*OCF**

Step 1. Request. Every unmatched input makes a request to every output for which it has a cell destined. Every request carries a weight equalling to the associated queue length for *i*LQF or the waiting time of HOL cell for *i*OCF.

Step 2. Grant. Each output grants to the largest request. Ties are broken randomly.

Step 3. Accept. Each input accepts a grant to the largest request. Similarly, Ties are broken randomly.

Repeat Step 1-3 for N iterations or until the previous iteration found no more matches.

Figure 2.4 An iterative algorithm approximating LQF or OCF. Each iteration consists of three steps: request, grant and accept. To approximate a maxweight algorithm, each output grants to a request with the largest weight, and each input accepts a granted request also with the largest weight. The iteration terminated when the previous iteration found no more matches or the N th iteration is completed.

time depends on two factors: the number of iterations and the time per iteration. For *i*LQF and *i*OCF, in the worst case, it can take up to N iterations to find an optimal match for an $N \times N$ switch [49]. But in practice, it has been found that only $\log_2 N$ iterations are needed to achieve close to the optimal performance [49].

The time per iteration, however, depends on the implementation. For the implementation in Figure 2.5, the iteration time can be simply calculated by adding up the running times of all functional blocks along the loop shown by the dotted lines. Among all components in the loop, the grant arbiters and the accept arbiters dominate the iteration time.[1] Both arbiters, each a modified N-input magnitude comparator, are slow due to the large number of input values that they need to compare and the relatively high complexity of the basic building block, a two-input integer comparator [71]. Figure 2.6 shows an example of

1. A logic synthesis result indicates that other blocks take less than 1 ns to complete while each grant and accept arbiter takes more than 7 ns (for a 32×32 switch with ten bits representing each weight implemented in Texas Instruments' 0.25 μm CMOS technology).

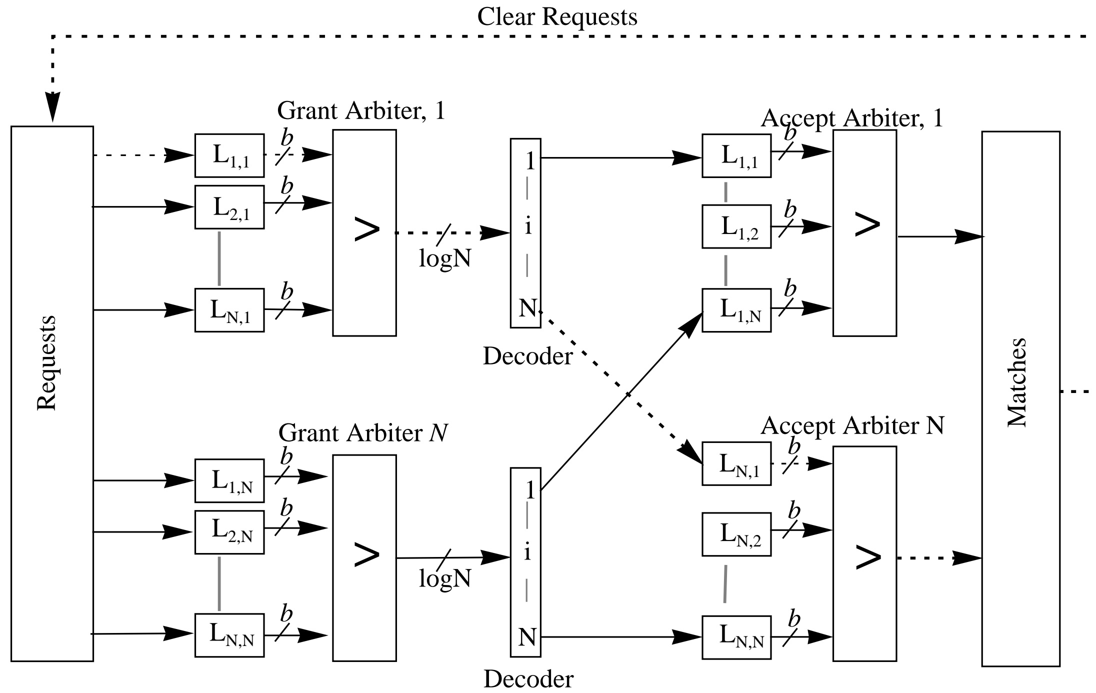


Figure 2.5 A block diagram of *i*LQF. An iteration starts from the requests block containing registers that hold request indicators and is completed by the feedback from the matches block. Between the requests block and the matches block, the iteration progresses through the set of registers holding request weights (in this case, queue occupancies $L_{i,j}$), the grant arbiters, the decoder and the accept arbiters. Upon completion, the match result is fed back to clear the requests of matched inputs and all requests to matched outputs. The critical path of the iteration is shown by the dotted lines.

an arbiter for a 4×4 switch. For instance, in one design,[1] the running time of the arbiters is approximately 75% of the iteration time. Even with the fastest implementation using carry-look-ahead, the two-input integer comparator still takes $O(\log b)$ time units to complete the comparison [13], where b is the number of bits of each input equalling to $\log L_{max}$.[2] For a binary tree implementation, which is the fastest implementation, comparing N values requires $\log_2 N$ stages of two-input integer comparators [25]. Thus, for an

1. For a 32×32 switch implemented in 0.25 μm CMOS technology.
2. L_{max} is the queue size, which is also the maximum value of queue occupancies.

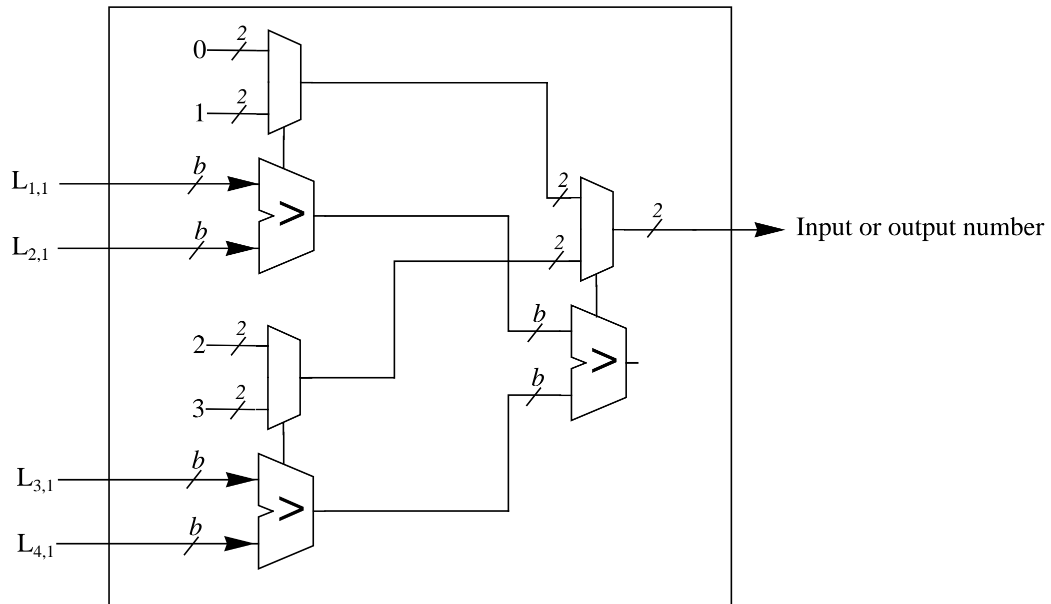


Figure 2.6 A schematic of a 4-input grant or accept arbiter. The basic components are comprised of two-input integer comparators [71] and two-input MUXes [48][71]. The four inputs to the arbiter are request weights which can be either queue occupancies or the waiting times of cells. The inputs are fed directly to the first stage integer comparators. The inputs into the first stage MUXes are the input or output numbers. Each comparator outputs the largest value of its inputs and then controls the selection of the MUX. For instance, if $L_{1,1} > L_{1,2}$, the top left comparator outputs $L_{1,1}$ and signals the MUX above to select the first input or output number whose queue occupancy is $L_{1,1}$. Likewise, the second stage compares the given queue occupanies and outputs the input or output number associated with the largest queue occupancy.

$N \times N$ switch, the best running time of each arbiter is $O(\log b \cdot \log N)$, compared to $O(\log N)$ for *i*SLIP [49].

4.1.1 Pipeline Technique

Despite the problem in the running time, there exists a simple improvement: a pipeline technique originally developed for *i*SLIP in the Tiny-Tera project [41]. Although it does not completely solve the problem, the technique can reduce the running time by almost half. This technique allows the grant arbiters and the accept arbiters to operate in parallel, effectively hiding the accept arbitration time. As shown in Figure 2.7, the basic concept is as follows.

Normally, in the non-pipeline case, granting of the next iteration must wait for the accepting of the current iteration to complete so that the matching result can be used to clear the requests of recently matched inputs, i.e., previously matched inputs should not continue to make requests. But clearing the requests does not have to wait until the accept step has completed. In order to allow the grant arbiters to proceed, the only information needed is *which* inputs will be matched in the current accept phase. To determine which inputs will be matched, the scheduler does not need to wait for the completion of the accept arbitration. An input is definitely matched to some output if at least one output grants its requests. Therefore, a fast way to find out whether or not an input will be matched is to detect if it receives *any* grant. This can be simply accomplished by ORing all incoming grant signals prior to performing the accept arbitration as shown in Figure 2.7 using the grant detecting blocks.

Similarly, each matched output should not continue to grant in later iterations. Preventing matched outputs from granting can be carried out in two ways. In the non-pipeline case, the approach is to clear all requests to all matched outputs at the request stage so that the outputs receive no request, and therefore do not grant. A drawback with this approach is that granting of the next iteration cannot proceed until a match of the current iteration is available, i.e., until the accept is complete. A more efficient approach is to allow the grant arbiters to continue regardless of the matching result. Stopping the grant arbiters from granting, in this case, is carried out by nulling their outputs after the accept arbitration has completed. As shown in Figure 2.7, nulling the grant outputs is accomplished by using maskable decoders. The added grant detectors and the maskable decoders do not add much silicon area or increase the arbitration time. Each grant detector is just an N-input OR gate, and masking the outputs of each decoder requires N 2-input AND gates.

In spite of the improvement above, *i*LQF and *i*OCF still remain slow for high-bandwidth switches. For a 32×32 switch, a logic synthesis result suggests that each arbiter

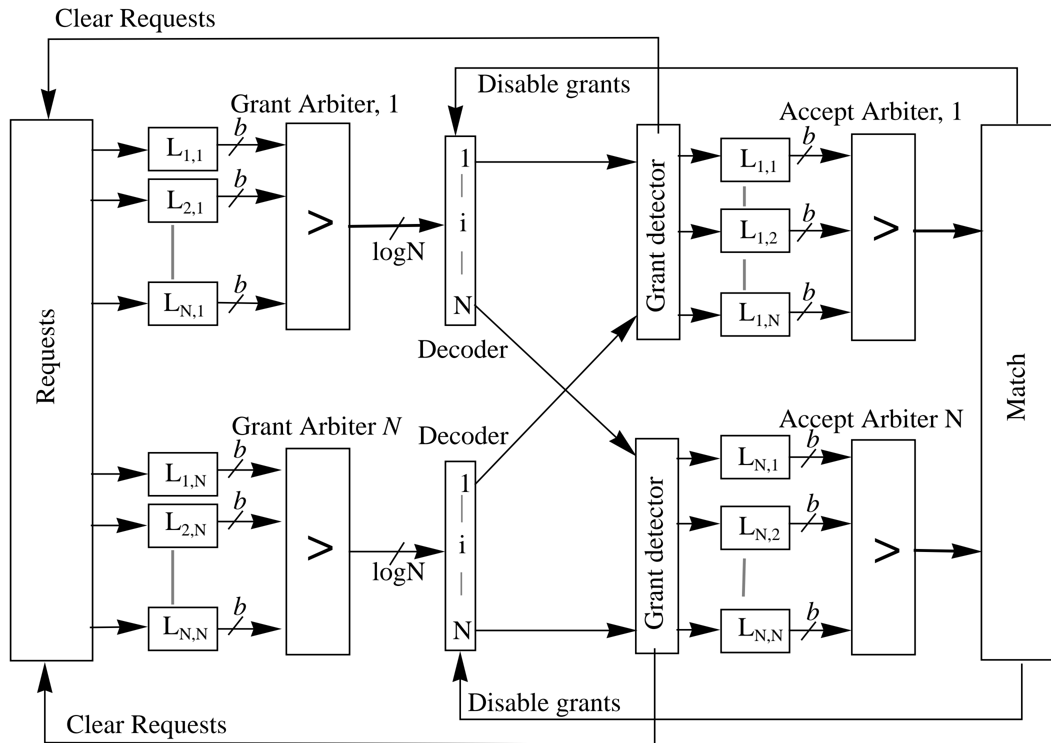


Figure 2.7 *i*LQF with pipelining. The design is similar to the one in Figure 2.5 except for the use of the grant detectors and the maskable decoders to allow the pipelining of the scheduler. Each grant detector determines whether or not the input will be matched by detecting the presence of its all incoming grant signals. Once it detects that the input will be matched, the grant detector tells the request block to immediately clear all requests of the input. The granting of matched outputs is stopped by disabling the outputs of the grant arbiters at the decoders. With this added mechanism, the grant arbiters and the accept arbiters can operate in parallel.

would take a silicon area equivalent to approximately 6,000 standard inverters and a running time of more than 7 ns.[1] As a result, $\log_2 N$ iterations, the scheduling time would be at least 35 ns for $N = 32$. Arguably, this is not fast enough for high-bandwidth switches [50]. As mentioned in Chapter 1, our goal is to arrive at a scheduling decision within 10-20 ns.

1. Using 0.25 μm CMOS process.

4.2 Performance of *i*LQF and *i*OCF

Despite their implementation complexity, *i*LQF and *i*OCF can achieve good performance. As compared to the non-weighted algorithms discussed in Chapter 1, *i*LQF and *i*OCF achieve high throughput for a broad range of traffic patterns. Shown by simulation results in [49], *i*LQF and *i*OCF perform well under both uniform traffic and nonuniform traffic. To illustrate the performance of *i*LQF and *i*OCF under non-uniform traffic, Figure 2.8 compares the average cell delay of *i*LQF and *i*OCF to *i*SLIP and WFA using the same

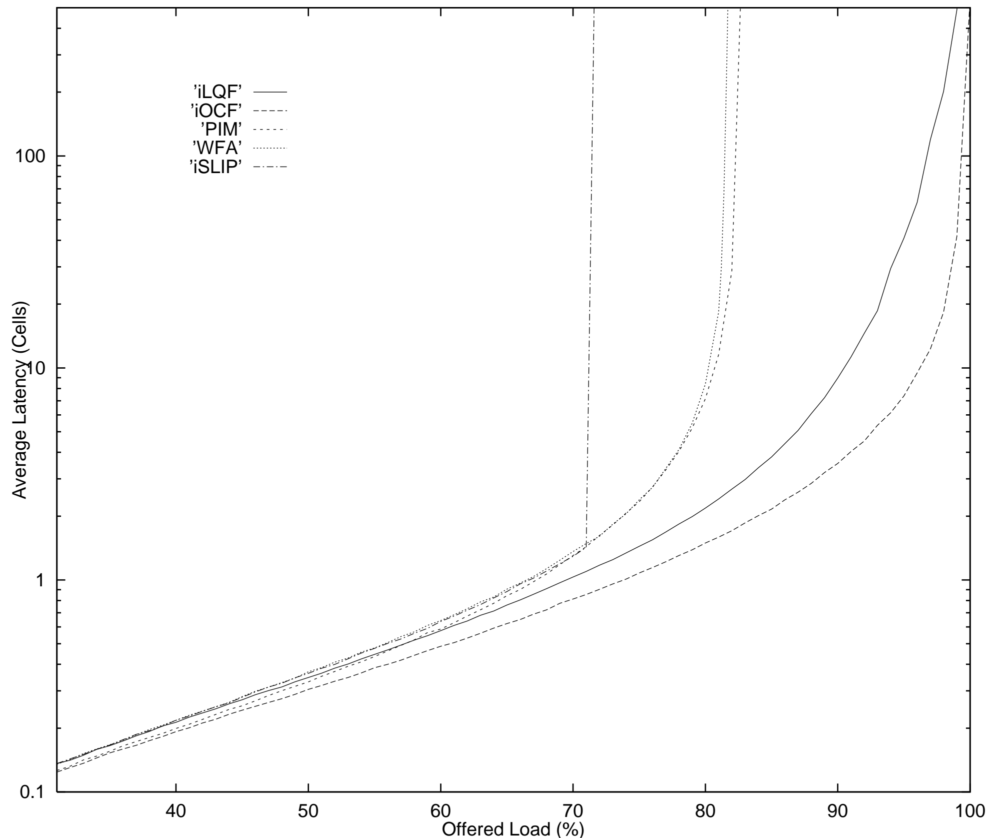


Figure 2.8 Performance comparison of *i*LQF and *i*OCF to PIM, WFA and *i*SLIP. The graph shows simulation results of the average cell latencies as a function of offered load of 3×3 switches under non-uniform traffic described in Chapter 1. Arrivals at each input are Bernoulli i.i.d. The number of iterations is three for *i*LQF, *i*OCF, PIM and *i*SLIP.

non-uniform traffic pattern used in Chapter 1. While *i*SLIP and WFA suffer low throughput as a result of the non-uniformity of the pattern, *i*LQF and *i*OCF achieve 100% throughput. Simulation results for other traffic patterns can be found in [49].

In summary, although too complex and too slow for high speed switches, LQF and OCF demonstrate the importance of considering the congestion in the switch when making scheduling decisions. Their approach (in using queue occupancies or the waiting times of cells to weight requests and selecting a maxweight match to ensure 100% throughput for all non-uniform traffic patterns) provides a basis on which the faster algorithms described in the next two chapters rely to achieve the same throughput.

CHAPTER 3

The LPF Algorithm

1 Introduction

This chapter introduces a new weighted scheduling algorithm called the *Longest Port First* (LPF) algorithm. By definition, LPF is a maxweight algorithm similar to LQF described in the previous chapter. Like LQF, LPF considers the occupancies of VOQs. It reduces congestion in the switch by giving preferential service to backlogged queues based on a specific function of all queue occupancies. Doing so, LPF can achieve 100% throughput for all independent arrival processes as LQF.

More importantly, LPF is much more readily implemented than LQF. Unlike LQF, which requires the use of a maxweight algorithm, LPF can be implemented using a maxsize algorithm. This arises from the following important property of LPF. By a careful definition of request weights, an LPF match is found to be both a maxweight *and* a maxsize match.

In terms of complexity, it is an advantage to use a maxsize algorithm. A maxsize algorithm does not need to compare request weights. The running time of the most efficient

maxweight algorithm known to date is $O(N^3 \log N)$ [67][37] while the running time of the most efficient maxsize algorithm is $O(N^{2.5})$ [22][67]. Furthermore, approximating LPF is relatively straightforward since a maxsize algorithm is already well approximated by a variety of existing algorithms which are fast and simple to implement in hardware [2][50][66]. An iterative algorithm to approximate LPF, called *i*LPF, is discussed in detail in Chapter 5.

Selecting a match that is of both a maxsize and a maxweight also gives LPF another advantage over LQF. As explained in Chapter 2, maximizing the number of cells forwarded depends on two criteria: selecting a maxsize match and maintaining a richly connected request graph. While both LQF and LPF accomplish the second criterion by selecting a maxweight match in an attempt to balance queue occupancies, LQF does not accomplish the first. As we shall see in Section 5, this difference leads LPF to achieve lower average cell latency than LQF.

This chapter begins with a description of LPF in Section 2. Section 3 describes the techniques required to implement LPF using a maxsize algorithm, including the need for a special form of maxsize algorithm. Section 4 presents our maxsize algorithm: a modified Edmonds-Karp, which is specifically developed to work with the techniques described in Section 3.

Sections 5 and 6 discuss the performance of LPF using both a theoretical analysis and simulation results. Section 5, assuming the ideal condition, proves that LPF can achieve 100% throughput for both uniform and non-uniform traffic. Considering that approximation algorithms including *i*LPF can take advantage of pipelined implementation to speed up their scheduling decisions, Section 6 analyzes the effect of a pipeline delay on LPF and proves that the delay has no effect on the throughput.

Lastly, Section 7 discusses the starvation problem of LPF. Like LQF and any maxsize algorithm [13][22], LPF can cause starvation to queues with low traffic.

2 The LPF Algorithm

The *Longest Port First* (LPF) algorithm is a weighted algorithm giving preference based on queue occupancies. Like all maxweight algorithms, LPF weights each request and then finds a maxweight match, i.e., a match that maximizes the total weight, $\sum_{i,j} S_{i,j}(n) w_{i,j}(n)$.

2.1 Request Weighting

LPF gives preference to each flow[1] based on the degree of congestion at its input and at its output. To accomplish this, LPF uses queue occupancies to form what are called *input occupancies and output occupancies.* The input occupancy for input i is defined as the number of cells waiting to leave the input one at a time, i.e., the sum of all queue occupancies at the input, $\sum_{j=1}^{N} L_{i,j}(n)$. Likewise, the output occupancy for output j is defined as the total number of cells at all inputs competing for transmission to the output, i.e., the sum of the occupancies of all queues for the output, $\sum_{i=1}^{N} L_{i,j}(n)$. LPF uses input occupancies and output occupancies to determine the request weights. A request weight, $w_{i,j}(n)$, for a request from input i to output j is defined as follows:

$$w_{i,j}(n) = \begin{cases} R_i(n) + C_j(n), & L_{i,j}(n) > 0 \\ 0, & \text{otherwise,} \end{cases} \tag{3.1}$$

1. A flow is defined as a traffic from one input to one output whose arrival rate is $\lambda_{i,j}$ as defined in Chapter 1.

where $R_i(n) = \sum_{j=1}^{N} L_{i,j}(n)$ and $C_j(n) = \sum_{i=1}^{N} L_{i,j}(n)$, which are the row sum and column sum, respectively, of the occupancy matrix, $L(n)$, defined in Chapter 1.

Definition 1: *An LPF match is any maximum weight match with request weights as defined by Equation 3.1.*

Property 1: *The total weight of an LPF match is equal to the sum of the occupancies of all matched inputs and outputs, i.e,* $\sum_{i,j} S_{i,j}(n) w_{i,j}(n) = \sum_{i \in I} R_i + \sum_{j \in J} C_j$ *, where* I *and* J *are the set of matched inputs and matched outputs respectively.*

Since an LPF match is a maxweight match, it can be directly found by using a maxweight algorithm. As we will see shortly, this is not the most efficient approach.

2.2 An LPF Match: a Maxsize and a Maxweight Match

Theorem 1: *The maximum weight match found by LPF is also a maximum size match.*

Proof: *The proof is in Appendix 3.*

The proof is straightforward and based on Property 1 and the Max-flow min-cut theorem [13][67].

3 Using a Maxsize Algorithm to Find an LPF Match

Although Theorem 1 suggests the possibility of using a maxsize algorithm, the theorem does not imply the reverse, i.e., not every maxsize match is a maxweight match (with weights as defined in Equation 3.1). Figure 3.1 illustrates this fact with a counter-example.

As a result, if more than one maxsize match exists, the LPF algorithm must choose the one that has the largest total weight. This is, however, not what existing maxsize algorithms can do: by definition they do not consider request weights [13]. There may be a number of ways to modify existing maxsize algorithms so that they can find an LPF match; but requiring a maxsize algorithm to be able to consider and compare request weights would not be an efficient approach. Such requirement would add more complexity and perhaps make the maxsize algorithm as complex as a maxweight algorithm.

Instead, we propose a simple technique that we call *input and output presorting*, which allows our maxsize algorithm to find an LPF match without comparing request weights. It gives our maxsize algorithm sufficient knowledge about the weight *relationship* of requests so that it does not need to compare the actual weight values.

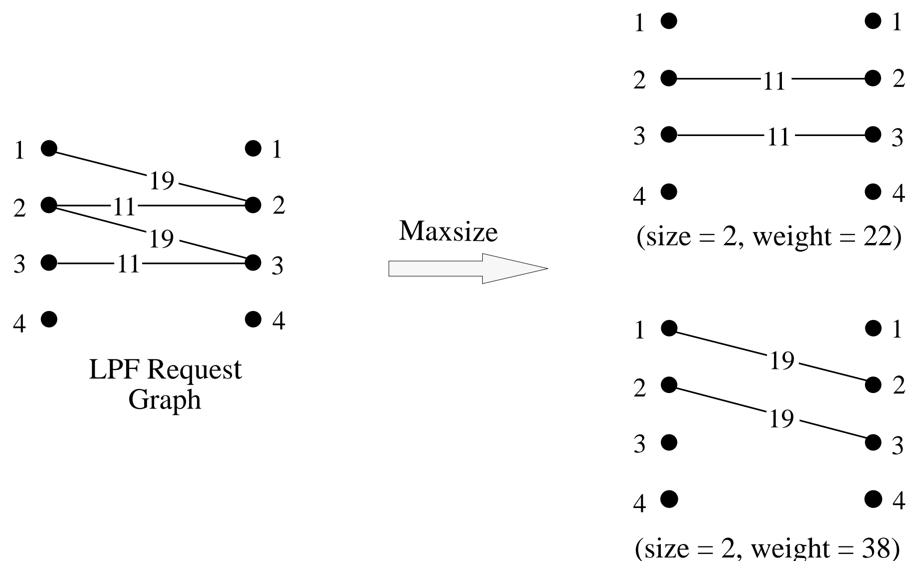


Figure 3.1 An illustration of finding an LPF match from all possible maxsize matches from a given LPF request graph. In this example, there exist two maxsize matches, but only one of them is also a maxweight LPF match.

3.1 Input and Output Presorting

The proposed technique rearranges the requests in the request matrix so that each request is larger than its neighbors below and to the right, i.e., sorts the matrix row-rise and column-wise in a decreasing order. Hence, our maxsize algorithm needs not compare request weight values. Section 4 describes a modified maxsize algorithm that can take advantage of this rearrangement to find an LPF match.

For LPF, such rearrangement is possible because each request weight is the sum of the corresponding input occupancy and output occupancy. Thus, we can observe that an LPF request matrix has the following properties:

Property 2: *For an LPF request matrix, if the output occupancies are in a sorted order in any one row, so are the requests within* ***every*** *row.*

Property 3: *For an LPF request matrix, if the input occupancies are in a sorted order in any one column, so are the requests within* ***every*** *column.*

Property 2 is true for LPF because requests in the same row of the request matrix share the same input occupancies and hence their orders in the row are determined only by their output occupancies. So, if the output occupancies are sorted (for instance, in a decreasing order), then requests within every row follow the same order. A similar argument applies to Property 3.

Because of Properties 2 and 3, sorting the request matrix row-wise (left-to-right) and column-wise (top-to-bottom) becomes possible, and is simply a matter of sorting the output occupancies by permuting the columns and sorting the input occupancies by permuting the rows. Figure 3.2 illustrates these rearrangement steps.

As a result of the top-to-bottom and left-to-right sorting, the request at the top left corner of the matrix has the largest weight while the one at the lower right corner has the

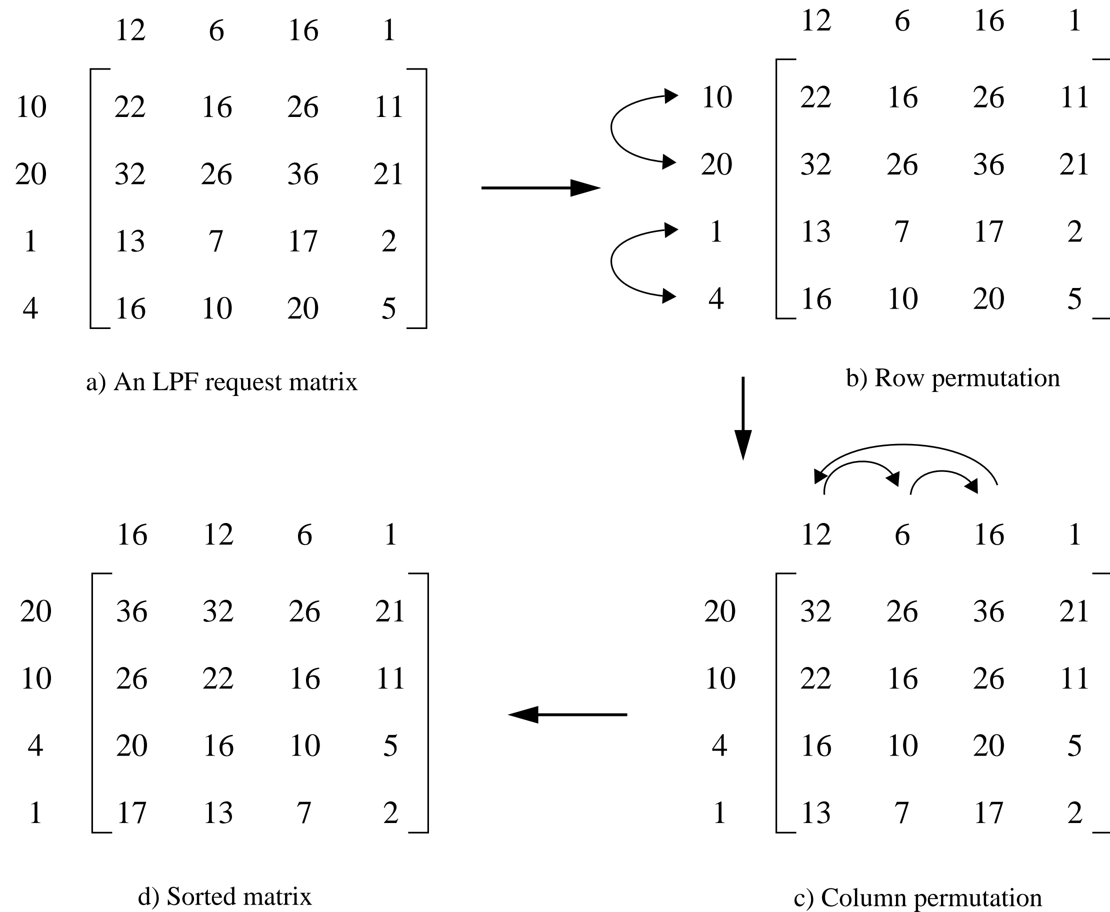


Figure 3.2 An illustration of input and output presorting and request matrix permutation. (a) A request matrix of a 4×4 switch: Each row represents an input, and each column represents an output. The number above each column is its output occupancy, and the number to the left of each row is its input occupancy. Each request weight is a sum of the associated input and output occupancies. (b) Row permutation: Based on the input occupancies, sorting the requests within each column is accomplished by switching row 1 and 2 and by switching row 3 and 4. (c) Column permutation: Based on the output occupancies, sorting the requests within the same row is accomplished by permuting the columns as shown by the arrows.

smallest weight as shown in Figure 3.2 (d). It is worth noting that neither the above row and column permutations or the modified maxsize algorithm require each individual request weight.[1] The request weights in Figure 3.2 are calculated for an illustration purpose only. In the actual implementation, LPF does not calculate the weights.

1. Input and output presorting relies on input and output occupancies, and the modified maxsize algorithm does not compare nor consider request weights.

In addition to eliminating the need for a maxsize algorithm to compare the request weights, the input and output presorting technique also reduces the complexity of magnitude comparison. Instead of comparing all N^2 request weights (as in LQF), the presorting only involves comparing N input and N output occupancies. This reduces the complexity of magnitude comparison from $O(N^2 \log N)$ to $O(N \log N)$.

3.2 Implementation Using a Maxsize Algorithm

Figure 3.3 shows an implementation of LPF using a maxsize algorithm. The implementation consists of two sorters, two crossbar switches and a block implementing a modified maxsize algorithm described in the next section. The inputs to the scheduler are a non-weighted (raw) request matrix and the occupancies of all inputs and all outputs. The output from the scheduler is a matching. The sorters and the crossbars work together to presort the request matrix according to the input and output occupancies. The two sorters perform input and output presorting. The input rank and the output rank are then used to configure the two crossbars to reorder the requests by permuting the request matrix first row-wise then column-wise. The presorted request matrix (with no weights attached) is then passed to the maxsize matching block, which is designed to give preference to requests from left-to-right and top-to-bottom. Because of the request permutation, the output of the maxsize matching block must be permuted back to its original order.

4 Modified Maxsize Algorithm

In order to find a maxsize match that is also a maxweight LPF match, the maxsize algorithm must give appropriate preference to requests when searching for a match. The problem, however, is that existing maxsize algorithms generally do not take preference into consideration when finding a match [13][18][23][67]. Nonetheless, it is possible to modify an existing maxsize algorithms to find an LPF match. Specifically, the search algo-

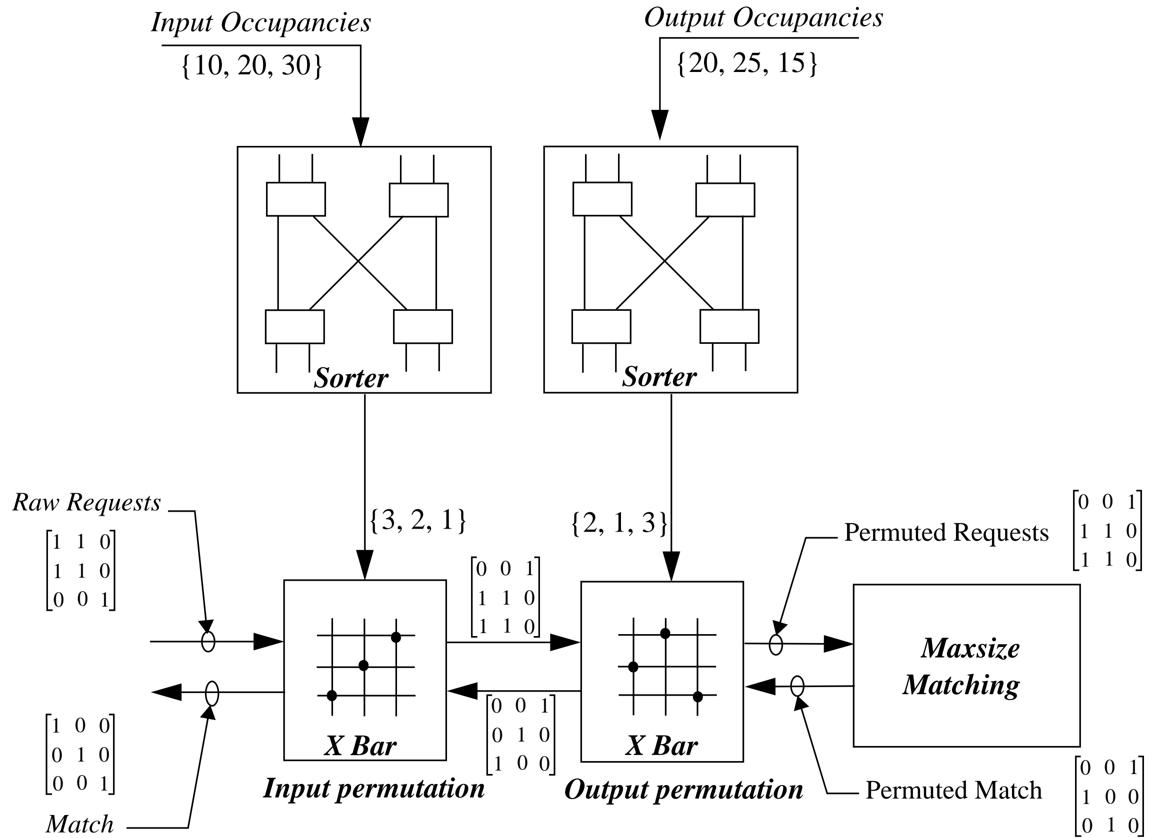


Figure 3.3 A block diagram of LPF. Based on their occupancies, the inputs and outputs are presorted by the two sorters. Raw requests (requests with weights removed) are given in a matrix form. Request reordering is done by the two crossbars, which are configured by the sorting results. Working on permuted requested, the maxsize matching block, which implements a modified maxsize algorithm described in Section 4 finds an LPF match. The match needs to be permuted back to its original order.

rithms may be changed so as to favor a request based on input and output occupancies. As an example, we choose to modify the well-known Edmonds-Karp algorithm [13][18]. We later prove that the modified algorithm finds a maxsize match which is also a maxweight LPF match. Moreover, we show that the modification does not increase the complexity either of the running time or of the arithmetic. The analysis of the running time is described later in the section.

Theorem 2: *For the request weighting defined in Equation 3.1, there exists a maximum size matching algorithm which finds a match that is both maximum size and maximum weight.*

Proof: *Such an algorithm is the modified Edmonds-Karp algorithm defined and proved in Section 4.1.*

4.1 The Modified Edmonds-Karp Algorithm

Because the construction and the analysis of our modified Edmonds-Karp algorithm rely heavily on the concept of a flow network, it is useful to first overview this concept. In principle, a matching problem can be transformed into a network flow problem [13][67], an example of which is shown in Figure 3.4. First, a source s and a sink t are added to the

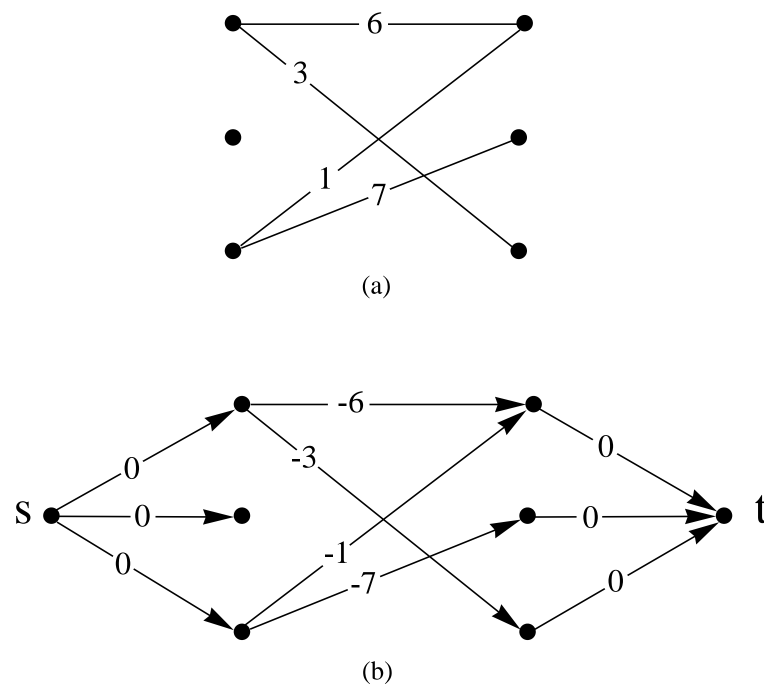


Figure 3.4 A transformation of a request graph into a flow network: (a) A weighted request graph; (b) The corresponding flow network.

request graph. Every input is then connected to s, and every output is connected to t by an edge of zero cost. The costs of all other edges are set to the negated values of the corresponding weights. All edges have a unity flow capacity in a direction as indicated by the arrows. Then, from a flow network, solving a maximum size matching problem is equivalent to finding a maximum flow on the network while a maximum weight match is obtained by finding a flow of a minimum cost [13][37][67].

Before describing our modified Edmonds-Karp algorithm, we first review the original Edmonds-Karp algorithm.

4.1.1 The Original Edmonds-Karp Algorithm

Described in the pseudocode in Figure 3.5 is the Edmonds-Karp algorithm [13][18]. G is a flow network, and $E[G]$ is the set of all edges in G. u, v are vertices in G, which represent an input or an output; (u, v) is an edge from u to v; $c[u, v]$ is the flow capacity of the network from u to v; $f[u, v]$ denotes the current flow from u to v. For a specific flow f, a residual network G_f, also known as a residual graph, consists of edges indicating excess capacities which can admit more flow [13]. The residual capacity of each edge in G_f can be calculated as follows:

$$c_f[u, v] = c[u, v] - f[u, v] . \tag{3.2}$$

Because a flow network has unit capacity and the algorithm considers only an on-off flow, all flows and residual capacities can take only the following set of values: $\{1, -1, 0\}$.

The Edmonds-Karp algorithm finds a maximum flow (a maxsize match) by continuing to increase the flow through the network by performing flow augmentation to find an aug-

Edmonds-Karp algorithm

1 **for** each edge $(u, v) \in E[G]$
2   **do** $f[u, v] = 0$
3         $f[v, u] = 0$
4 **while** BFS finds the shortest path p from s to t in the residual network G_f
5   **for** each edge (u, v) in p
6   **do if** $f[u, v] = 0$ **then** $f[u, v] = c[u, v]$
7   **else** $f[v, u] \leftarrow 0$
8         $f[u, v] = -f[v, u]$

Figure 3.5 The original Edmonds-Karp algorithm [13][18]. G is a flow network or graph constructed as described in Figure 3.4. $E[G]$ is the set of all edges in G; u, v are vertices in G representing an input or an output; (u, v) is an edge from u to v; f is the total flow through the network; $f[u, v]$ denotes a flow from u to v. G_f is a residual network [13][67].

menting path[1] on G_f until no such path exists [13][18]. Using the Max-flow min-cut theorem [13][67], which states that a flow is a maximum flow if and only if the residual network contains no augmenting path, it can be easily verified that a flow found by the Edmonds-Karp algorithm is indeed a maximum flow.

The Edmonds-Karp algorithm uses a breath-first search (BFS) to find the shortest augmenting path. Choosing the shortest path does not necessarily yield a maxweight match because an input may be matched to the worst output, the smallest one, which happens to be the nearest, even though there are other larger unmatched outputs further away. For a similar reason, using depth-first search (DFS) does not lead to a maxweight match.

1. An augmenting path is a path on which an addition flow can be carried from s to t.

4.1.2 The Modified Algorithm

Like the original Edmonds-Karp algorithm, our modified algorithm finds a maxsize match by performing flow augmentation on G_f until no augmenting path exists. However, unlike the Edmonds-Karp algorithm, which uses BFS to find the shortest augmenting path, our modified algorithm uses largest-unmatched-port-first search (LPFS) in finding a different augmenting path to ensure that a match it finds is of a maxweight as well as a maxsize. The following is the pseudocode of our modified algorithm, which only differs from the original Edmonds-Karp algorithm at line 4 where a search for an augmenting path takes place.

Modified Edmonds-Karp algorithm

1 **for** each edge $(u, v) \in E[G]$

2   **do** $f[u, v] = 0$

3         $f[v, u] = 0$

4 **while** LPFS finds a path p from s to t in the residual network G_f

5   **for** each edge (u, v) in p

6   **do if** $f[u, v] = 0$ **then** $f[u, v] = c[u, v]$

7   **else** $f[v, u] \leftarrow 0$

8         $f[u, v] = -f[v, u]$

Figure 3.6 A modified Edmonds-Karp algorithm.

4.1.3 Largest-unmatched-Port-First Search (LPFS)

LPFS differs from BFS and DFS in the following way. BFS and DFS do not consider the sizes of the inputs and the outputs when building a tree from the root to all leaves in the process of augmenting the flow [13]. LPFS, on the other hand, gives preference to the largest undiscovered [13] input and output by exploring them first at every step of expanding the tree. For the purpose of understanding the algorithm, consider a forward tree with s as its root. For LPFS, a search for an augmenting path (from s to t) in a residual graph begins at the largest unmatched input and ends at the largest unmatched output. If no path is found, the search explores another path to the next smaller unmatched output. The search continues until an unmatched output is found or all unmatched outputs are explored. If no unmatched output is found, the input is dropped from a set of to-be-matched inputs since it cannot be matched in any later step, and another search begins from the next smaller input. Searching is terminated after an augmenting path is found or the set of to-be-matched inputs is exhausted.

However, for implementation simplicity, an implementation of LPFS may differ from its conceptual description above. Instead of a forward search from an unmatched input, a reverse search from an unmatched output to find the largest unmatched input may be used. Now a reverse tree with t as its root is constructed. This approach allows LPFS to reuse most parts of the well optimized and analyzed DFS algorithm; only a minor change to the original DFS is required. To maximize the reuse, we construct LPFS as follows.

1. The direction of every edge in a residual graph is reversed to allow a reverse search from t to unmatched inputs.
2. A modified DFS begins from t to find the largest unmatched input, starting with the largest unmatched output and then going through all unmatched outputs in the order of their sizes.

3. The largest unmatched input is dropped from the unmatched set regardless of whether it is matched or not since it cannot be matched in later steps. Any matched output is also dropped from the set.
4. Step 2 is repeated until the unmatched input set is empty or an augmenting path is found.

We modify the original DFS to perform LPFS as follows. As shown in Figure 3.7, Line 2 of DFS-Visit is modified to explore the largest unmatched input or output first, instead of an arbitrary unmatched input or output. The same notations as described in Section 4.1.1 are also used here. Every input or output is initially colored white — undiscovered — then grayed when it is discovered, and finally blackened when it is finished. $\pi[v]$ is the predecessor of v. First, LPFS builds a tree with t as its root. From the tree, if it exists, a path from any unmatched input to t can be found by walking the predecessor list which begins at the input.

4.1.4 Running Time

Since all inputs and outputs have been presorted prior to LPFS, the modification at line 2 of LPFS-Visit in Figure 3.7 does not require extra computation: that is, it requires no magnitude comparison. Hence, the running time of LPFS remains the same as DFS and BFS, which is $O(N^2)$. Therefore, the complexity of the modified algorithm is not different from the original Edmonds-Karp algorithm, which uses BFS. For an $N \times N$ switch, there are at most N path augmentations [13], each with a running time of $O(N^2)$ leading to a total running time of $O(N^3)$. Like the Edmonds-Karp algorithm, the modified algorithm performs only one-bit logical operations: true or false, and so there is no increase in arithmetic complexity.

<u>LPFS(G)</u>

```
1   for each vertex u ∈ V[G]
2         do color[u] ← white
3              π[u] ← nil
4   LPFS-Visit(t)
```

<u>LPFS-Visit(u)</u>

```
1   color[u] ← gray
2   for each v ∈ Adjacent[u] , starting the largest to the smallest.
3         do if color[v] = white
4                then π[v] ← u
5                     LPFS-Visit(v)
6   color[u] = black
```

Figure 3.7 A largest-unmatched-port first search (LPFS). First, LPFS builds a tree with t as its root. Initially every input and output is colored white — undiscovered — then grayed when it is discovered, and finally blackened when it is finished. $\pi[v]$ is the predecessor of v. From the tree, an augmenting path from s to t which must go through an unmatched input can be found by walking the predecessor list which begins at a selected unmatched input.

4.1.5 Equivalency to LPF.

From the Max-flow min-cut theorem, it is given by construction that a match found by the modified Edmonds-Karp algorithm is also a maxsize match. Yet for it to be an LPF match, the match must also be of maximum weight. Therefore, we need to prove that a maxsize match found by the modified Edmonds-Karp is also a maxweight match in order to establish that the algorithm is equivalent to LPF. We prove the theorem below by using a method of a network flow.

Theorem 3: *A match found by the modified Edmonds-Karp algorithm is also a maximum weight match whose weights are as defined in Equation 3.1.*

Proof: *The proof is in Appendix 3.*

5 Performance Analysis of LPF without the Pipeline Delay

This section discusses both theoretical analysis and simulation results of LPF without pipelining in comparison to other existing algorithms.

5.1 Stability Analysis

LPF can achieve 100% throughput for all traffic patterns with independent arrivals. We prove that using the notion of *stability* [38]. We define a switch to be stable for a particular arrival process if the expected length of the input queues does not grow without bound, i.e.,

$$E\left[\sum_{i,j} L_{i,j}(n)\right] < \infty, \forall n\,. \tag{3.3}$$

Definition 2: *A switch can achieve 100% throughput if it is stable for all independent and admissible arrival processes.*

Theorem 4: *Under the LPF algorithm, the queue occupancies are stable for all admissible and independent arrival processes, i.e.,* $E[\|\underline{L}(n)\|] \le C < \infty$.

Proof: *The proof is in Appendix 2.*

5.2 Simulation Results

To complement the above theoretical analysis, we present simulation results to illustrate the performance of LPF. We simulated LPF under uniform and non-uniform traffic.

5.2.1 Uniform Traffic

Shown in Figure 3.8 is an average cell latency comparison of LPF with LQF and a conventional maxsize algorithm [18][23]. The latency of output-queueing is used as a lower bound. LPF is shown by the graph to achieve slightly lower latency than both LQF and a standard maxsize algorithm. This improvement in the latency is possibly due to the fact that LPF selects a match that is of both maxweight and maxsize. By choosing a maxsize match, LPF can immediately maximize the number of queued cells transferred even though it still attempts to balance queue occupancies. LQF, on the other hand, cannot achieve this. As discussed in Chapter 2, it must wait until queue occupancies become balanced before a maxweight match it chooses is also a maxsize match.

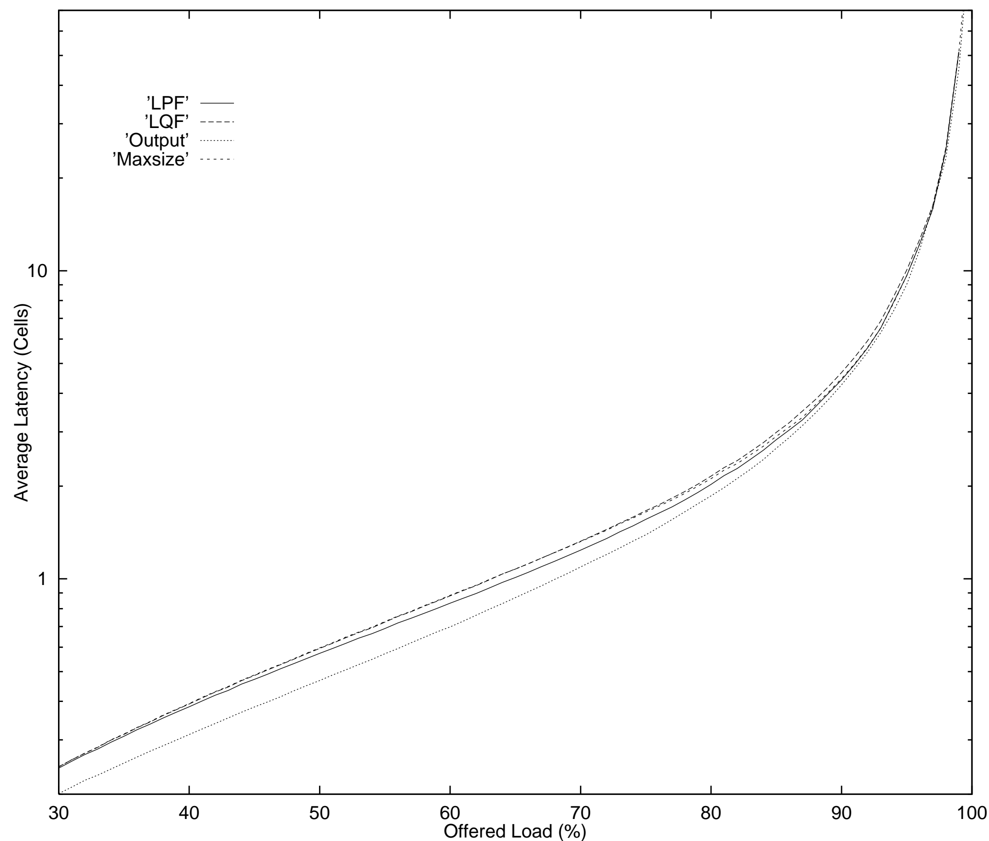


Figure 3.8 Performance comparison of LPF with LQF, a conventional maxsize algorithm and output-queueing. The graph shows the simulation results of the average cell latency as a function of offered load of 16×16 switches under uniform traffic. Arrivals at each input are Bernoulli i.i.d.

5.2.2 Non-uniform Traffic

The performance difference between weighted algorithms and non-weighted algorithm is much clearer under non-uniform traffic. Using the non-uniform traffic patterns shown in Figure 3.9, the simulation result in Figure 3.10 shows that a conventional maxsize algorithm, as predicted in [51], performs poorly. It achieves less than 90% throughput. Under light load, the latency of LQF is not very distinguishable from that of a conventional maxsize algorithm. However, under heavy load, LQF continues to perform well and thus achieves the maximum throughput. Unlike the uniform traffic case, in this case, the simulation result clearly shows that LPF achieves lower latency than both LQF and a conventional maxsize algorithm across the range of offered load. Perhaps surprisingly, LPF achieves almost the same latency as output queueing for this traffic pattern.

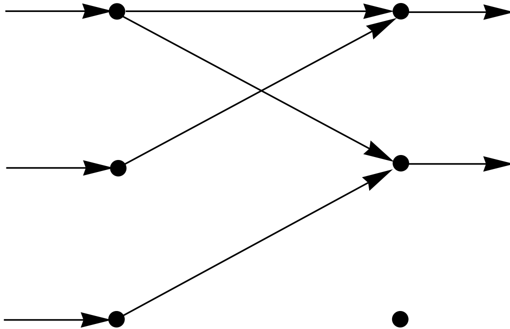


Figure 3.9 A non-uniform traffic pattern for a 3×3 switch, under which a maxsize algorithm is shown to perform poorly [51]. All flows have the same arrival rate.

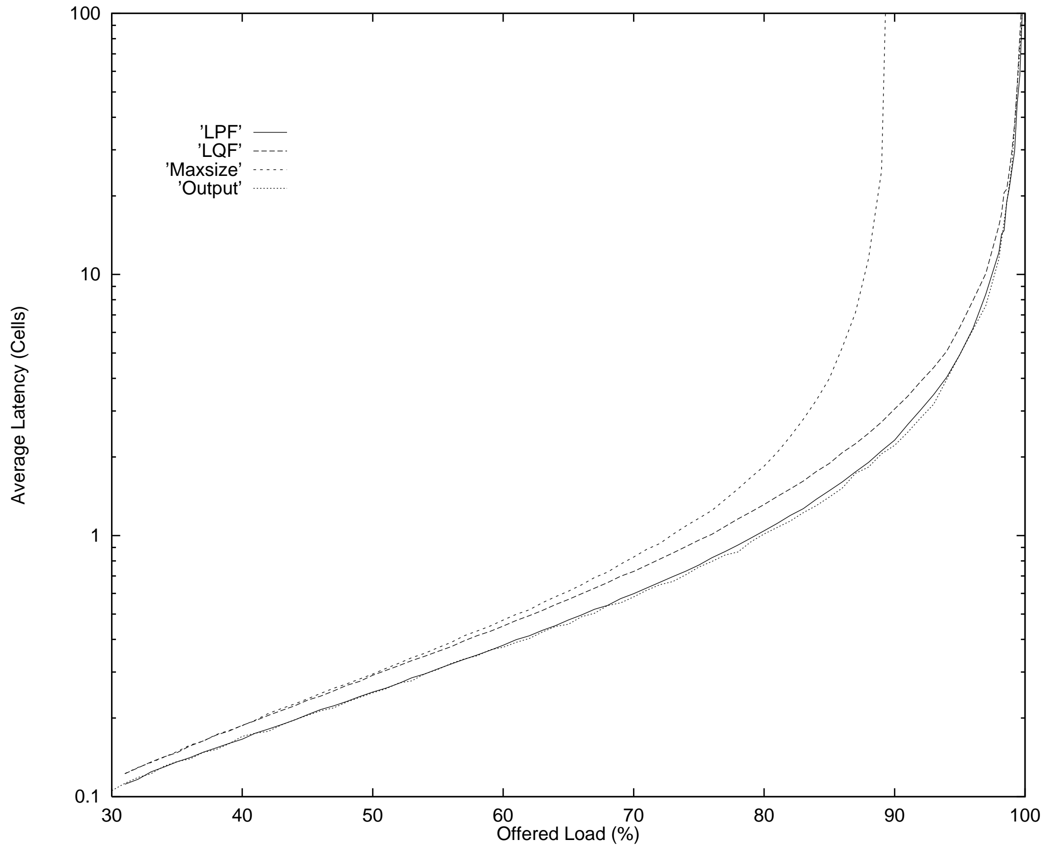


Figure 3.10 Performance comparison of LPF with LQF, a conventional maxsize algorithm and output queueing. The graph shows simulation results of the average cell latency as a function of offered load of 3×3 switches under non-uniform traffic shown in Figure 3.9. Arrivals at each input are Bernoulli i.i.d.

The second non-uniform traffic pattern that we considered is for a 16×16 switch as shown in Figure 3.11. Similar to the traffic pattern in Figure 3.9, this traffic pattern is deliberately constructed to cause a low throughput for a conventional maxsize algorithm. From the pattern, we can observe that input 1 and output 1 are the hot spots with sixteen times more traffic than all other inputs and outputs. Similar to the result for the traffic pattern in Figure 3.9, both LPF and LQF achieve 100% throughput while a conventional

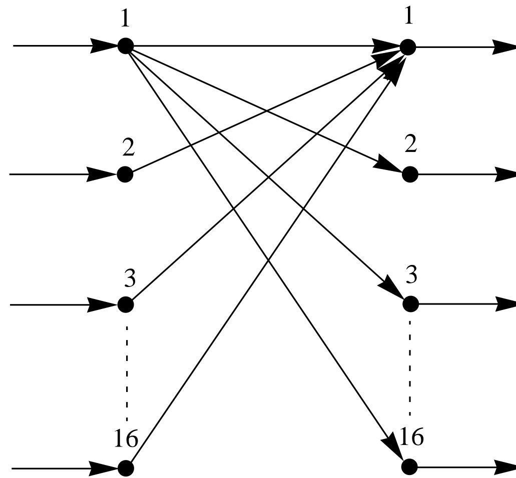


Figure 3.11 A non-uniform traffic pattern for a 16×16 switch. From input 1, there is a flow to every output. From all other inputs, there is a flow to output 1 only. Like the pattern in Figure 3.9, all flows have the same arrival rate.

maxsize algorithm achieves only 94%. In terms of the latencies, LPF achieves lower latency than both LQF and a conventional maxsize algorithm. Like the case of the traffic in Figure 3.9, the latency of LPF is almost identical to that of output queueing.

Although these two traffic patterns do not represent “the worst non-uniform pattern,” their simulation results support the result in the uniform traffic case that selecting a match that is of both a maxsize and maxweight essentially leads LPF to outperform LQF in terms of achieving lower latency.

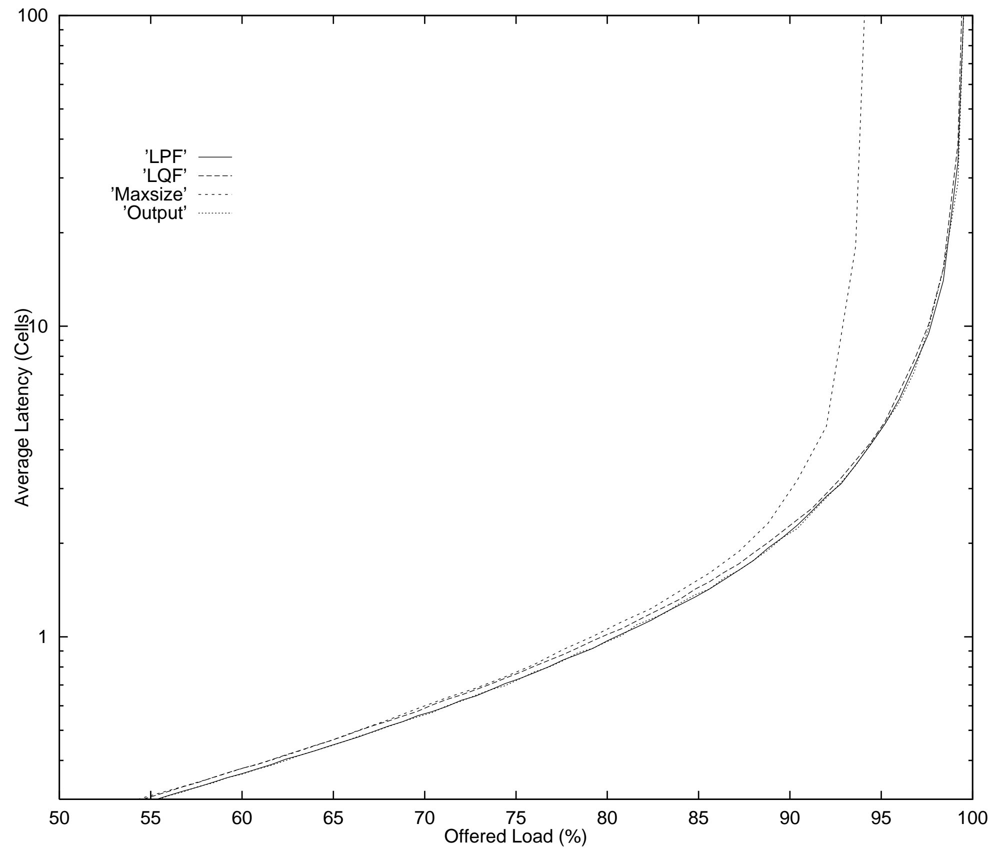


Figure 3.12 Performance comparison of LPF with LQF, a conventional maxsize algorithm and output queueing. The graph shows simulation results of the average cell latency as a function of offered load of 16×16 switches under non-uniform traffic shown in Figure 3.11. Arrivals at each input are Bernoulli i.i.d.

6 Performance Analysis of LPF with the Pipeline Delay

6.1 Improving Scheduling Time by Pipelining

One way to increase the frequency of scheduling decisions is to pipeline the design. For the implementation shown in Figure 3.3, without pipelining, the total scheduling time is the sum of the sorting time, the crossbar configuration time and the maxsize matching time. Exploratory design work suggests that the crossbar configuration time is negligible

compared to the other two.[1] As we will see later, the matching time can also be kept small. The sorting time can be overlapped with the maxsize algorithm using pipelining. Furthermore, the sorters can be broken up into several pipeline stages. As a result of the pipelining, the sorter outputs needed to configure the crossbars are "out of date" by a number of slots equal to the number of the pipeline stages. As a consequence of the delay, the crossbars may be incorrectly configured and thus reorder the requests incorrectly. For instance, using the k slot late outputs, the sorters might mis-configure the crossbars to move row 1 to the bottom row even though row 1 has the largest input occupancy. This can happen because the input occupancy of row 1 may have been the smallest one k slots ago. Incorrect configuration of the crossbars essentially causes the matching block to give the wrong preference to some requests.

Giving incorrect preference can affect performance. As chapter 2 demonstrated, weighted algorithms rely on the ability to give preference to requests based on congestion conditions in order to maintain high throughput under non-uniform traffic. We might expect that the pipeline delay, which causes LPF to give the wrong preference to some requests, may result in low throughput. However, and somewhat surprisingly, we will see from our analysis below that this is not the case.

6.2 Stability Analysis

In this subsection, we consider the effect of the pipeline delay on the performance of LPF. The following analysis includes the stability of LPF with the pipeline delay.

Referring to Figure 3.3, as a consequence of having k pipeline stages in the sorters, the outputs of the sorters are k slots late, causing incorrect preference to be given to some requests. In fact, the effect of the k slot pipeline delay is simply equivalent to attaching k

1. The configuration time of each crossbar is less than 1 ns for a 32×32 crossbar implemented in 0.25 μm CMOS.

slot old weights to current requests and giving them to non-pipelined LPF. Effectively, non-pipelined LPF, therefore, sees k slot old weights, $w_{i,j}(n-k)$, instead of the current weights, $w_{i,j}(n)$. Hence, it finds the match based on the wrong weights, i.e., maximizes $\sum_{i,j} S_{i,j}(n) w_{i,j}(n-k)$. Normally, this can have an adverse effect on the throughput since it impairs the ability of a scheduler to see congestions in the switch. For LPF, however, using out-of-date weights only results in slightly higher average latency not lower throughput. Pipelined LPF still achieves 100% throughput.

Theorem 5: *Using k slot old weights, the LPF algorithm is stable for all admissible independent arrival processes,* $0 < k < \infty$.

Proof: *Theorem is proved in Appendix 2.*

6.3 Simulation Results

Even though the pipeline delay has no effect on the throughput, it degrades the performance by increasing the queueing delay. We study this effect using computer simulation.

6.3.1 Uniform Traffic

Shown in Figure 3.13 is the average latency graph of LPF with different values of the pipeline delay under uniform traffic. Apparently, the increase of queueing delays as a result of the pipeline delay is relatively small and difficult to differentiate from the graph. Figure 3.14 gives a close-up of the graph in Figure 3.13 in a region where the queueing delays differ the most. From the close-up in Figure 3.14, the increase of the average latency graph of LPF with one slot pipeline delay is relatively small. For a pipeline delay of eight slots, there is a noticeable increase in the average latency, but nonetheless the latency is still lower than that of LQF. LPF with thirty-two slot pipeline delay experiences slightly higher latency than the latency of LQF. In summary, this simulation's results sug-

gest no significant adverse effect of the pipeline delay on the performance of LPF under uniform traffic.

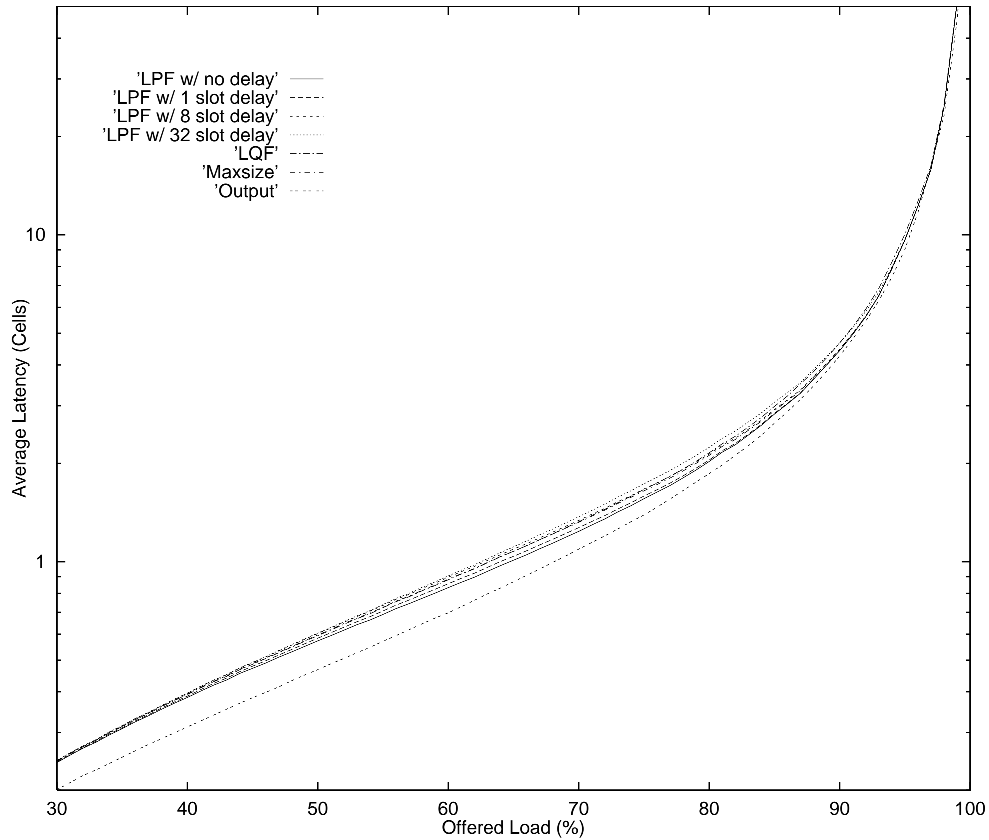


Figure 3.13 Performance comparison of pipelined LPF with various pipeline delays with LQF, a conventional maxsize algorithm and output queueing. The graph shows simulation results of the average cell latency as a function of offered load of 16×16 switches under uniform traffic. Arrivals at each input are Bernoulli i.i.d.

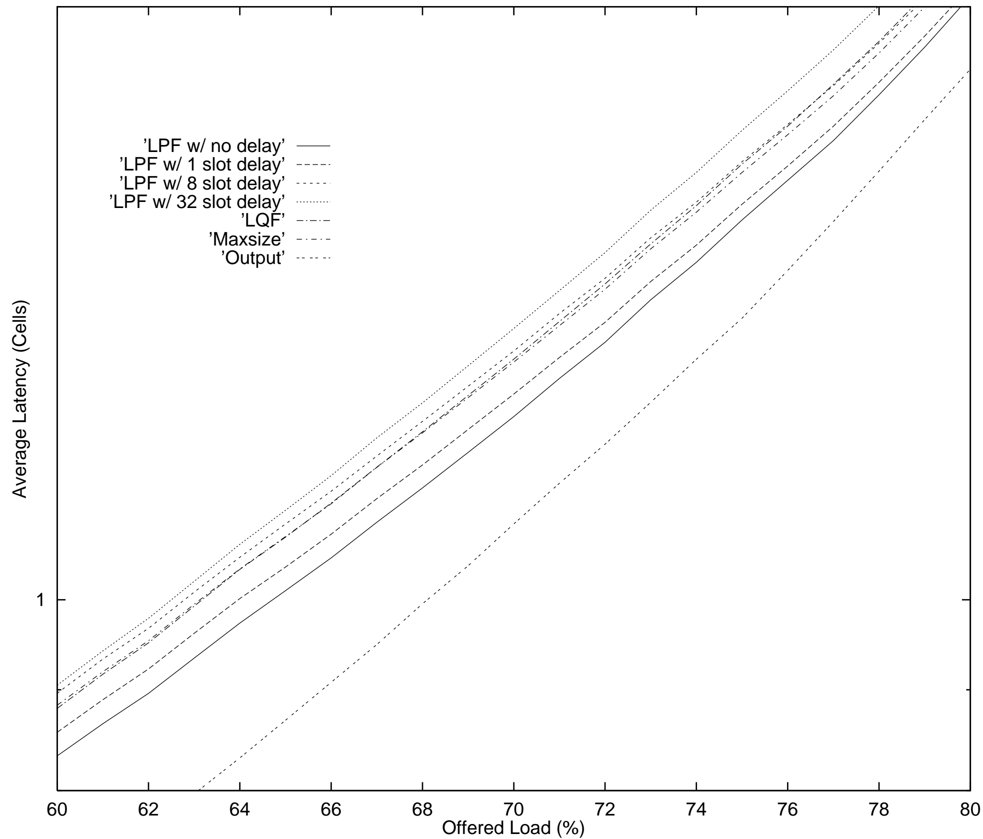


Figure 3.14 A close-up of the graph in Figure 3.13 from 60% to 80% offered load showing a small increase in cell delay as a result of the pipeline delay.

6.3.2 Non-uniform Traffic

The effect of the pipeline delay for non-uniform traffic is greater than for uniform traffic as indicated by the graph in Figure 3.15 (the graph is for the 3×3 traffic pattern in Figure 3.9). This is mainly because, for non-uniform traffic, weighted scheduling algorithms rely on request weights to give proper service to the queues according to current traffic conditions. The degradation in latency is very noticeable across pipeline delay values. But

overall, the increase is not severe; the latency of LPF with thirty two slot delay is lower than that of LQF except between 85% and 95% offered load.

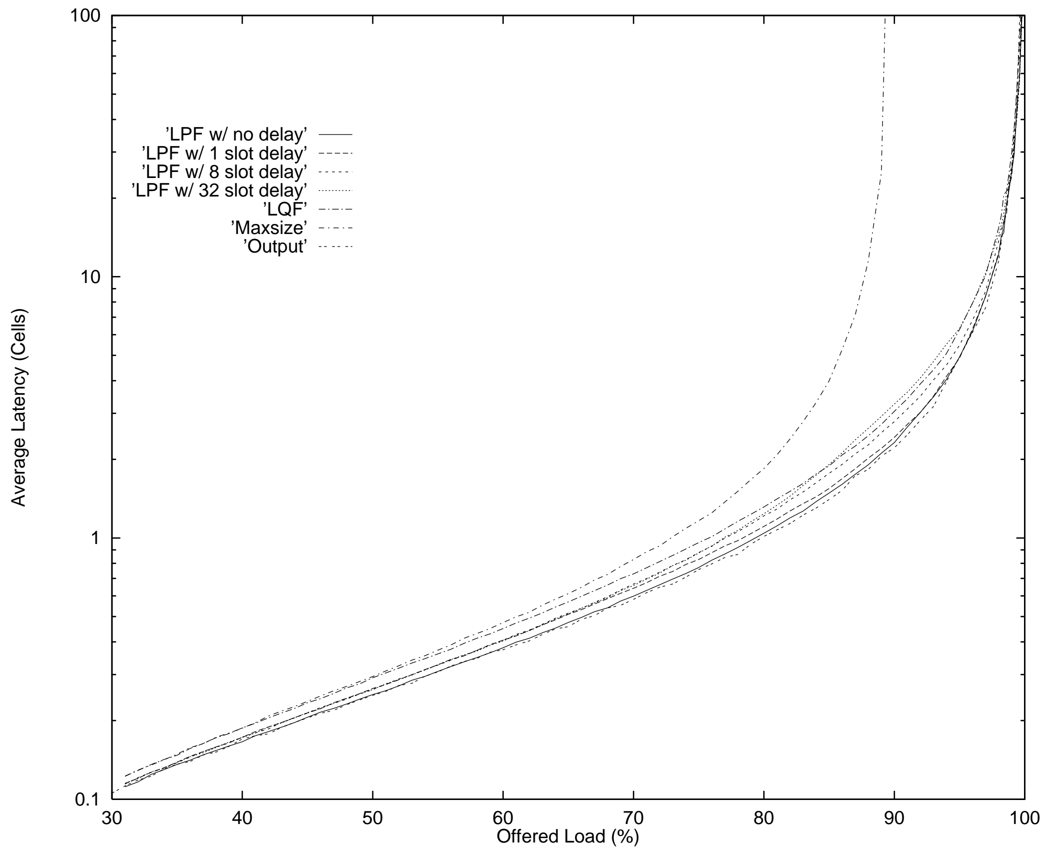


Figure 3.15 Performance comparison of pipelined LPF with various pipeline delays with LQF, a conventional maxsize algorithm and output queueing. The graph shows simulation results of the average cell latency as a function of offered load of 3×3 switches under non-uniform traffic shown in Figure 3.9. Arrivals at each input are Bernoulli i.i.d.

For the traffic pattern in Figure 3.11, however, the effect of the pipeline delay on the latency is difficult to observe. As shown by the graph in Figure 3.16. the latencies of LPF with eight and thirty-two pipeline delay lie between the latencies of LQF and LPF with no pipeline delay, which do not differ much to begin with. Nevertheless, this simulation result

for a 16×16 switch supports the previous result for a smaller switch that the pipeline delay does not severely increase the latency.

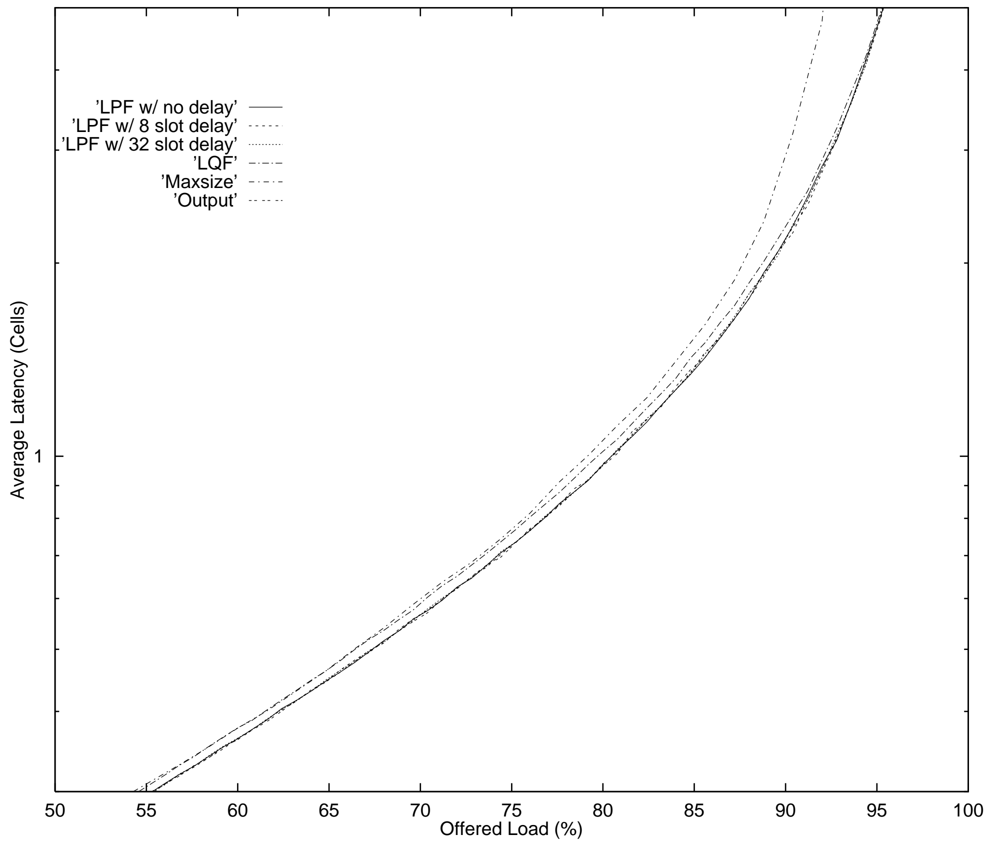


Figure 3.16 Performance comparison of pipelined LPF with various pipeline delays with LQF, a conventional maxsize algorithm and output queueing. The graph shows simulation results of the average cell latency as a function of offered load of 16×16 switches under non-uniform traffic shown in Figure 3.11. Arrivals at each input are Bernoulli i.i.d.

7 Starvation Problem

Like any other algorithm that does not consider the waiting times of cells in the queues, LPF can cause starvation. With LPF, it is possible that a cell remains unserved for

an indefinite time. Since LPF selects a match that is both a maxsize match and a max-weight match, it can cause starvation in the same way as a maxsize algorithm [49] or in a similar way as LQF [49][52]. The following two examples illustrate each of the cases.

Figure 3.17 illustrates a situation in which LPF can cause starvation in a 2×2 switch in the same way as a maxsize algorithm by selecting a maxsize match. Consider the following scenario. At the beginning $Q_{1,1}$ has three cells while $Q_{2,2}$ has none, and the other two queues have one cell waiting in each. Assume that both $Q_{1,1}$ and $Q_{2,2}$ have no arrival

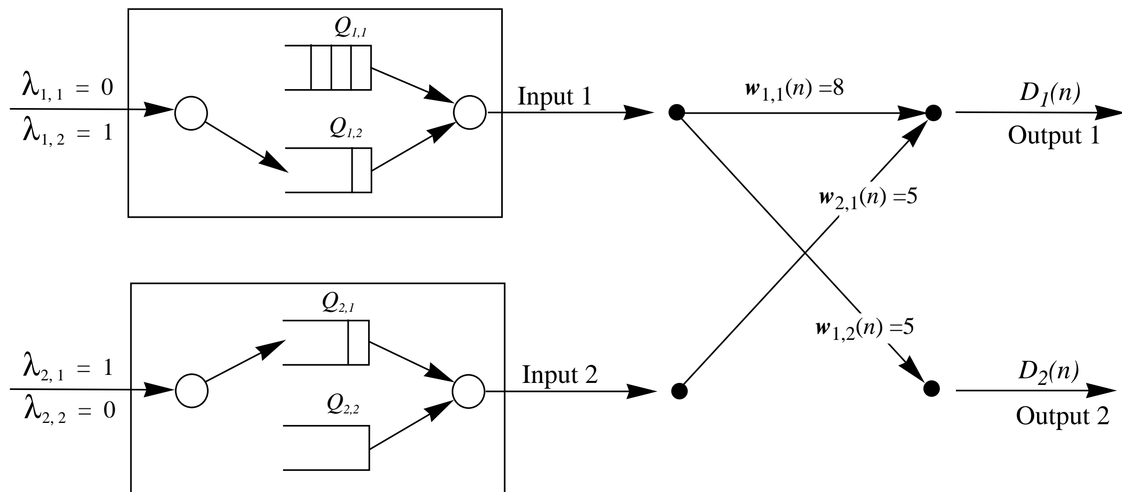


Figure 3.17 An example of a 2×2 switch for which, using the LPF algorithm, an input queue may be starved as a result of LPF selecting a maxsize match. $w_{i,j}(n)$ indicates a request weight defined by Equation 3.1. In this example, $Q_{1,1}$ is starved even though it has three cells waiting while the other two active queues each have only one cell.

in any slot while the other two queues always have arrivals in every slot. As a result, the occupancies of $Q_{1,2}$ and $Q_{2,1}$ will never fall below one. Consequently, LPF always serves $Q_{1,2}$ and $Q_{2,1}$ because it sees a larger size and hence a larger weight match[1] even though there are three cells waiting in $Q_{1,1}$. Since $Q_{1,1}$ is never served, the three cells remain in the queue indefinitely.

Another example illustrates the case in which LPF starves a queue in the same way that LQF does. Consider the example in Figure 3.18. The switch starts with one cell in $Q_{1,1}$ and two cells in $Q_{2,1}$. In every subsequent slot, there is always one arrival at $Q_{2,1}$ but none at any other queues. Hence, there are only two queues that can make requests: $Q_{1,1}$ and $Q_{2,1}$. Consequently, the maximum match size is always one, and LPF can

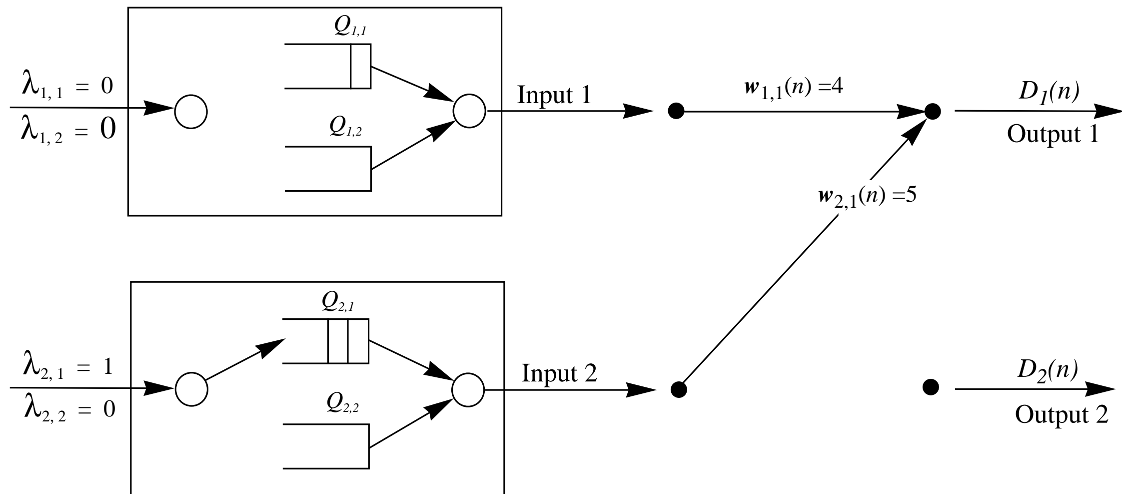


Figure 3.18 An example of a 2×2 switch for which, using the LPF algorithm, an input queue may be starved as a result of LPF selecting a maxweight match. $w_{i,j}(n)$ indicates a request weight defined by Equation 3.1. As a result of the traffic pattern, $Q_{1,1}$ is starved while $Q_{2,1}$ continuously get served.

1. Refer to Property 1 on page 46.

choose to match either $Q_{1,1}$ or $Q_{2,1}$ but not both. As a result of such arrivals, the occupancy of $Q_{1,1}$ will never increase while the occupancy of $Q_{2,1}$ will never decrease. This means that the occupancy of $Q_{1,1}$ is never greater than one and that the occupancy of $Q_{2,1}$ is never less than two. This condition forces LPF to always choose to match $Q_{2,1}$ because it gives a larger weight match, leaving $Q_{1,1}$ unserved indefinitely, i.e., being starved.

Although it does not affect the throughput, starvation is highly undesirable [8][49][52]. Presented in the following chapter is another new algorithm that is designed to eliminate the starvation caused by giving preferential service based on queue occupancies.

CHAPTER 4

The OPF Algorithm

1 Introduction

This chapter introduces another new algorithm called *the Oldest Port First* (OPF) designed to address the starvation problem of LPF. Following a similar approach as OCF, OPF uses the waiting times of cells to form request weights in order to avoid the problem of starvation due to giving preferential service based on queue occupancies. As discussed in Chapter 2, using the waiting times has a desirable property in preventing starvation because all unserved cells age every cell-time, hence increasingly being given a higher preference by the algorithm and becoming more likely to be served as a result. Besides the use of the waiting times instead of queues occupancies, OPF weights requests in the same way that LPF does: it uses the same function to derive request weights from the waiting times. As a result, OPF retains all the properties and the advantages of LPF including the implementation.

This chapter begins with the introduction of OPF. Section 2 describes an OPF request weighting scheme and discusses its rationale. Section 3 outlines implementing OPF using a maxsize algorithm.

As in the case of LPF, OPF also benefits greatly from pipelined implementation to speed up its scheduling decisions. Accordingly, we analyze OPF's performance under both conditions: with and without the pipeline delay. Section 4 considers the performance of OPF without the pipeline delay, showing that OPF can achieve 100% throughput for both uniform and non-uniform traffic.

Section 5 considers the impact of the pipeline delay on OPF's performance. First, we prove that, in the presence of a finite pipeline delay, OPF still achieves 100% throughput. Then, using simulation results, we show that the delay has an insignificant impact on the performance of OPF in terms of cell delay.

2 The OPF Algorithm

Although eventually OPF will be proved to be both a maxsize and a maxweight algorithm like LPF, for an analytical purpose, it is simpler to start by considering OPF as a maxweight algorithm.[1] By definition, OPF is a maxweight algorithm, giving preference based on the waiting times of cells instead of queue occupancies. Each request weight is a function of the waiting times of HOL cells. Like LPF, OPF does not use an individual waiting time to weight a request, but instead uses what we call *an input waiting time* and *an output waiting time*. The waiting time of input i, R_i, is defined as the waiting time sum of all HOL cells at the input, and the waiting time of output j, C_j, is defined as the waiting time sum of all HOL cells destined for the output. In the same fashion as LPF, an input waiting time serves as a congestion indicator of cells competing for the outgoing link of the input, and an output waiting time serves as a congestion indicator of the incoming link of the output. Accordingly, a request weight, $w'_{i,j}(n)$, of every non-empty queue consists of two quantities as defined below:

1. The proof in Appendix 4 relies on an OPF match to be a maxweight match.

$$w'_{i,j}(n) = \begin{cases} R_i + C_j, & L_{i,j}(n) > 0 \\ 0, & \text{otherwise,} \end{cases} \tag{4.1}$$

where $R_i = \sum_{j=1}^{N} W_{i,j}(n)$ and $C_j = \sum_{i=1}^{N} W_{i,j}(n)$, which are the row sum and column sum of the waiting-time matrix, $W(n)$, defined in Chapter 1.

Definition 1: *An OPF match is any maximum weight match with request weights as defined by Equation 4.1.*

Property 1: *The total weight of an OPF match is equal to the waiting time sum of all matched inputs and outputs, i.e,* $\sum_{i,j} S_{i,j}(n) w_{i,j}(n) = \sum_{i \in I} R_i + \sum_{j \in J} C_j$ *, where* I *and* J *are the set of matched inputs and matched outputs respectively.*

3 Using a Maxsize Algorithm to Find an OPF Match

As in the case of LPF, using a maxweight algorithm to implement OPF would not be of interest because of its high complexity [49][54]. With the request weights depending on the input and output waiting times, an OPF match is found to be both a maxweight and a maxsize match. Thus, to avoid the complexity problem, an efficient way to implement OPF is to use a maxsize algorithm as done for LPF.

Theorem 1: *The maximum weight match found by OPF is also a maximum size match.*

Proof: *The proof is the same as the proof of Theorem 1 in Chapter 3.*

A maxsize algorithm can be adapted to find an OPF match using the same techniques used for LPF. Shown in Figure 4.1 is a block diagram of OPF, which is nearly identical to the block diagram of LPF shown in Chapter 3, except the inputs to the two sorters are now

input and output waiting times instead of input and output occupancies. OPF still benefits from input and output presorting in the same way as LPF. There is only a minor change in the way the input and output presorting and request reordering is done. Instead of being sorted based on their occupancies, the inputs and the outputs are now sorted based on their waiting times. The maxsize algorithm is also the same one used for LPF. For example, it could use the modified Edmonds-Karp algorithm described in Chapter 3. OPF can also be

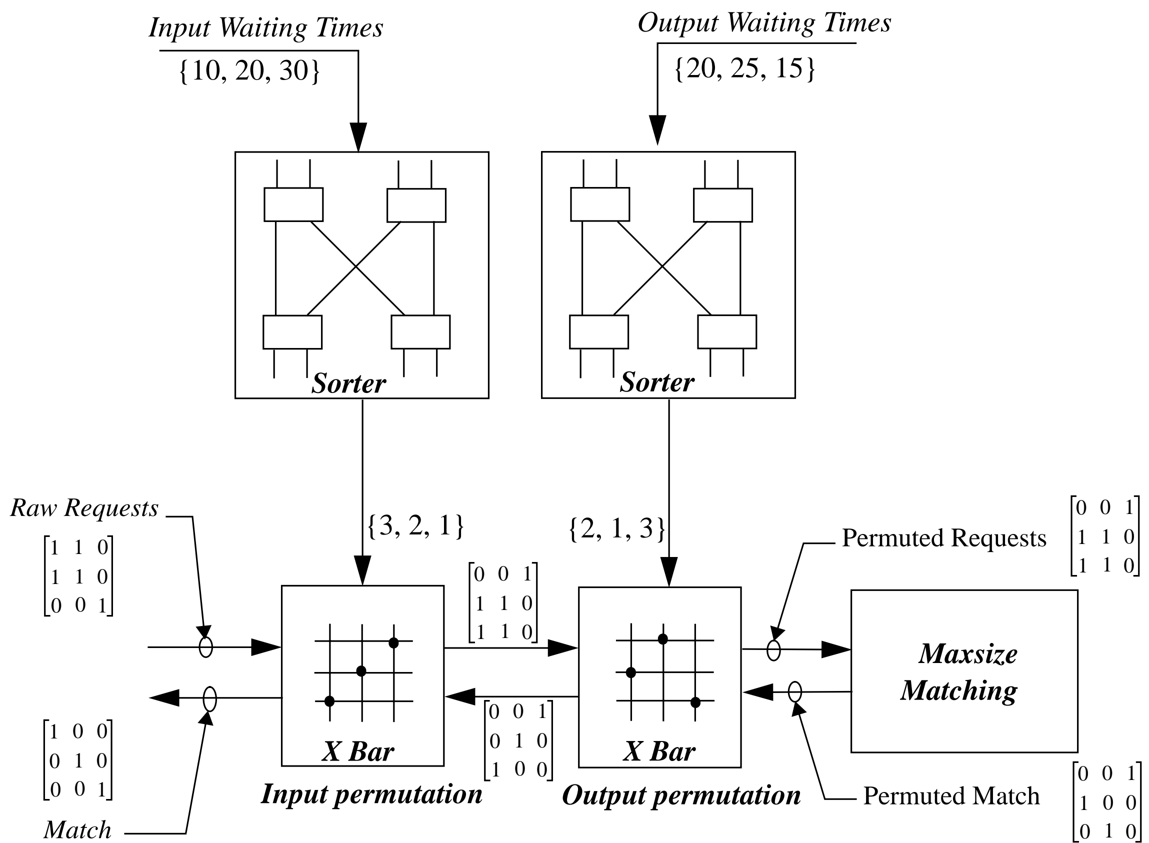


Figure 4.1 A block diagram of OPF. Similar to LPF, the inputs and the outputs are ranked based on their waiting times by the two sorters. Raw requests (requests with weights removed) are given in a matrix form. Request reordering is done by the two crossbars which are configured by the sorting results. The maxsize matching block, which implements the modified algorithm described in Chapter 3, finds a maxsize match that is also a maxweight match. Before being sent out, the match needs to be permuted back to its original order.

pipelined in the same way as LPF. Overall, there is no significant difference from LPF in the way OPF is implemented using a maxsize algorithm.

4 Performance Analysis of OPF without the Pipeline Delay

4.1 Stability Analysis

As designed, OPF performs well under non-uniform traffic. Similar to the service policies of LQF, OCF and LPF, OPF gives preferential service to backlogged queues, resulting in queue occupancies remain relative well-balanced. As outlined in Chapter 2, having queue occupancies balanced is a desirable property because it allows a scheduling algorithm to achieve high throughput for non-uniform traffic.

Like LPF, we use the notion of *stability* [38] to prove that OPF can achieve 100% throughput for all traffic patterns with independent arrivals. The stability criteria and the definition of achieving 100% throughput are described in Chapter 3.

Theorem 2: *Under the OPF algorithm, the queue occupancies are stable for all admissible and independent arrival processes, i.e.,* $E[\|\underline{L}(n)\|] \le C < \infty$.

Proof: *The proof is in Appendix 4.*

4.2 Simulation Results

In addition to the above theoretical analysis, we also use simulation results to gain more understanding on OPF's performance in terms of queueing delay. We simulated OPF under both traffic conditions: uniform and non-uniform. For non-uniform traffic, we choose the same traffic patterns used in Chapter 3, under which a maxsize algorithm performs poorly [51].

4.2.1 Uniform Traffic

The latency graph in Figure 4.2 shows that OPF performs as well as LPF under uniform traffic: the average latency of OPF under uniform traffic is almost indistinguishable from that of LPF. Both achieve slightly lower latency than a conventional maxsize algorithm. However, the performance difference between OPF and OCF is substantial: the average latency of OPF is much lower than that of OCF across the range of offered load. The difference can be attributed to the fact that an OPF match is both a maxweight and maxsize match while an OCF match is only a maxweight match. In this case where a con-

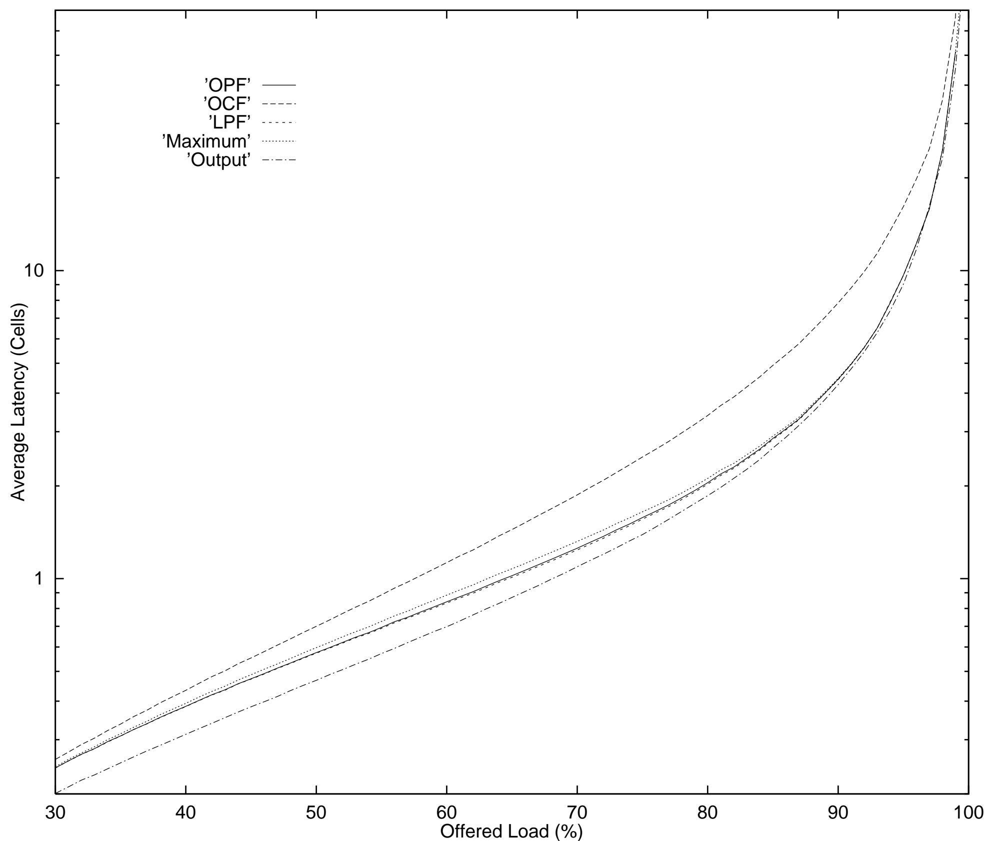


Figure 4.2 Performance comparison of non-pipelined OPF with LPF, OCF, a conventional maxsize algorithm and output queueing. The graph shows simulation results of the average cell latency as a function of offered load of 16×16 switches under uniform traffic. Arrivals at each input are Bernoulli i.i.d.

ventional maxsize algorithm performs much better than OCF, this outcome supports our early observation that there is an added benefit in selecting a maxweight match that is also a maxsize match.

As for the variance of cell latency, the simulation results in Figure 4.3 show that the variance is lower for OPF than LPF, an outcome similar to comparison of LQF and OCF [49]. This is as a result of OPF giving preference to cells based on their waiting times.

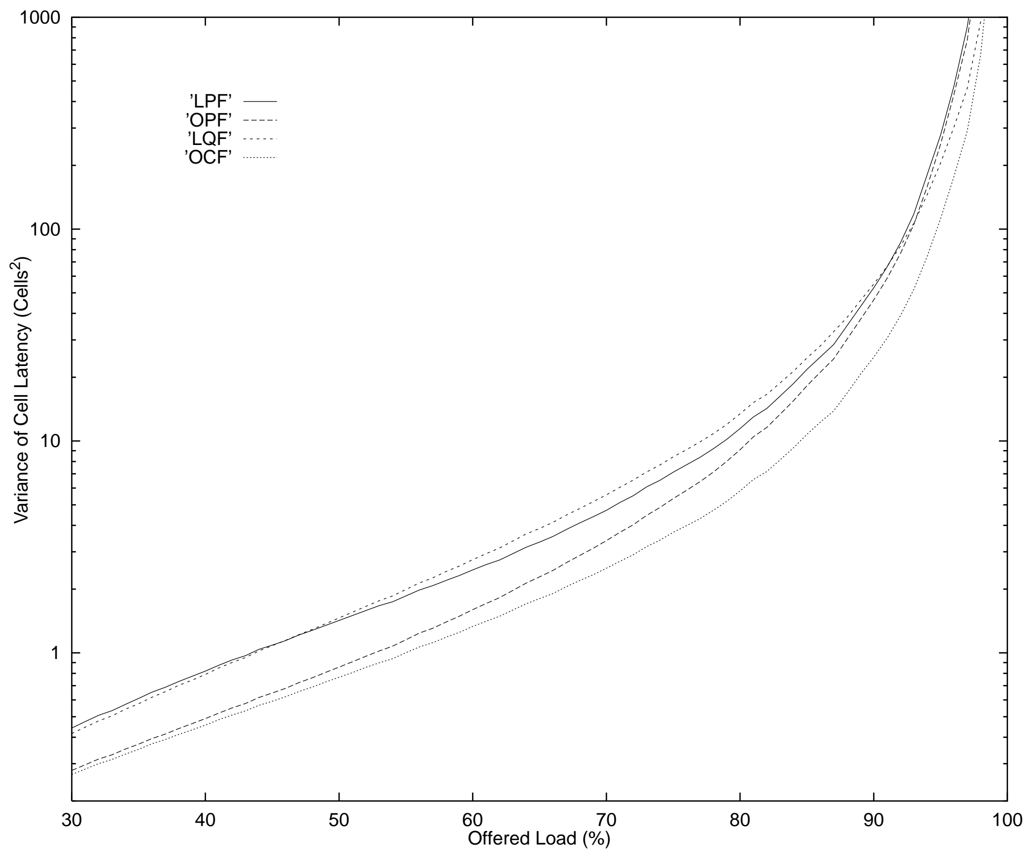


Figure 4.3 Performance comparison of non-pipelined OPF with LPF, LQF and OCF. The graph shows simulation results of the variance of cell latency as a function of offered load of 16×16 switches under uniform traffic. Arrivals at each input are Bernoulli i.i.d.

4.2.2 Non-uniform Traffic

As in the case of LPF, OPF, by far outperforms a conventional maxsize algorithm for the 3×3 traffic pattern in Figure 3.9. As shown by the graph in Figure 4.4, a conventional maxsize algorithm [18][23] achieves less than 90% throughput for this traffic pattern while OPF achieves 100% throughput. However, similar to an OCF vs. LQF comparison [49], OPF incurs slightly higher latency than LPF. Nonetheless, for this traffic pattern, OPF still achieves lower latency than OCF, just as it does in the uniform traffic case.

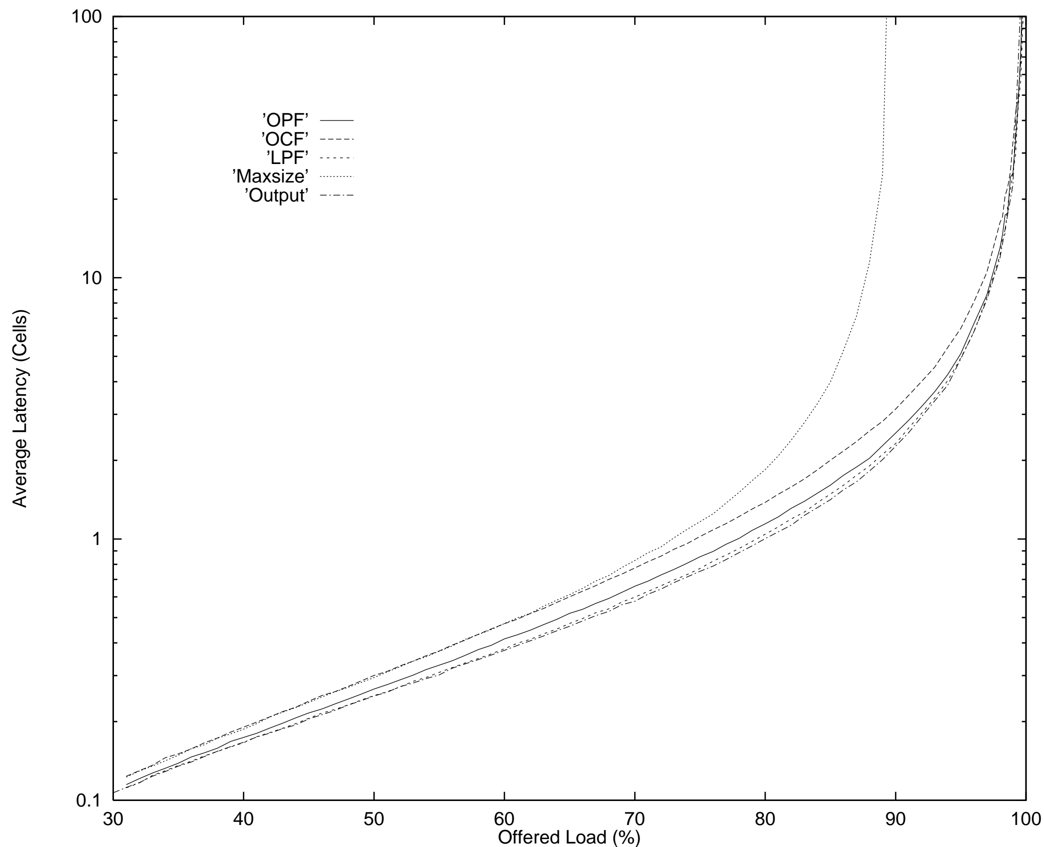


Figure 4.4 Performance comparison of non-pipelined OPF with LPF, OCF, a conventional maxsize algorithm and output queueing. The graph shows simulation results of the average cell latency as a function of offered load of 3×3 switches under the non-uniform traffic pattern shown Figure 3.9. Arrivals at each input are Bernoulli i.i.d.

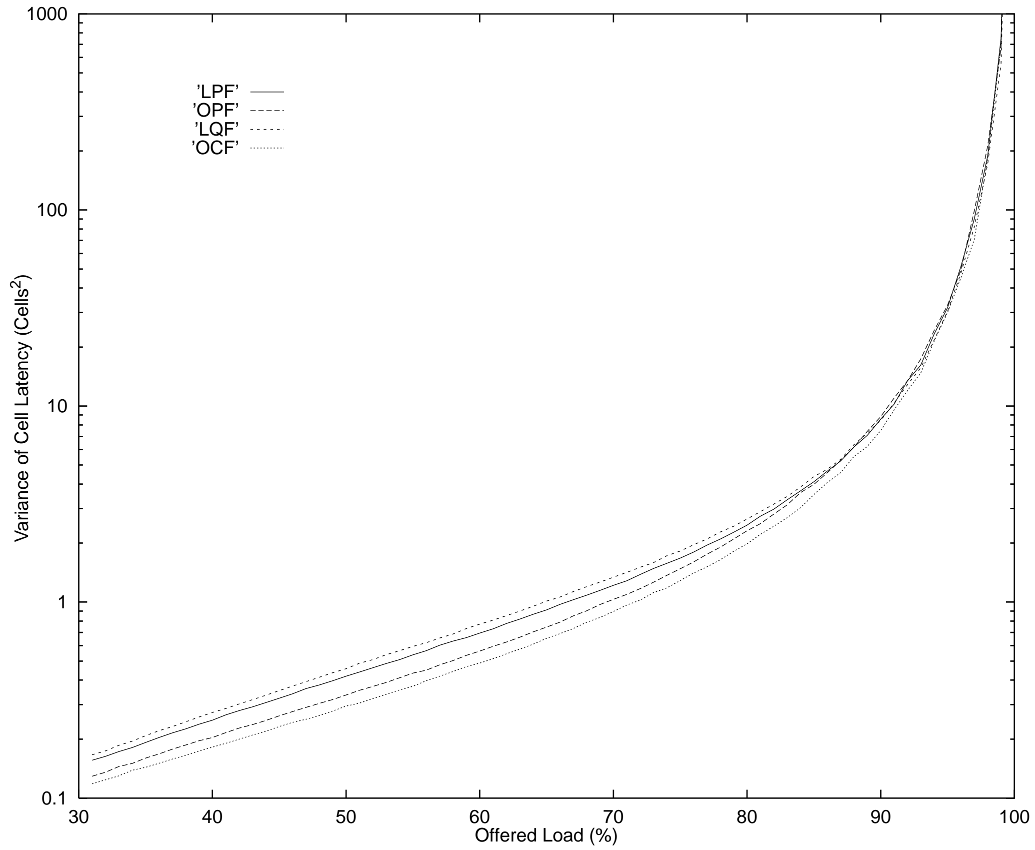


Figure 4.5 Performance comparison of non-pipelined OPF with LPF, LQF and OCF. The graph shows simulation results of the variance of cell latency as a function of offered load of 3×3 switches under the non-uniform traffic pattern shown Figure 3.9. Arrivals at each input are Bernoulli i.i.d.

For the 16×16 traffic pattern in Figure 3.11, OPF achieves lower latency than OCF but slightly higher than LPF as shown by the graph in Figure 4.6.

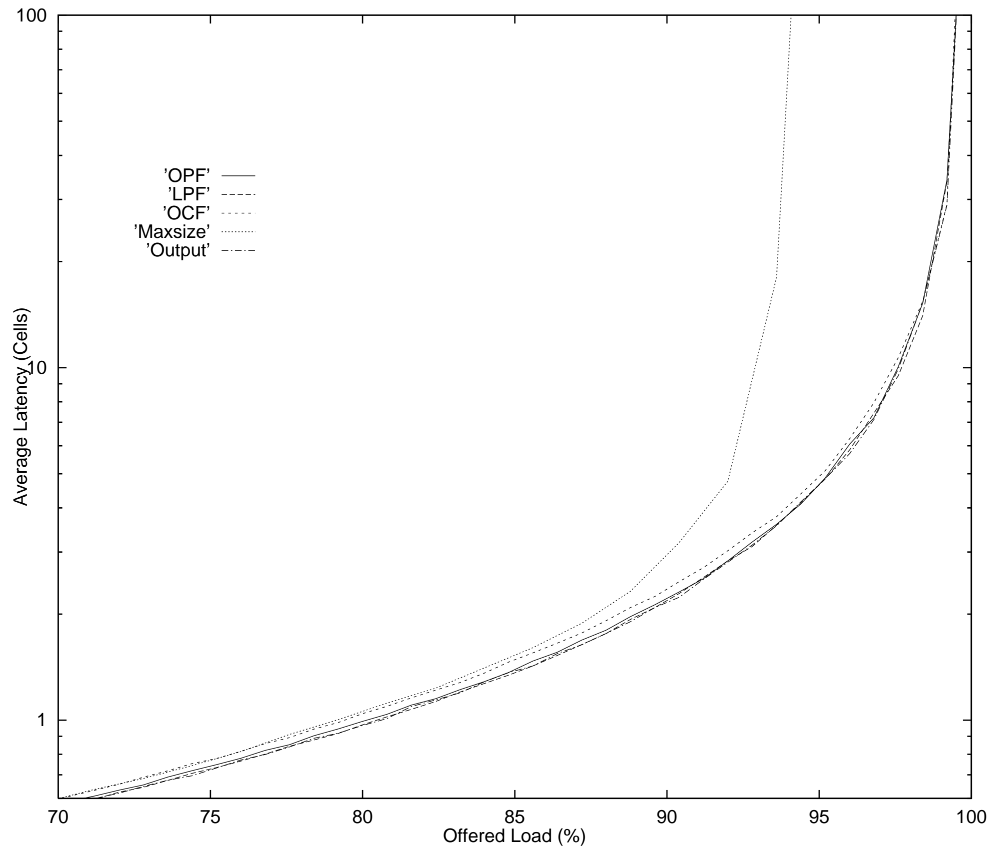


Figure 4.6 Performance comparison of non-pipelined OPF with LPF, OCF, a conventional maxsize algorithm and output queueing. The graph shows simulation results of the average cell latency as a function of offered load of 16×16 switches under the non-uniform traffic shown Figure 3.11. Arrivals at each input are Bernoulli i.i.d.

5 Performance Analysis of OPF with the Pipeline Delay

5.1 Stability Analysis

OPF can also benefit greatly from pipelined implementation to speed up its scheduling decisions. As proved below, OPF, like LPF, can tolerate a finite pipeline delay with no throughput loss.

Exactly the same as LPF, the effect of the k slot pipeline delay[1] is equivalent to attaching k slot old weights, $w_{i,j}(n-k)$, to current requests before passing them to non-pipelined OPF. Non-pipelined OPF, hence, faultily selects a match that maximizes $\sum_{i,j} S_{i,j}(n) w_{i,j}(n-k)$ rather than the optimum one that maximizes $\sum_{i,j} S_{i,j}(n) w_{i,j}(n)$. It is also true for OPF that using out-of-date weights does not lower the throughput. Pipelined OPF still achieves 100% throughput.

Theorem 3: *Using k slot old weights, the OPF algorithm is still stable for all admissible independent arrival processes,* $0 < k < \infty$.

Proof: *Theorem is proved in Appendix 4.*

5.2 Simulation Results

We use simulation to examine the impact of the pipeline delay on the performance in terms of cell delay. We evaluate the effect of the pipeline delay for both uniform and non-uniform traffic using the same patterns as in the non-pipelined case.

5.2.1 Uniform Traffic

Figure 4.7 shows the performance comparison of OPF with various pipeline delay values ranging from zero to thirty two slots. Remarkably, OPF with a pipeline delay of thirty two slots still by far outperforms OCF: the latency of OPF with thirty two slot delay is much lower than the latency of OCF. Overall, for uniform traffic, the pipeline delay has a minor impact on the average latency across the range of the offered load. An explanation for the small impact has to do with the fact that the queues statistically experience the same level of congestion when traffic is uniform and hence are more likely to have the same request weight. When all requests have the same weight, it does not matter which

1. k is the number of pipeline stages of the sorters.

maxsize match OPF (with or without the pipeline delay) selects: all maxsize matches give the same total weight, the maximum weight.

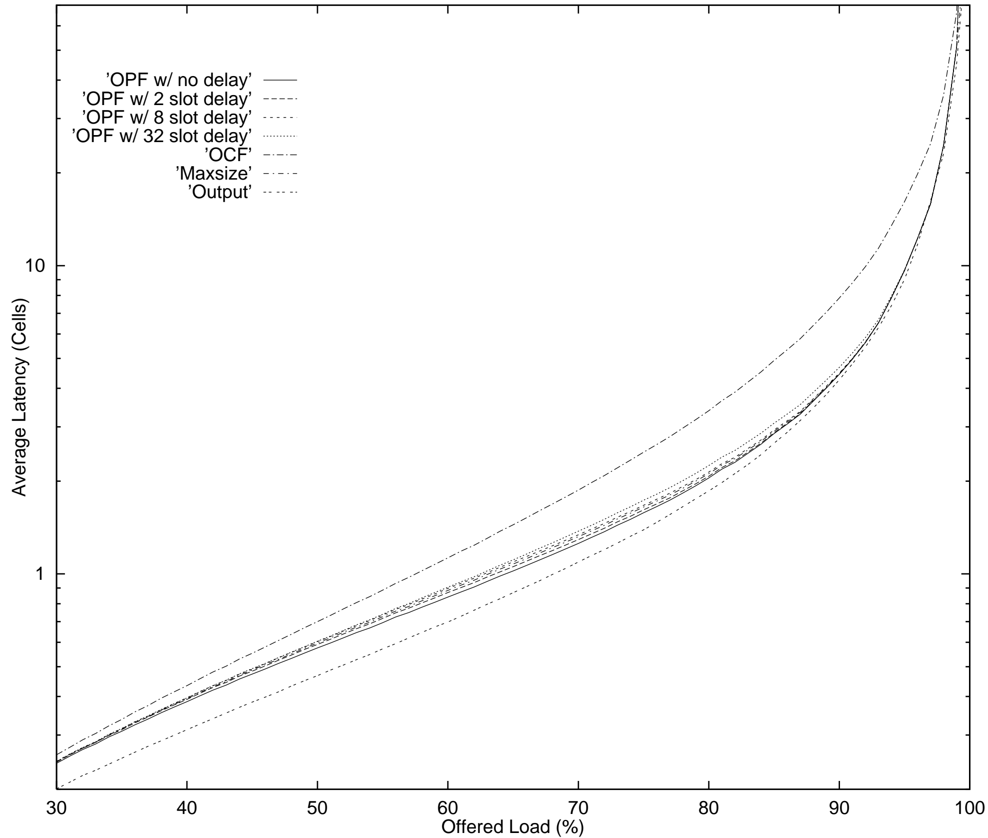


Figure 4.7 Performance comparison of pipelined OPF with various pipeline delays with OCF, a conventional maxsize algorithm and output queueing. The graph shows simulation results of the average cell latency as a function of offered load of 16×16 switches under uniform traffic. Arrivals at each input are Bernoulli i.i.d. The graph indicates that OPF queueing delay gradually increases as a function of the number of pipeline stages.

5.2.2 Non-uniform Traffic

The impact of the pipeline delay is more noticeable for non-uniform traffic than for uniform traffic. For the 3×3 traffic pattern in Figure 3.9, pipelined OPF exhibits a significant increase in the average latency under a heavy load as shown in Figure 4.8. In all, the

impact is not severe: the latency of OPF with thirty two slot delay is not much worst than that of OCF.

For the 16×16 traffic pattern in Figure 3.11, the increase in latency due to the pipeline delay is difficult to observe as in the case of LPF. From the graph in Figure 4.9, OPF with thirty two slot delay experiences noticeably higher latency than non-pipelined OPF but lower latency than OCF from 70% to 90% offered load.

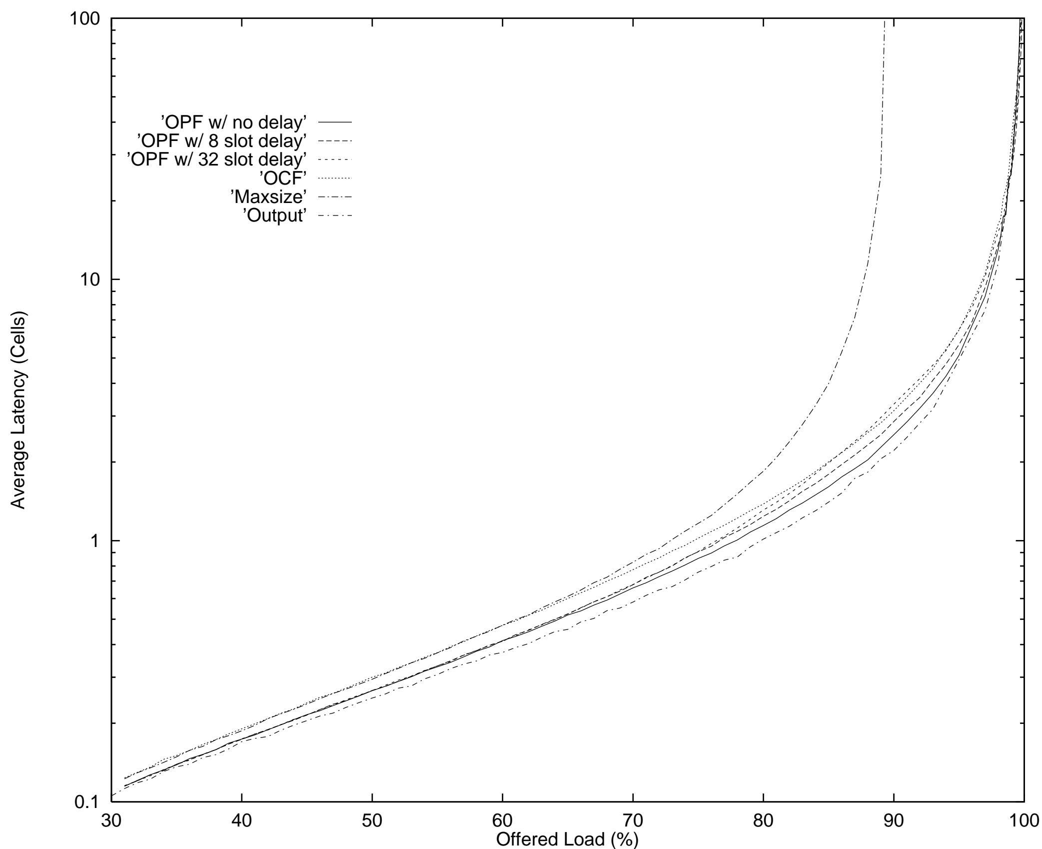


Figure 4.8 Performance comparison of pipelined OPF with various pipeline delays with OCF, a conventional maxsize algorithm and output queueing. The graph shows simulation results of the average cell latency as a function of offered load of 3×3 switches under non-uniform traffic. Arrivals at each input are Bernoulli i.i.d.

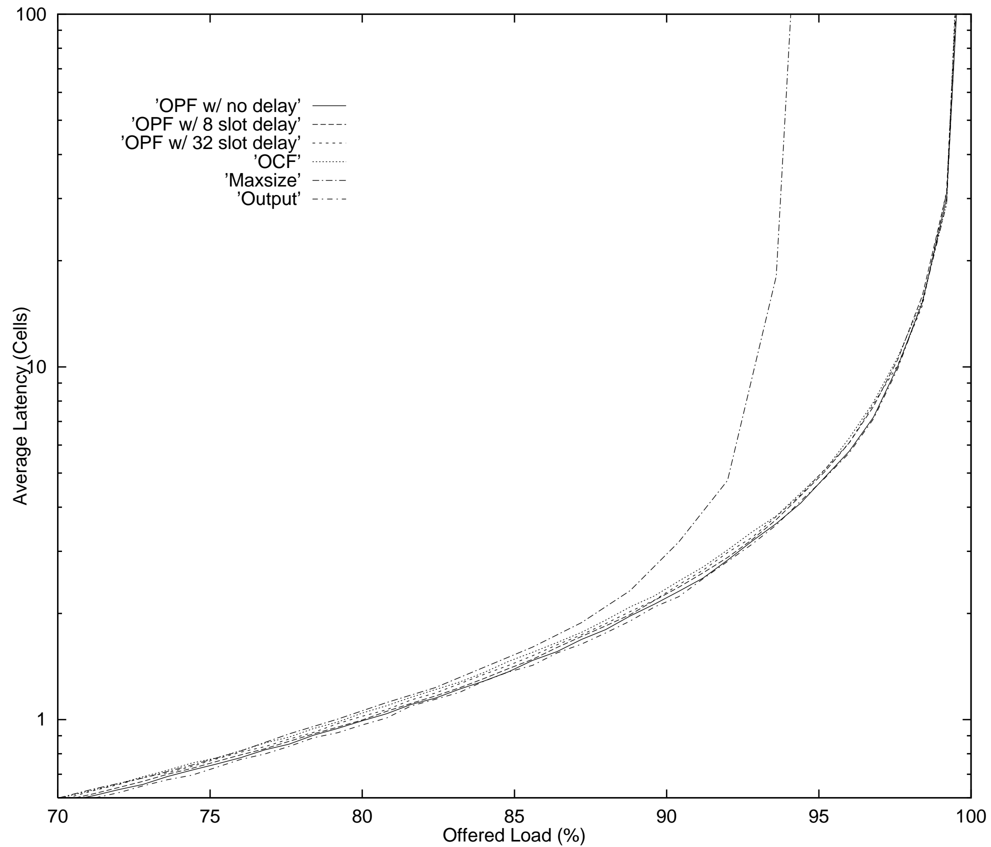


Figure 4.9 Performance comparison of pipelined OPF with various pipeline delays with OCF, a conventional maxsize algorithm and output queueing. The graph shows simulation results of the average cell latency as a function of offered load of 16×16 switches under non-uniform traffic. Arrivals at each input are Bernoulli i.i.d.

6 Starvation Problem

OPF solves part of the starvation problem. It prevents starvation that is caused by selecting a maxweight match in the same way that OCF does. Among all matches of the

same size, OPF selects one with the largest total weight to favor old cells, preventing starvation due to selecting a maxweight match.

However, it cannot prevent starvation due to selecting a maxsize match as explained in Chapter 3 because OPF always selects a maxsize match.

CHAPTER 5

The *i*LPF and *i*OPF Algorithms

1 Introduction

In this chapter, we introduce two iterative algorithms for approximating LPF and OPF. The algorithms are referred to as *i*LPF when approximating LPF and as *i*OPF when approximating OPF. LPF and OPF, though simpler than LQF and OCF, are still too complex and slow for high-bandwidth switches.[1] For high speed applications, we need to find ways to closely approximate LPF and OPF. Unlike the approximation of LQF and OCF, *i*LPF and *i*OPF are more practical for high-bandwidth switches. Because LPF and OPF use a maxsize algorithm, *i*LPF and *i*OPF can use a variety of fast and efficient algorithms (such as *i*SLIP and WFA) that approximate a maxsize algorithm [49][66]. As we will see in this chapter, *i*LPF and *i*OPF perform well under non-uniform traffic and are as fast as *i*SLIP and WFA.

For both *i*LPF and *i*OPF, we also present a detailed hardware design along with performance and complexity analysis.

1. With today's technologies, and for a switch of a moderate size, the modified maximum size matching that LPF uses cannot find a match within a 10-40 ns range.

2 Approximating LPF and OPF

As shown in Figure 5.1, the implementation of the approximating algorithm is very similar to that of the original algorithm. It still consists of two sorters, two crossbars, and a block that now finds a maximal size match.[1] The block can be implemented in a number of ways [2][53][63][66]. As in the case of LPF and OPF, the heuristic algorithm cannot use any maximal size matching algorithm: it must give preference to presorted requests as described in Chapter 3. However, we can borrow heavily from existing techniques.

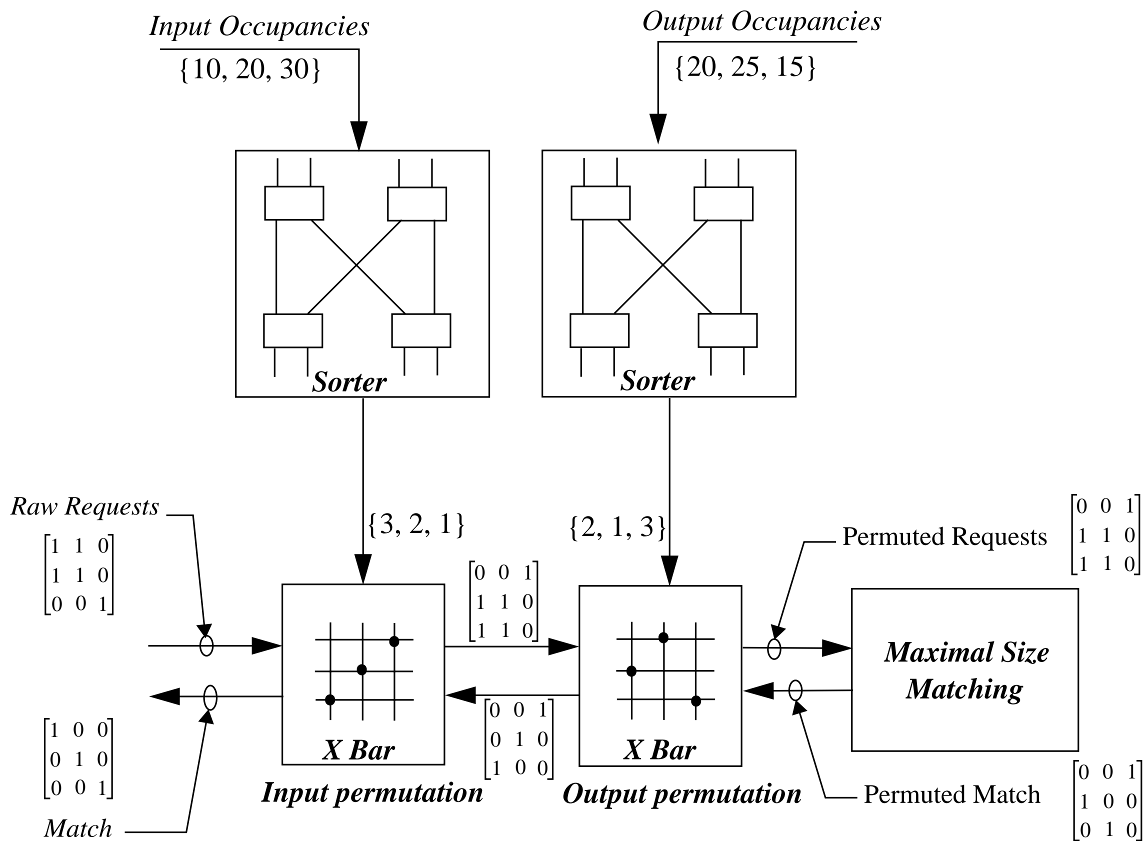


Figure 5.1 A block diagram of *i*LPF (for *i*OPF the inputs to the sorters are the waiting times rather than the occupancies). Except for the matching block, all blocks are functionally identical to the ones for LPF. Instead of finding a maxsize in accordance with LPF requirement, the matching block finds a maximal size match that approximates an LPF match.

1. One that leaves no request from an unmatched input to an unmatched output [2].

Borrowing from *i*SLIP, the first heuristic algorithm is a three-step iterative algorithm; the three steps are request, grant and accept. Our second heuristic algorithm is also iterative. It uses a double for-loop instead of a three-step approach. There are a number of ways to implement this algorithm, for example, using the arbiter array used in the Wave Front Arbiter (WFA) [8][9][66].

2.1 Three-step Algorithm

The three-step algorithm shown in Figure 5.2 is a simplified version of *i*SLIP. Essentially, it is *i*SLIP with preference always given to the lowest numbered input and output. Each output searches for a request from top to bottom, and each input searches for a granted request from left to right. As we shall see in Section 3, removing the pointers from *i*SLIP makes the three-step algorithm faster and simpler to implement (i.e. less logic). Otherwise, the algorithm is very similar to *i*SLIP. The implementation techniques utilized

3-STEP Algorithm.

Step 1 *Request*. Every unmatched input sends a request to every output for which it has a queued cell.

Step 2 *Grant*. Each unmatched output grants to the first requesting input. The output then notifies each input whether or not its request was granted.

Step 3 *Accept*. Each unmatched input accepts the first granting request, therefore completing its matching.

Repeat Steps 1-3 The algorithm stops after N iterations or if the previous step 3 found no match.

Figure 5.2 Similar to *i*SLIP, every input and output searches in round robin fashion. But unlike *i*SLIP, the round robin pointers do not move since the requests are presorted as described in Chapter 3: the pointers either point to the first input or the first output of a presorted request matrix. Fixing the pointers greatly simplifies hardware implementation, resulting in a faster and smaller arbiter. Instead of using a programmable priority encoder to perform grant and accept arbitration as in *i*SLIP [40][50], both *i*LPF and *i*OPF can use a much simpler priority encoder.

for *i*SLIP can also be applied to the three-step algorithm. This includes the grant and accept pipelining technique used for *i*SLIP, *i*LQF and *i*OCF described in Chapter 2.

Performance-wise, the three-step algorithm is not without its shortcomings. The algorithm is greedy: no connection made in an earlier iteration is removed even if doing so would result in a larger match. As we will see, this sub-optimality causes the algorithm to experience higher average latency when compared to less greedy algorithms, such as a maximum size matching algorithm.

2.2 Double For-Loop Algorithm

The double for-loop heuristic is as shown in Figure 5.3. Because one input and one output are considered at a time, the double for-loop is less greedy than the three-step algorithm. Although at first the algorithm appears to have the running time of $O(N^2)$, concurrency is possible, allowing the reduction to $O(N)$. The concurrency behaves as follows. As illustrated using WFA's block diagram in Figure 5.4, while an output is considering input i, its next smaller output (its right neighbor) can begin to consider any input which is larger than input i without waiting for the larger output to finish examining all inputs.

Double-For Loop Algorithm

1 **for** each output from the first to the last
2   **for** each input from the first to the last
3   **if (**there is a request) and (both input and output are unmatched)
4   **then** match them

Figure 5.3 An iterative algorithm approximating LPF or OPF using a double for-loop.

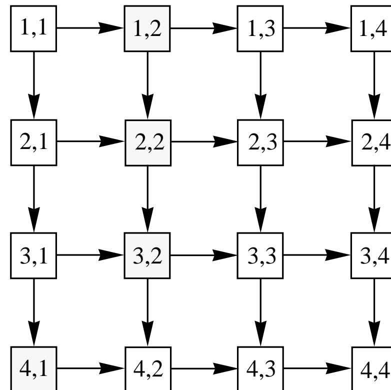


Figure 5.4 A block diagram of the Wave Front Arbiter for a 4×4 switch demonstrating the concurrency of the double for-loop. This figure shows that while output 1 is attempting to match input 4, the second output can consider the other three inputs because they have already been rejected by the first output (i.e. they have no request for the first output).

3 Implementations

3.1 Sorters and Crossbars

The design of the sorters and the crossbars is relatively straightforward with many design choices [25][71]. The crossbars are the least complex, and designs are widely available in the literature [62]. Therefore, the following summarizes only some important design considerations for LPF and *i*OPF. Depicted in Figure 5.5 is a simple design of a crossbar. In one of our designs,[1] which looks very much like Figure 5.5 and uses an NMOS transistor to implement a cross point, the propagation delay is less than 1 ns.

Various designs can be used for the sorters [35] with a shuffle-exchange sorting network providing a good trade-off in terms of silicon area and speed [3][25]. However, *i*LPF and *i*OPF do not need the sorted input values but rather the list of input numbers associated with the sorted values. Therefore, an extension to a basic sorting network is necessary

1. A 32×32 crossbar implemented in 0.25 μm CMOS technology.

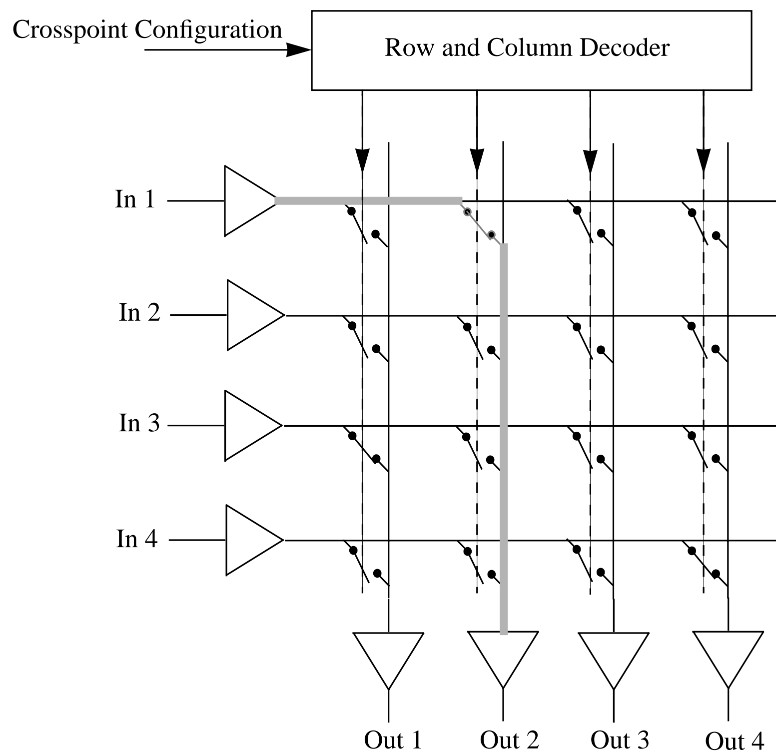


Figure 5.5 A schematic of a 4×4 crossbar. Mainly, it can be divided into a datapath and a controller. The datapath consists of the crosspoint switches, the wiring and the buffers. The controller includes the row and column decoder, which decides the on-off state of every switch. The wiring from the controller to all switches is shown by the dotted lines. The shaded lines indicate the critical path from the inputs to the outputs. Configuration information is supplied by the sorters.

to give the kind of outputs that *i*LPF and *i*OPF need. Figure 5.6 shows an 8×8 modified sorter. Besides the use of special 2×2 switching elements, the design is identical to a standard shuffle-exchange sorter [3][71]. We consider here the sorter for the input values in *i*LPF. The sorter for outputs and for *i*OPF are identical. At the first stage, inputs are the occupancies associated with an input number. At the last stage, the occupancy values are discarded leaving only the input port numbers, which are used to configure the crossbar. For an $N \times N$ switch, the number of stages grows at rate $\frac{1}{2}((1 + \log_2 N) \log_2 N)$, and each stage contains exactly $\frac{N}{2}$ switching elements [3][25]. Hence, the total number of

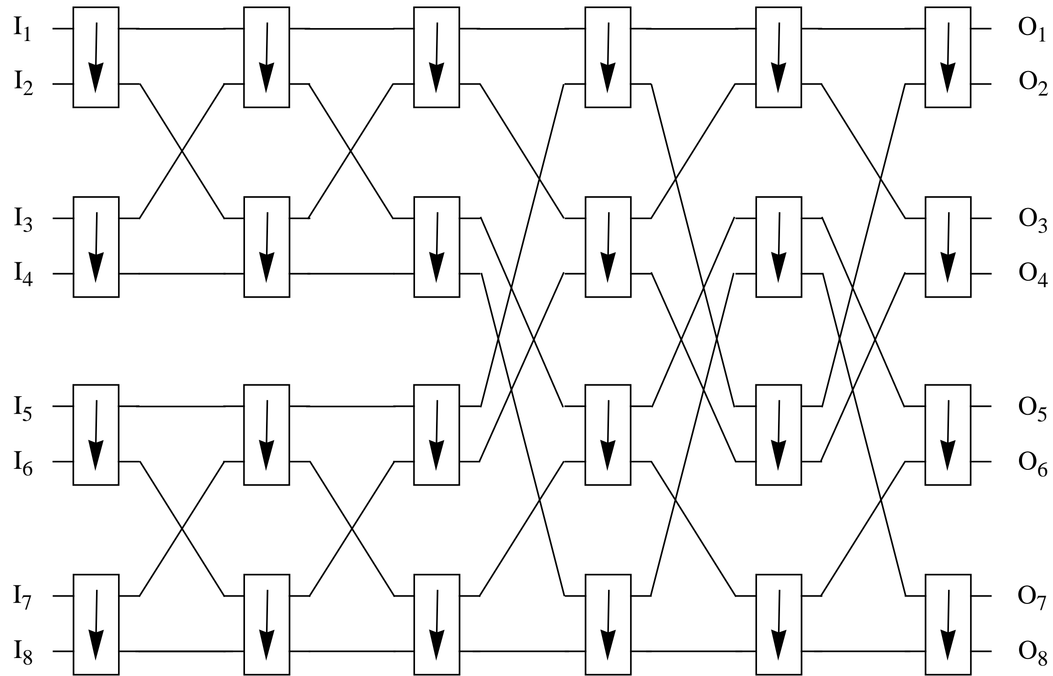


Figure 5.6 An 8×8 shuffle-exchange sorting network for an input sorter in *i*LPF. The network consists of six stages of the interconnected 2×2 switching elements shown in Figure 5.7. Each input is comprised of two integer values: one represents an input occupancy and the other identifies the input. The outputs of the network are sorted in ascending order from top to bottom based on the input port occupancies.

switching elements is $O(N(\log_2 N)^2)$. A design of a switching element is shown in Figure 5.7.

A static timing analysis[1] suggests a propagation delay of less than 1 ns for the switching element.[2] The pipelining of the sorters can be accomplished by inserting flip-flops between switching stages. Once the propagation delay of each stage is known and the slot time is specified, the minimum number of pipeline stages and equivalently the maximum number of switching stages between each pipeline stage can be calculated. For example,

1. Using Synopsys [39].
2. With ten bits representing each value and implemented in 0.25 μm CMOS technology.

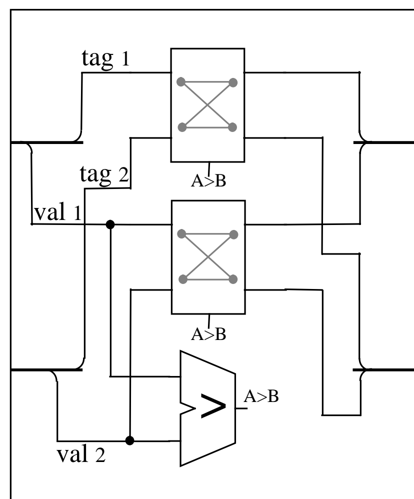


Figure 5.7 A switching element for the sorting network in Figure 5.6. Each input to the element consists of a tag and a value. The magnitude comparator compares the two values and configures the middle 2×2 switch so that the largest value is switched to the bottom output, and the smaller value is switched to the top output. The same switch configuration is also used to configure the top switch to switch the two tags in the same way. The tags flow out along with the values.

with a 1 ns delay per stage, 10 ns slot time and 1 ns flip-flop delay (setup time plus clock-to-output delay), the maximum number of switching stages between two pipeline flip-flops is nine.

3.2 Three-Step Algorithm

The implementation of the three-step algorithm at a block level, shown in Figure 5.8, is almost identical to *i*SLIP [53]. The only difference is that a complex arbiter is replaced by a simple priority encoder. As a result, the *i*LPF arbiters no longer dominate the iteration time, and take less silicon area than that of *i*SLIP. A wide range of speed and area trade-offs is available to implement a priority encoder. For area critical applications, a ripple design, an example of which is shown in Figure 5.9, is more desirable. For speed critical applications a look-ahead design, also known as a parallel design, an example of which is shown in Figure 5.10, may be more suitable because it is faster than a ripple design.

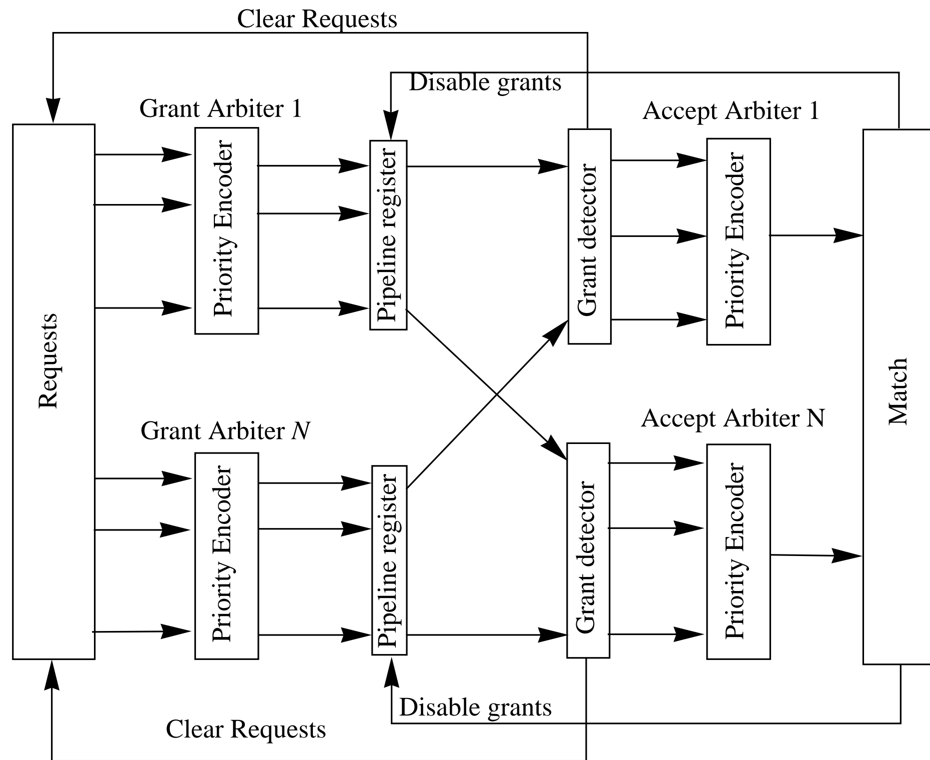


Figure 5.8 A block diagram of the three-step algorithm. Shown here with the iteration pipelining technique described in Chapter 2, this diagram only differs from the one in Figure 2.6 in the places where the priority encoders replace the magnitude comparators.

However, this task of exploring the design space is more conveniently left to logic synthesis [40].

As compared to the 1.74 ns running time and the area equivalent to 603 inverters of an *i*SLIP arbiter,[1] a look-head priority encoder takes only 0.61 ns and an area equivalent to 209 inverters for a 32×32 switch.

1. Detailed in Chapter 1.

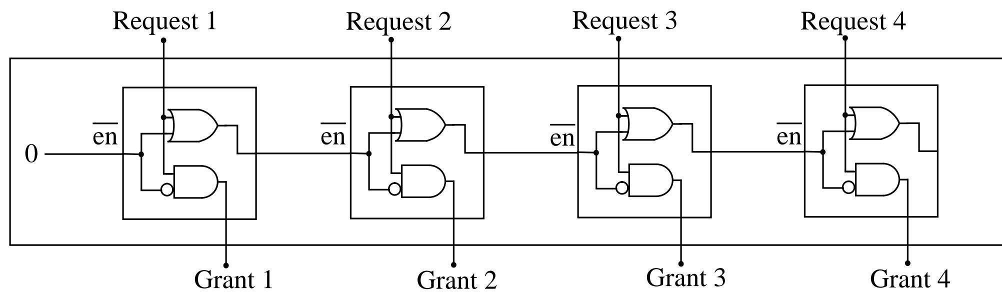


Figure 5.9 A four-input ripple priority encoder. The encoder, with requests as the inputs and grants as the outputs, consists of serially connected simple cells. As connected, priority is given to the inputs from left to right. The $\overline{en}$ signals are used to indicate the existence of a higher priority request. The rippling of the $\overline{en}$ signals starts from the output of the first cell and ends at the last cell. As a result, only the output of the first cell with an active input is asserted.

3.3 Double For-Loop Algorithm

The double for-loop algorithm is simpler to implement. In general, a range of design trade-offs can be easily explored using logic synthesis. WFA can also be modified to implement the double for-loop algorithm. The priority rotation mechanism of WFA [8][9] is disabled so that it always gives preference to the inputs and the outputs from first to last

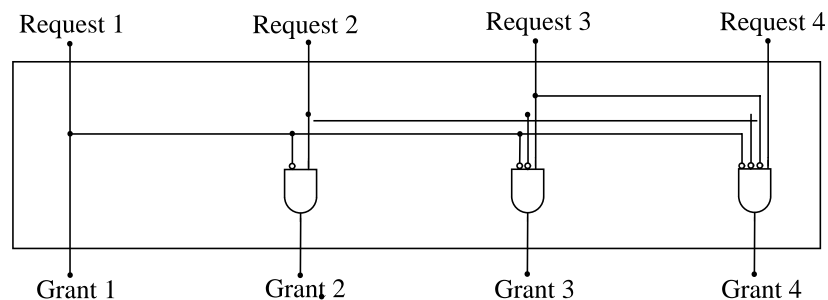


Figure 5.10 A four-input look-ahead priority encoder. Functionally equivalent to the one in Figure 5.9, the encoder generates each output in parallel and is therefore faster. However, for a larger number of inputs, this implementation takes a larger silicon area.

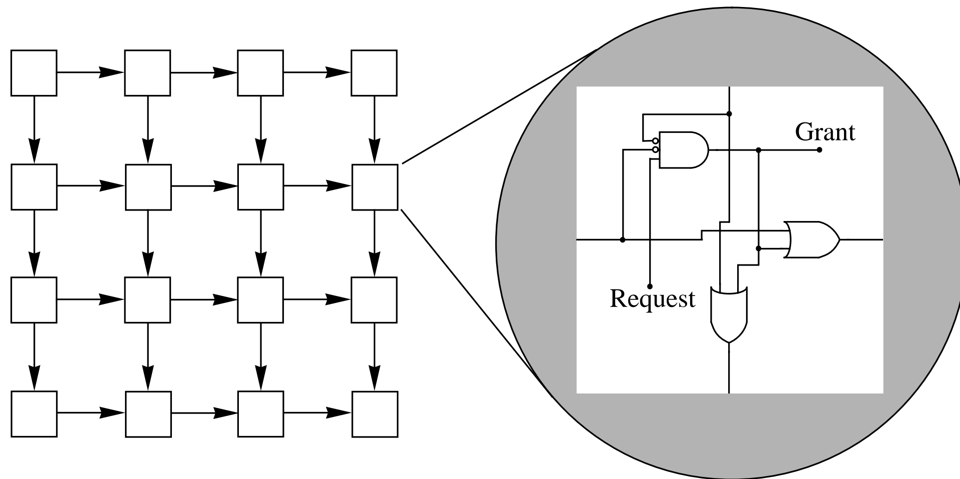


Figure 5.11 One of several ways to implement the double for-loop algorithm. This implementation is a subset of WFA [66], which has the priority rotation removed. Each cell of the two dimensional interconnected array is a simple combinational logic shown in the shaded circle. Each cell holds a request one-to-one corresponding to a request matrix. As shown, the request of a particular cell is granted if there is no cell from the left and no cell from the top with a granted request (see section 2.4 on page 18). The worst case delay per cell is the delay of the AND gate plus the delay of the OR gate.

(i.e. it searches a request matrix left-to-right and top-to-bottom). A schematic implementation for a 4×4 switch is shown in Figure 5.11. With a 50-100 ns propagation delay per cell,[1] the running time of the implementation in Figure 5.11 for a 32×32 switch is approximately 6.3-12.6 ns.

4 Comparison with *i*LQF and *i*OCF

4.1 Hardware Complexity

Table 5.1 shows estimates of the silicon area for each functional block obtained from logic synthesis for a 32×32 switch. The purpose of this estimate is to provide a rough indication of the hardware complexity of each algorithm. It is important to note that this estimate does not take into consideration other aspects such as area occupied by wiring,

1. Under a typical condition using Texas instruments' 0.25 μm CMOS technology.

Block Name	**Quantity**	***i*LQF**	***i*LPF: three-step**	***i*LPF: double for-loop**
Arbiter array	1	—	—	18688
crossbar (16Kx4)	2	—	128000	128000
Sorters	2	—	113040	113040
Grant Arbiters	32	106640	6688	—
Accept Arbiters	32	106640	6688	—
Maskable Decoders	32	8080	—	—
Maskable Registers	32	—	6424	—
Grant Detectors	32	776	776	—
Matches	1	7168	7168	—
Request	1	7168	7168	—
Weight Register	64	4048	—	—
Total Area		**240520**	**275952**	**259728**

Table 5.1 A silicon area estimate of each functional block of *i*LQF and *i*LPF for a 32×32 switch (measured in a unit equivalent to the size of a minimum size inverter).[1]

1. From Texas Instruments' 0.25 μm CMOS standard cell library.

peripheral circuitries around the core of the schedulers, and gate fanins and fanouts. These may substantially increase the total area.

The total silicon area needed for *i*LQF, as indicated at the bottom of Table 5.1, is about 13% smaller than the areas of both implementations of *i*LPF. The crossbars and the sorters of *i*LPF make up approximately 87% of the total area: a consequence of our design choice to minimize the propagation delay through the crossbars and to minimize the pipeline delay through the sorters. Each crosspoint of the crossbars in this design is thirty-two bits wide so that the crossbars can permute a request matrix in one pass, and the sorters have

only one pipeline stage at their outputs. If a smaller design is required, the crosspoints can be made as small as one bit wide, resulting in thirty two times less area for the crossbars, or one sorter can be eliminated by time sharing a sorter between inputs and outputs. With one bit wide crossbars and a time-sharing sorter, both implementations of *i*LPF can be made 60% smaller than *i*LQF.

4.2 Running Time

Tabulated in Table 5.2 are the running times for each functional block of *i*LPF and *i*LQF. With one slot pipeline delay after the sorters (to hide the sorting time), an estimate of the scheduling time for *i*LPF using the double for-loop can be calculated as follows:

Block Name	***i*LQF**	***i*LPF: three-step**	***i*LPF: double for-loop**
Arbiter array	—	—	0.14x63
Crossbar	—	0.2x4	0.8
Sorter	—	0.97x15	0.97x15
Grant Arbiter	7.38	0.61	—
Accept Arbiter	7.38	0.61	—
Maskable Decoder	0.19	—	—
Maskable Register	—	0.19	—
Grant Detector	0.64	0.64	—
Matches	0.19	0.19	
Request	0.19	0.19	
Weight Register	0.19	—	—

Table 5.2 A propagation delay estimate of each functional block of *i*LQF and *i*LPF for a 32×32 switch (measured in nanoseconds).

$$t_{scheduling} = 4t_{crossbar} + t_{arbiter} = 11.51ns. \tag{5.1}$$

For the three-step implementation, we consider the case of $\log_2 32$ iterations. The scheduling time estimate for the three-step algorithm with an iteration pipeline stage between the grant stage and accept stage in addition to the sorter pipeline is

$$t_{scheduling} = 4t_{crossbar} + 5(t_{accept} + t_{grant\,detect} + t_{request}) = 8ns. \tag{5.2}$$

For *i*LQF with the similar iteration pipeline and with the same number of iterations, the time estimate is

$$t_{scheduling} = 5(t_{accept} + t_{grant\,detect} + t_{request} + 2t_{register}) = 42ns. \tag{5.3}$$

Note that *i*LQF is approximately four times slower than *i*LPF. From Equation 5.3, the accept arbiter running time constitutes almost 88% of *i*LQF's scheduling time (36.9 ns out of 42 ns). As explained in Chapters 1 and 2, the slowness of the accept arbiter can be attributed to the magnitude comparators that it has inside. In contrast, the accept arbiter of *i*LPF (in the three-step algorithm) constitutes approximately 38% of the scheduling time (3.05 ns out of 8 ns).

Based on our experience designing the scheduler for *i*SLIP in the Tiny-Tera switch [39], the scheduling times of *i*LQF and *i*LPF with the three-step implementation may be significantly underestimated because of the omission of gate delays due to the fanins and the fanouts. For instance, each feedback from the grant detectors must drive thirty two registers, increasing the grant detector delay by at least 1 ns. So, it is unlikely that in practice the three-step with five iterations will be faster than the double for-loop implementation.

4.3 Performance Comparison using Simulation of the Three-Step and the Double For-Loop Algorithms

The following compares the performance in terms of throughput and average cell latency of the three-step and the double for-loop algorithms to that of *i*LQF, *i*SLIP and WFA. The comparison is based on simulation results for both uniform and non-uniform traffic.

4.3.1 Uniform Traffic

The graph in Figure 5.12 compares the latency of the three-step algorithm with the latencies of *i*SLIP and *i*LQF under uniform traffic. The graph in Figure 5.13 compares the

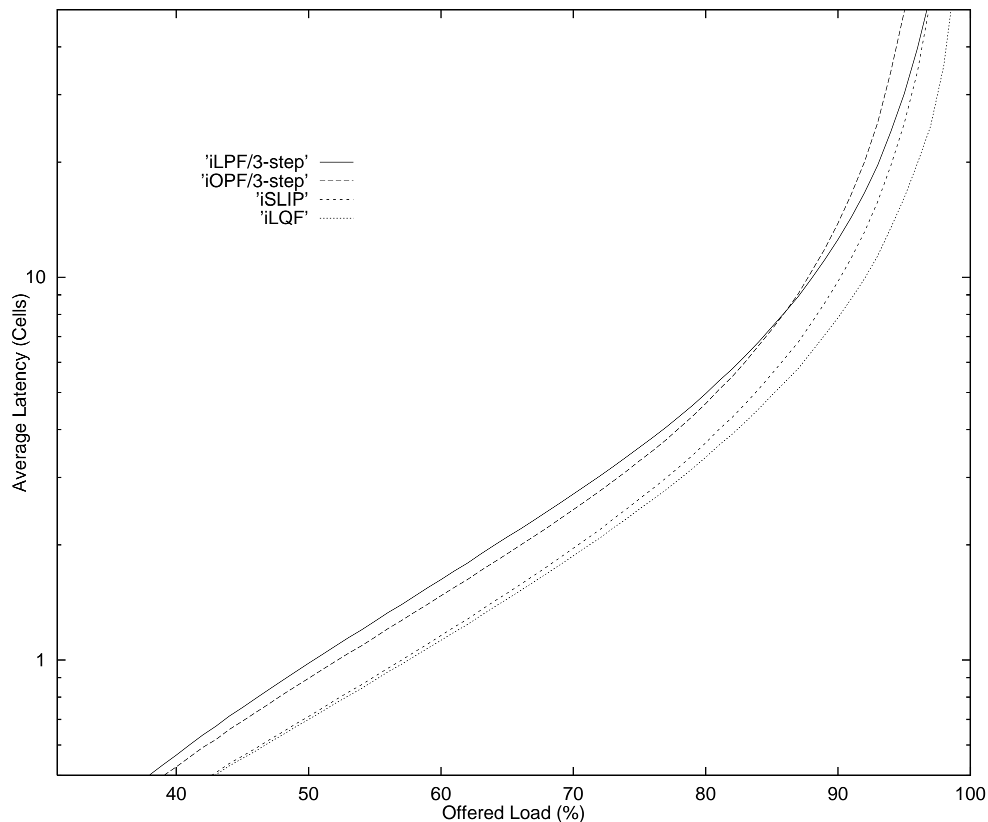


Figure 5.12 Performance comparison of *i*LPF and *i*OPF with the three-step implementation with *i*SLIP and *i*LQF. The number of iterations for all algorithms is sixteen. The graph shows the simulation results of the average cell latency as a function of the offered load of 16×16 switches under uniform traffic. Arrivals at each input are Bernoulli i.i.d.

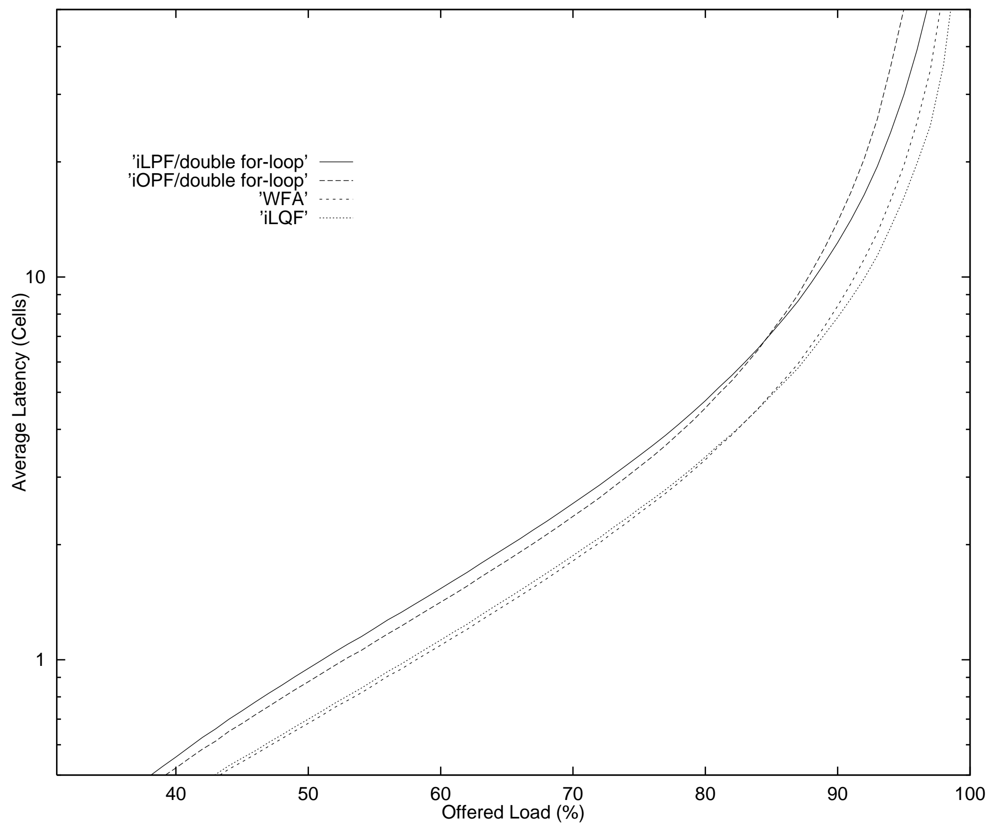


Figure 5.13 Performance comparison of *i*LPF and *i*OPF with the double for-loop implementation with WFA and *i*LQF. The number of iterations for all algorithms is sixteen. The graph shows the simulation results of the average cell latency of switches under the same traffic considered in Figure 5.12 as a function of offered load.

latency of the double for-loop algorithm with WFA and *i*LQF. Although all algorithms achieve 100% throughput,[1] their latencies differ noticeably. With both implementations, *i*LPF and *i*OPF experience higher latency than the other three algorithms. The cause for this increase in latency, which we call *matching blocking*, is explained in the following section. It has to do with the way in which *i*LPF and *i*OPF rearrange requests and because the three-step and the double for-loop are greedy algorithms. *i*LPF achieves lower latency than *i*OPF under a heavy load, an outcome similar to the latency comparison of *i*LQF and

1. Because the vertical axis does not show beyond 500 slots, all algorithms may appear to achieve less than 100% throughput.

*i*OCF [49]. Apparently, giving preference based on the queue occupancies has a tendency to result in lower latency than giving preference based on the waiting times.

The graph in Figure 5.14 compares the three-step algorithm with the double for-loop algorithm for both *i*LPF and *i*OPF. As shown by the graph, the latencies between the two implementations (algorithms) do not differ noticeably. Nevertheless, the numerical values of the data points of the graph show that the double for-loop implementation yields slightly lower latency than the three-step implementation, which is similar to the comparison between the latencies of WFA and *i*SLIP given in Chapter 1.

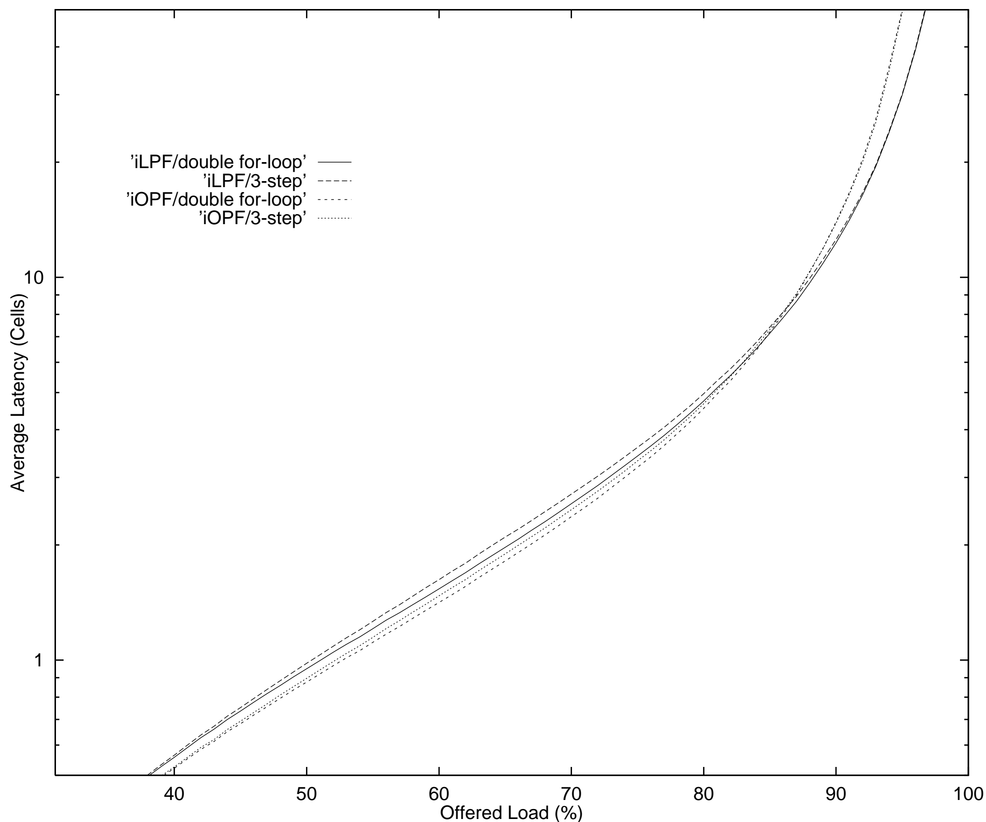


Figure 5.14 Performance comparison of the three-step implementation with the double for-loop implementation for *i*LPF and *i*OPF under the identical conditions as those in Figure 5.12 and Figure 5.13.

In addition to evaluating the latencies, we also used uniform traffic to study the performance of the three-step algorithm as a function of the number of iterations. Figure 5.15 shows simulation results of the three-step algorithm with two, four, eight, and twelve iterations. The results reveal a problem: the three-step algorithm seems to require more than the $\log_2 N$ iterations needed by iterative algorithms such as PIM, *i*SLIP and *i*LQF. The graph in Figure 5.15 indicates that the three-step algorithm needs at least twelve iterations while *i*SLIP and *i*LQF need only four iterations for a 16×16 switch.

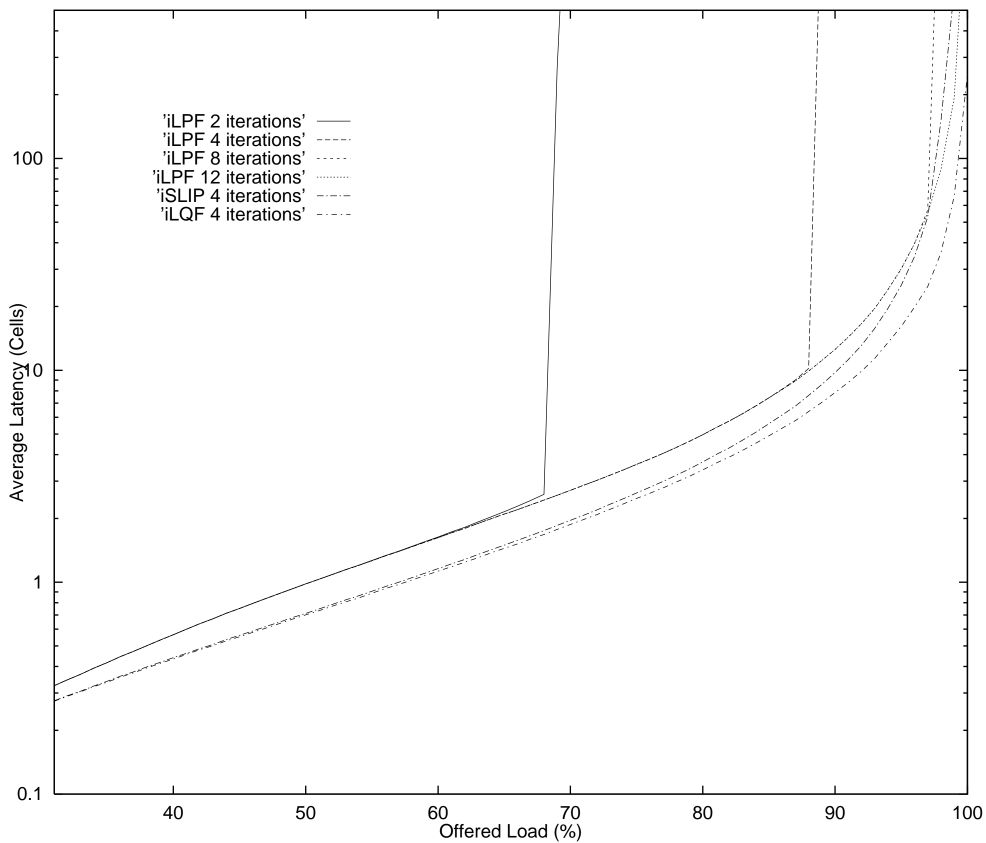


Figure 5.15 Number of iterations comparison of the three-step algorithm implementing *i*LPF with *i*LQF and *i*SLIP under uniform traffic identical to the one considered in Figure 5.12.

This problem is related to pointer synchronization (multiple round robin pointers preferring the same input or output), which is studied and presented in detail in [49]. For the three-step algorithm, all pointers are unfortunately perfectly synchronized: they all point to an input or to an output with the largest occupancy (or waiting time). As described in [49], the problem causes an iterative algorithm to match fewer number of inputs and outputs in each iteration and subsequently to need more iterations to complete the matching.

However, this problem does not pertain to the double for-loop algorithm because it always goes through the maximum number of iterations for the two nested loops (see section 3.3 on page 98). Thus, the double for-loop algorithm may be more desirable.

4.3.2 Non-uniform Traffic

The graph in Figure 5.16 compares the different algorithms for the non-uniform traffic pattern in Figure 1.9 of Chapter 1. Since the switch size is only 3×3, we do not consider cases with different numbers of iterations. *i*SLIP, *i*LQF and the three-step algorithm were given a maximum number of three iterations. For this traffic pattern, *i*LPF and *i*OPF achieve noticeably lower latency than the other algorithms. Hence, the effect of the matching blocking is not detectable from the graph. Reasons for the missing effect include the small size of the switch simulated, which does not promote matching blocking, and the non-uniformity of the traffic, which lessens the blocking.

As in the case of uniform traffic, *i*LPF achieves lower latency than *i*OPF for both implementations. However, between the two implementations, the latency is almost indistinguishable due to the small size of the switch.

5 Matching Blocking Problem

Like all iterative algorithms that do not allow backtracking, the three-step and the double for-loop algorithms cannot avoid the problem of matching blocking. Matching block-

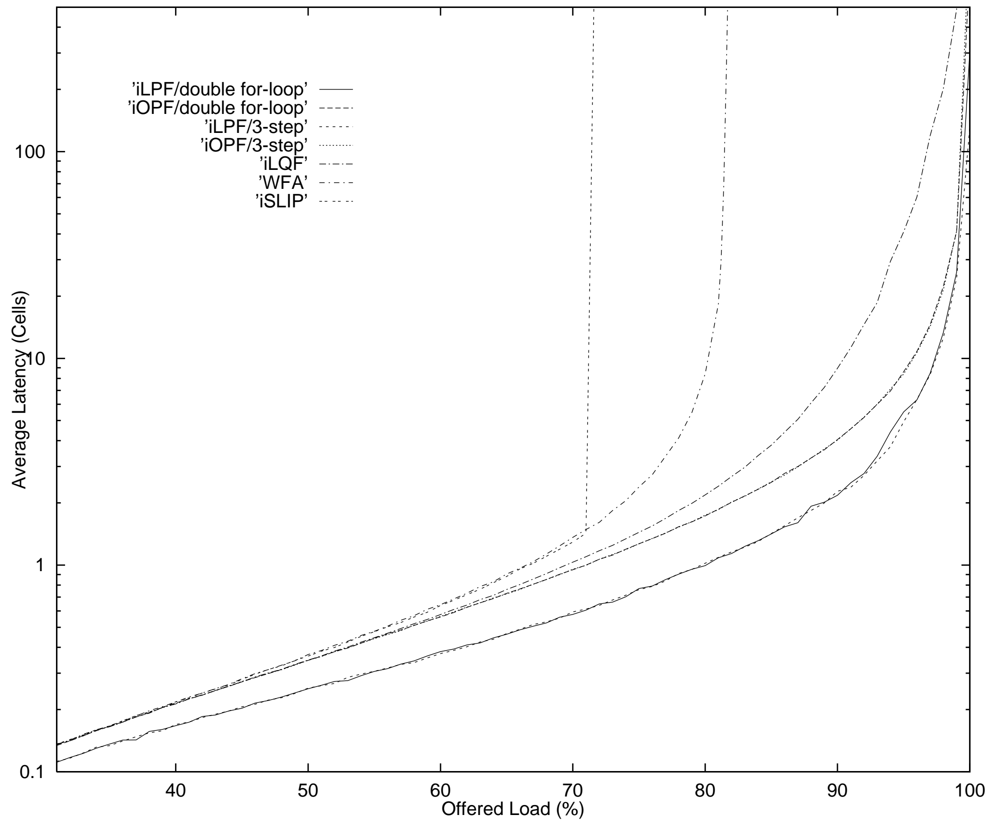


Figure 5.16 Performance comparison of *i*LPF and *i*OPF with *i*LQF, *i*SLIP and WFA. The graph shows simulation results of the average cell latency as a function of the offered load of 3×3 switches under non-uniform traffic shown in Figure 1.9. Arrivals at each input are Bernoulli i.i.d.

ing occurs when a match made in earlier iterations prevents (*blocks*) a scheduler from selecting a larger size match. The effect of matching blocking can vary from high latency to low throughput. Although we have not been able to find traffic under which *i*LPF and *i*OPF achieve less than 100% throughput, we have found that matching blocking can cause *i*LPF and *i*OPF to experience high latency for some traffic such as uniform traffic.

The effect of matching blocking is more noticeable in *i*LPF and *i*OPF than in other algorithms because *i*LPF and *i*OPF unintentionally stimulate matching blocking by giving preference to large inputs and outputs. Figure 5.17 illustrates how this problem can arise

a) Original request matrix with a WFA match shown by the circles.

b) Reordered request matrix with an *i*LPF match shown by the circles.

Figure 5.17 An illustration of a matching blocking problem in *i*LPF. The circles indicate matches found by WFA and *i*LPF, and the squares indicate a larger size match that *i*LPF should have selected.

for *i*LPF. Shown in Figure 5.17 (a) is an occupancy matrix of a 2×2 switch, which also happens to be an unweighted request matrix. Starting its search from the top-left corner toward the lower-right corner of the request matrix, WFA will find the match with two connections: input 1 to output 2 and input 2 to output 1, as indicated by the circles in Figure 5.17 (a). The double for-loop algorithm implementing *i*LPF, however, can make only one connection. As shown in Figure 5.17 (b), before finding a match, *i*LPF needs to give preference to the inputs and the outputs based on their occupancies by switching the first row with the second row and the first column with the second column.

When the double for-loop algorithm is applied to the reordered matrix, similar to WFA, it first considers the top input and the left output (the second input and the second output) and matches them. As a result, the other two requests cannot be matched because the input or the output that they want to be matched to has already been matched (input 2 and output 2), i.e., they are *blocked* by the matching of a higher priority input and output. So without unmatching the previous match, *i*LPF cannot improve a matching size nor a matching weight in this case.

6 Solution to the Matching Blocking Problem

The matching blocking problem can be solved by allowing a scheduling algorithm to remove previously made connections, i.e., to rip up a match, so that a larger match can be then selected. For instance, in the above example, if *i*LPF had known that the other two requests exist and had been allowed to unmatched the second input and the second output, it would unmatch the single connection match and then select the two connection match as depicted by the squares in Figure 5.17 (b). In other words, it would exchange the single connection match for the two connection match. This is a basic principle of a technique known in [63] as “extension-by-exchange.” Note that such an exchange exemplified in this figure does not result in the removal of an input or an output from a set of matched inputs or a set of matched outputs. Every matched input or output before the exchange takes place still remains matched after the exchange. This property guarantees that improving a match size does not result in a small weight match.

As indicated by the simulation result shown by the graph in Figure 5.18, such a technique substantially reduces the latency. The latency of *i*LPF with the exchange technique is almost as low as that of its optimum algorithm, LPF. However, this technique presents another design trade-off. While it reduces the latency, as described in [63], the exchange technique adds hardware complexity and substantially increases the scheduling time.

In conclusion, this chapter has presented two high speed approximating algorithms that can be used for both LPF and OPF. Our exploratory design work indicates that it is highly feasible to implement *i*LPF and *i*OPF for a 32×32 switch with a 10-40 Gbp/s line rate for a cell size of 53 bytes or larger.

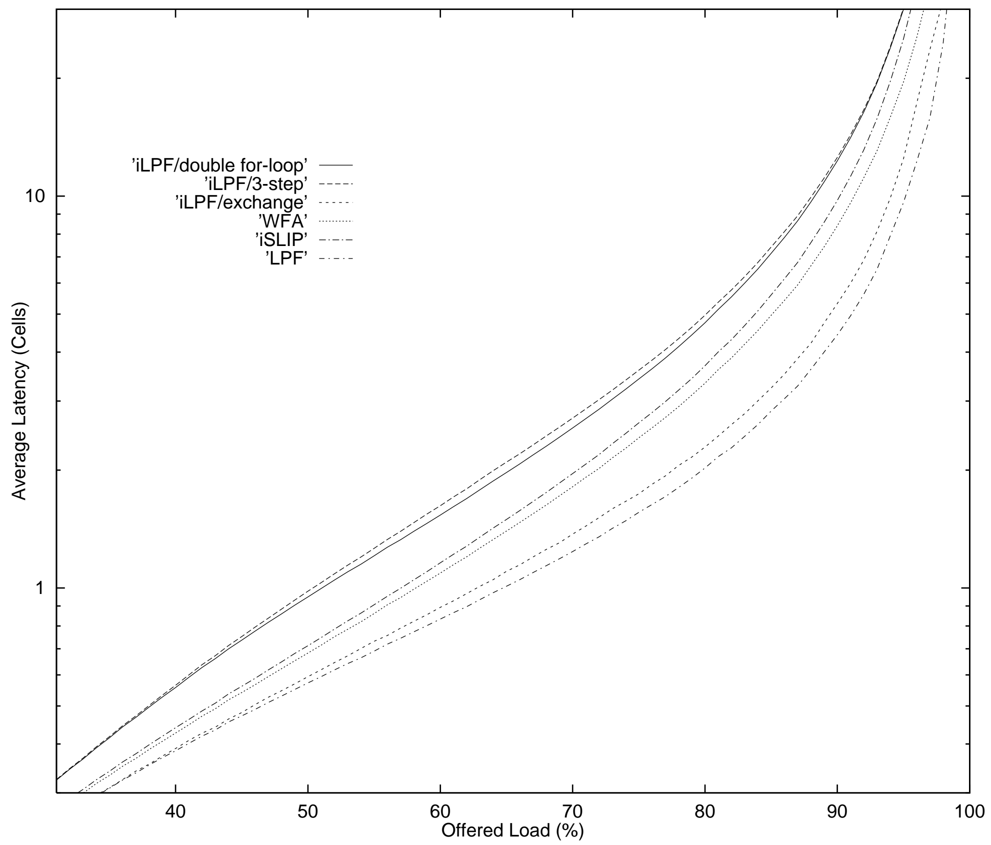


Figure 5.18 Performance comparison of *i*LPF (with the extension-by-exchange technique) with the three-step and the double for-loop implementations under uniform traffic identical to that of Figure 5.12 and Figure 5.13. The latency graphs of LPF, WFA and *i*SLIP are used as a reference.

CHAPTER 6

Conclusion

1 Summary

The background to this problem may be summarized as follows. Input-queued switches can achieve 100% throughput for *uniform* traffic using fast and fair scheduling algorithms implemented in dedicated hardware. Several scheduling algorithms have been designed for 32×32 switches making decisions in 50 ns. or less using 0.25 μm CMOS technology. However, as shown in Chapter 1, these algorithms can perform poorly when traffic is non-uniform.

While algorithms are known that achieve 100% throughput for non-uniform traffic, they are known to be too complex for implementation in fast hardware.

The LPF and the OPF algorithms introduced here demonstrated that achieving 100% throughput for all non-uniform traffic patterns does not necessarily compromise the scheduling speed. For a 32×32 switch, *i*LPF and *i*OPF can make a scheduling decision within 10 ns, which is as fast as achievable by algorithms good for uniform traffic only.

In terms of their practicality, our exploratory design work suggests that for a 32×32 switch *i*LPF and *i*OPF can be implemented using fewer than two million gates. As we have learned from designing the *i*SLIP scheduler, most of the gates are consumed not by the scheduler but by counters and flip-flops maintaining the states of queue occupancies as well as data formatting logics. The same is true for *i*LPF and *i*OPF. Among the two million gates, the core of the schedulers consists of fewer than three hundred thousand gates.

Performance-wise, simulation results show that LPF and OPF achieve lower average cell latency than previously reported algorithms. For some traffic patterns, they achieve almost the same latency as output-queueing. For *i*LPF and *i*OPF, however, we discover that they can experience higher cell latency for uniform traffic as shown Figure 5.12. As for the latency variance, OPF achieves a smaller variance than LPF, a result similar to the comparison of LQF and OCF.

With 10 ns per decision, *i*LPF and *i*OPF can allow a switch to operate at up to 40 Gbps line rate. For a 32×32 switch, this is equivalent to 1.28 Tbps aggregate throughput. Increasing switch size does not present a major problem in terms of the scheduling time because the running time of the maximal size matching (the double for-loop algorithm) scales linearly with the number of ports. The problem is with the number of gates required, which scales quadratically with the number of ports. For a 64×64 switch, we do not foresee a single chip implementation using 0.25 μm CMOS technology.

2 Future Research

Switch scheduling for terabit switches and routers is still a subject of active research. In addition to receiving high throughput, network users often have other requirements such as a delay guarantee, a low packet loss rate and data security. At the time of this writing, progress is being made in the areas of integrating QoS into the switch scheduler,[1] distributed algorithms[2] and, perhaps most exciting of all, emulating an output-queued switch using a form of input-queued switch.

2.1 QoS Integration

Integrating QoS directly into a switch scheduler still remains a difficult task at the present because quality and throughput in many cases are conflicting specifications. Often, it is boiled down to the issue of quality vs. quantity. A group of researchers at Massachusetts Institute of Technology (MIT) proposed an algorithm that is very similar to LQF and LPF.[3] Rather than using queue occupancies or waiting times to weight requests, their algorithm uses rate-based credits. However, they were unable to show that their proposed algorithm can achieve 100% throughput for all non-uniform traffic patterns.

2.2 Distributed Algorithms

In an attempt to solve the scaling problem of centralized schedulers, researchers turn to distributed algorithms. Because they are distributed, each individual scheduler is much simpler than a centralized scheduler. Distributed algorithms scale linearly or even sub-linearly with the switch size. Up until now proposed distributed algorithms have been limited for uniform traffic only. One of the problems in designing a distributed algorithm is distributing information among all schedulers residing at different inputs and outputs. It is harder to distribute multi-bit information such as queue lengths, waiting times and credits

---

1. Based on to-be-published manuscripts.
2. Also based on to-be-published manuscripts.
3. The manuscript is being reviewed for publication.

than one-bit requests. For a high-speed switch, this can be a problem because a significant portion of high-speed interconnections, otherwise available for packet forwarding, need to be allocated to interconnect the schedulers. Furthermore, the interconnection delay can increase the scheduling time substantially, especially those of iterative algorithms.

2.3 Output Queueing Emulation

Output queueing emulation is a novel approach to address the QoS issue. The idea is that if an input-queued switch also maintains queues at the outputs and the switching fabric is sped up moderately (two to four time the line rate), most arriving cells will be immediately forwarded to and queued at the outputs, making the switch appear as an output-queued switch. Once cells are queued at the outputs, each output can independently provide QoS, therefore, relieving the switch scheduler from this task.

While recent work shows that there exists a scheduling algorithm capable of perfectly emulating an output-queued switch with a speedup of two [11], the proposed algorithm is too complex for terabit switches and routers.

References

[1] Ali, M.; Nguyen, H.; “A neural network implementation of an input access scheme in a high-speed packet switch,” *Proceedings. of GLOBECOM 1989*, pp.1192-1196.

[2] Anderson, T.; Owicki, S.; Saxe, J.; Thacker, C.; “High speed switch scheduling for local area networks,” *ACM Transaction. on Computer Systems,* Nov. 1993, pp. 319-352.

[3] Awdeh R.Y.; Mouftah H.T.; “Survey of ATM switch architectures,” *Computer Networks & ISDN Systems*, 1995, vol. 27, pp. 1567-1613.

[4] Batcher K.E.; “Sorting networks and their applications,” *Proceedings of 1968 Spring Joint Computer Conference*.

[5] Bolot, J.C.; Crepin, H.; Garcia, A.V.; “Analysis of audio packet loss in the Internet,” *Proceedings of the 5th International Workshop on Network and Operating System Support for Digital Audio and Video*, Durham, April 1995, pp. 163-174.

[6] Chaney, T,; Fingerhut, J.A.; Flucke, M.; Turner J.S.; “Design of a gigabit ATM switch,” *Proceedings of IEEE INFOCOM ‘97*, Kobe, Japan, 7-11 April 1997, vol.1, pp. 2-11.

[7] Chen, M.; Georganas, N.D.; “A fast algorithm for multi-channel/port traffic scheduling,” *Proceedings of IEEE Supercom/ICC ‘94*, pp.96-100.

[8] Chi, H.C.; Tamir, Y.; “Starvation prevention for arbiters of crossbars with multi-queue input buffers,” *Proceedings of COMPCON ‘94*, San Francisco, February, 1994, pp. 292-297.

[9] Chi, H.C.; Tamir, Y.; “Decomposed arbiters for large crossbars with multi-queue input buffers,” *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, Cambridge, October, 1991, pp. 233-238.

[10] Choudhury, A.K.; Hahne L.E.; “Dynamic queue length thresholds in a shared memory ATM switch,” *Proceedings of IEEE INFOCOM ‘96*, San Francisco, March, 1996, vol.2, pp. 679-687.

[11] Chuang, S.-T.; Goel A.; McKeown, N.; Prabhakar, B.; “Matching output queueing with a combined input output queued switch,” *Computer Systems Technical Report* CSL-TR-98-758, March 1998.

[12] Cisco Systems; “Fast switched backplane for a gigabit switched router,” *http://www.cisco.com/warp/public/733/12000/technical.shtml*.

[13] Cormen, T.; Leiserson C.E.; Rivest R.L.; “Introduction to Algorithms,” The MIT Press, Cambridge, Massachusetts, March 1990.

[14] Coudreuse, J.P.; Servel, M.; “Prelude: an asynchronous time-division switched network,” *Proceedings of IEEE International Conference on Communications ‘87*, New York, 1987, vol.2, pp. 769-773.

[15] Dally, W.J.; Carvey, P.P.; Dennison, L.R.; “The Avici terabit switch/router,” *Proceedings of Hot Interconnects 6*, Stanford, Aug 1998, pp. 41-49.

[16] Demers, A.; Keshav, S.; Shenker, S.; “Analysis and simulation of a fair queueing algorithm,” *Internetworking: Research and Experience,* Sept. 1990, vol.1, no.1, pp. 3-26.

[17] Dinic, E.A.; "Algorithm for solution of a problem of maximum flow in a network with power estimation," *Soviet Math. Dokl.*, 1970, vol.11, pp. 1277-1280.

[18] Edmonds, J.; Karp, R.M.; "Theoretical improvements in algorithmic efficiency for network flow problems," *Journal of the Association for Computing Machinery*, April 1972. vol.19, no.2, pp. 248-264.

[19] Endo, N.; Kozaki, T.; Ohuchi, T.; Kuwahara, H.; Gohara, S.; "Shared buffer memory switch for an ATM exchange," *IEEE Transactions on Communications,* Jan. 1993, vol.41, no.1, pp. 237-245.

[20] Fowler, T.B.; "Internet access and pricing: sorting out the options," *Telecommunications*, February 1997.

[21] Giacopelli, J.; Hickey, J.; Marcus, W.; Sincoskie, D.; Littlewood, M.; "Sunshine: A high-performance self-routing broadband packet switch architecture," *IEEE J. Selected Areas Communications*, Oct 1991, vol.9, no.8, pp.1289-1298.

[22] Glynn, P.;" Applied probability," EESOR-321 class note, 1997.

[23] Hopcroft, J.E.; Karp, R.M.; "An $n^{5/2}$ algorithm for maximum matching in bipartite graphs," *Society for Industrial and Applied Mathematics J. Comput.,* 1973, pp.225-231.

[24] Huang, A.; Knauer, S.; "Starlite: A wideband digital switch," *Proceedings of GLOBECOM '84*, 1984, pp.121-125.

[25] Hui, J. N.; "Switching and traffic theory for integrated broadband networks," Kluwer Academic Publishers, Boston, 1990.

[26] Inukai, T.; "An efficient SS/TDMA time slot assignment algorithm," *IEEE Trans. Communications*, October, 1979, vol.COM-27, no.10, pp. 1449-1455.

[27] Joo, Y.M.; McKeown, N.; "Doubling memory bandwidth for network buffers," *Proceedings of INFOCOM*, San Francisco, April 1998, vol.2, pp. 808-815.

[28] Karol, M.; Eng, K.; Obara, H.; “Improving the performance of input-queued ATM packet switches,” *Proceedings of INFOCOM ‘92*, pp.110-115.

[29] Karol, M.; Hluchyj, M.; “Queueing in high-performance packet-switching,” *IEEE J. Selected Area Communications*, Dec. 1988, vol.6, pp.1587-1597.

[30] Karol, M.; Hluchyj, M.; and Morgan, S.; “Input versus output queueing on a space division switch,” *IEEE Trans. Communications*, 1987, pp.1347-1356.

[31] Katsube, Y.; Nagami, K.; Matsuzawa, S.; Esaki, H.; “Internetworking based on cell switch router-architecture and protocol overview,” *Proceedings of the IEEE*, December, 1997, vol.85, no.12, pp. 1998-2006.

[32] Kelley, F.P.; Williams, R.J.; “Stochastic networks,” Springer-Verlag, New York, 1995.

[33] Keshav, S.; Sharma, R.; “Issues and trends in router design,” *IEEE Communications magazine*, May 1998, vol.36, no.5, pp. 144-151.

[34] Kleinrock, L.; “Queueing systems,” Wiley, New York, 1974-1976.

[35] Knuth, D. E.; “The art of computer programming Vol III:sorting and searching,” Addison-Wesley, Reading, Massachusetts, 1997-1998.

[36] Koniq, D.; “Theorie der endlichen und unendlichen Graphen,” Chelsea, New York, 1950.

[37] Kreko, B.; “Lehrbuch der linearen Optimierung,” Pitman, London, 1968.

[38] Kumar, P.R.; Meyn, S.P.; “Stability of queueing networks and scheduling Policies”, *IEEE Transactions on Automatic Control*, Feb. 1995, vol.40, no.2, pp. 251-260.

[39] Kurup, P.; Taher A.; “Logic synthesis using synopsys,” Kluwer Academic, 1997, Boston.

[40] Kupta, P.; McKeown N.; “Design and implementation of a fast crossbar scheduler,” *Proceedings of Hot Interconnects 6*, Stanford, Aug 1998, pp. 77-84.

[41] Kupta, P.; "Scheduling in input queued switches: a survey," unpublished manuscript, http://www-cs-students.stanford.edu/~pankaj/research.html.

[42] LaMaire, R. O.; Serpanos, D. N.; "Two-dimensional round-robin schedulers for packet switches with multiple input queues," *IEEE/ACM Transactions on Networking*, Oct. 1994, vol. 2, no. 5, pp. 471-482.

[43] Lawton, G.; "In search of real-time Internet service," *IEEE Computer. Society*, Nov. 1997, vol.30, no.11, pp. 14-16.

[44] Li, M.; Yan, J.; "Dimensioning of line-access systems serving Internet traffic," *XVI International Switching Symposium*, Toronto, 21-26 September 1997, pp. 67-73.

[45] Liu, C. L.; "Topics in combinatorial mathematics," Mathematical Association of America, Massachusetts, 1972.

[46] Lund, C.; Phillips, S.; Reingold, N.; "Fair prioritized scheduling in an input-buffered switch," *Proceedings of the International IFIP-IEEE Conference on Broadband Communications*, Montreal, Que., Canada April 1996, pp. 358-69.

[47] Matsuzawa, S.; Nagami, K.; Mogi, A.; Jinmei, T.; Esaki, H.; Katsube, Y.; "Architecture of cell switch router and prototype system implementation" *IEICE Transactions on Communications*, Aug. 1997. vol.E80-B, no.8, pp. 1227-1238.

[48] McCluskey, E. J.; "Logic design principles: with emphasis on testable semicustom circuits," Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[49] McKeown, N.; "Scheduling algorithms for input-queued cell switches," Ph.D. Thesis, University of California at Berkeley, 1995.

[50] McKeown, N.; Izzard, M.; Mekkittikul, A.; Ellersick, W.; Horowitz, M.; "The Tiny Tera: a packet switch core," *Proceedings of Hot Interconnects IV*, Stanford, Aug 1996, pp. 161-173.

[51] McKeown, N.; Anantharam, V.; Walrand, J.; “Achieving 100% throughput in an input-queued switch,” *Proceedings of INFOCOM*, 1996, pp. 296-302.

[52] Mekkittikul, A.; McKeown, N.; “A starvation-free algorithm for achieving 100% throughput in an input-queued switch.”, *Proceedings of ICCCN’96*, October, 1996, pp. 226-231.

[53] Mekkittikul, A.; McKeown, N.; Izzard, M.; “A small high-bandwidth ATM switch.” *Proceedings of SPIE*, Boston, November, 1996.

[54] Mekkittikul, A.; McKeown, N.; “A practical scheduling algorithm to achieve 100% throughput in input-queued switches,” *Proceedings of INFOCOM*, 1998, San Francisco, 29 March - 2 April 1998, vol.2, pp. 792-799.

[55] Morgan, S.; Delaney, M.; “The Internet and the local telephone network: conflicts and opportunities,” *XVI International Switching Symposium*, Toronto, 21-26 September 1997, pp. 561-569.

[56] Naldi, M.; “Size estimation and growth forecast of the Internet,” *Centro Vito Volterra preprints*, University of Rome, Tor Vergata, Preprint no. 303, October 1997.

[57] Obara, H.; “Optimum architecture for input queueing ATM switches,” *IEE Electronics Letters*, 28th March 1991, pp.555-557.

[58] Obara, H.; Hamazumi, Y.; “Parallel contention resolution control for input queueing ATM switches,” *IEE Electronics Letters*, 23rd April 1992, vol.28, no.9, pp.838-839.

[59] Obara, H.; Okamoto, S.; and Hamazumi, Y.; “Input and output queueing ATM switch architecture with spatial and temporal slot reservation control,” *IEE Electronics Letters*, 2nd Jan 1992, pp.22-24.

[60] Parekh, A.K.; Gallager, R.; “A generalized processor sharing approach to flow control in integrated services networks: the multiple node case,” *IEEE/ACM Transactions on Networking,* April 1994, vol.2, no.2, pp. 137-150.

[61] Partridge, C.; et al.; “A 50-Gb/s IP router,” *IEEE/ACM Transactions on Networking*, June 1998, vol.6, no.3, pp. 237-248.

[62] Patyra, M.; Maly, W.; “Circuit design for a large area high-performance crossbar switch,” *Proceedings. 1991 International Workshop on Defect and Fault Tolerance on VLSI Systems*, 18-20 Nov. 1991, pp. 32-45.

[63] Rose, C.; “Rapid optimal scheduling for time-multiplex switches using a cellular automaton,” *IEEE Transactions on Communications,* May 1989, vol.37, no.5, pp. 500-509.

[64] Suzuki, H.; Nagano, H.; Suzuki, T.; Takeuchi, T.; Iwasaki, S.; “Output-buffer switch architecture for asynchronous transfer mode,” *IEEE International Conference on Communications*, Boston, June 1989, vol 1, pp. 99-103.

[65] Tamir, Y.; Frazier, G.; “High performance multi-queue buffers for VLSI communication switches,” *Proc. of 15th Ann. Symp. on Comp. Arch.*, June 1988, pp.343-354.

[66] Tamir, Y.; Chi, H.C.; “Symmetric crossbar arbiters for VLSI communication switches” *IEEE Transactions on Parallel and Distributed Systems,* Jan 1993. vol.4, no.1, pp. 13-27.

[67] Tarjan, R.E.; “Data structures and network algorithms,” *Society for Industrial and Applied Mathematics,* Pennsylvania, Nov 1983.

[68] Tassiulas, L.; Ephremides, A.; “Stability properties of constrained queueing systems and scheduling policies for maximum throughput in multihop radio networks,” *IEEE Trans. Automatic Control*, Dec. 1992, vol. 37, no. 12, pp.1936-1948.

[69] Tobagi, F.A.; “Fast packet switch architectures for broadband integrated services digital networks,” *Proceedings of the IEEE*, Jan. 1990, vol.78, no.1, pp. 133-167.

[70] Troudet, T.P.; Walters, S.M.; "Hopfield neural network architecture for crossbar switch control," *IEEE Trans. Circuits and Systems,* Jan.1991, vol.CAS-38, pp.42-57.

[71] Weste, N.; Eshraghian, K.; "Principles of CMOS VLSI design: a systems perspective," Addison-Wesley, Massachusetts, 1993.

[72] Zhang, H.; "Service disciplines for guaranteed performance service in packet-switching networks," *Proceedings of the IEEE*, Oct. 1995, vol.83, no.10, pp. 1374-96.

[73] Zhang, L;. "VirtualClock: a new traffic control algorithm for packet-switched networks," *ACM Transactions on Computer Systems,* May 1991, vol.9, no.2, pp. 101-124.

APPENDIX 1

Stability of an NxN Switch under OCF with Independent Arrivals

1 Definitions

The stability proof of OCF in this appendix is derived primarily from the properties of the waiting times of HOL cells in a switch and the properties of the arrival processes. Therefore, we first define such properties and related definitions. The properties can be observed as follows. Consider Figure A1.1. $C_{i,j}(n)$ is the HOL cell of $Q_{i,j}$ at slot n which arrived at slot $n - W_{i,j}(n)$ and, thus, has been waiting in the queue for $W_{i,j}(n)$ slots.

If the cell does not leave the queue, i.e., does not get served, its waiting time increases by one per slot:

$$W_{i,j}(n+1) = W_{i,j}(n) + 1 . \qquad (1)$$

If the cell leaves the queue (the queue is scheduled), a cell behind advances to the head of the queue, becoming the new HOL cell. From Figure A1.1, it can be seen that the waiting time of the new HOL is the previous HOL cell waiting time minus the interarrival time between the two cells, $\tau_{i,j}(n)$:

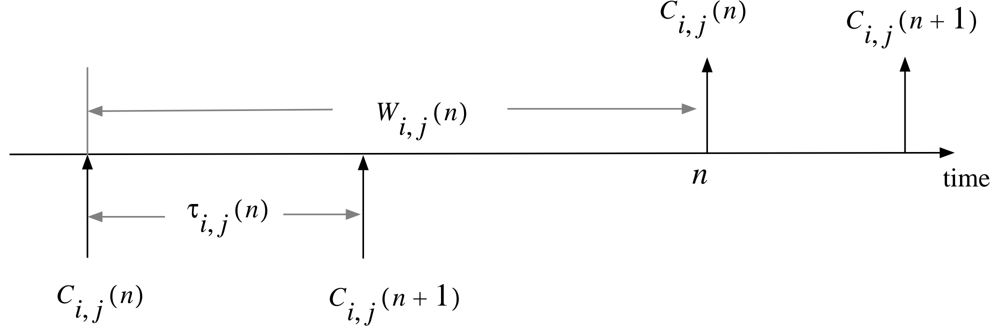


Figure A1.1 Arrivals and departures time line. Arrivals are shown below the line, departures are shown above the line. This figure shows the current HOL cell $C_{i,j}(n)$ departing at slot n and its successor $C_{i,j}(n+1)$ departing some slots later.

$$W_{i,j}(n+1) = W_{i,j}(n) - \tau_{i,j}(n), \quad \forall \tau_{i,j}(n) \le W_{i,j}(n). \tag{2}$$

In the case where there is no arrival while $C_{i,j}(n)$ is waiting in the queue, the queue becomes empty when $C_{i,j}(n)$ leaves the queue. Hence, the waiting time in the next slot is defined to be zero:

$$W_{i,j}(n+1) = 0, \qquad \forall \tau_{i,j}(n) > W_{i,j}(n). \tag{3}$$

Associated with the above waiting time equations, we define the approximate next state vector of the waiting times as follows:

$$\underline{\tilde{W}}(n+1) \equiv \underline{W}(n) + \underline{1} - [\underline{S}(n) \cdot \underline{\tau}(n)]. \tag{4}$$

It is worth noting that $\tilde{W}_{i,j}(n)$ can become negative when $\tau_{i,j}(n) > W_{i,j}(n)$ while $W_{i,j}(n)$ can never become negative.

Further, we derive the following properties from the properties of the arrival processes as described in Chapter 1. These properties are needed for the proofs of the subsequent lemmas.

Property 1 $\tau_{i,j}(n)$ is independent of a waiting time, $W_{i,j}(n), \forall i, j, n$.

Property 2 $\tau_{i,j}(n) \geq 1$. Since there is only at most one arrival per slot, the arrival time of any two consecutive cells must be at least one slot apart.

Property 3 $W_{i,j}(n) \geq L_{i,j}(n), \forall i, j, n$ because there is at most one arrival per slot.

Property 4 The sum of the weights of all granted requests, $\sum_{(i,j) \in M} W_{i,j}(n)$, is equal to $\underline{W}^T(n)\underline{S}(n)$.

Property 5 For any queue whose arrival rate is zero, $\lambda_{i,j} = 0$, $L_{i,j}(n) = 0$; thus $W_{i,j}(n) = 0, \forall n$. Considering the fact that a zero waiting time does not contribute to the sum value, $\underline{W}^T(n)\underline{S}(n)$, without loss of generality, we can set the corresponding service indicator, $S_{i,j}(n)$ to zero for all time, $S_{i,j}(n) = 0, \forall n$.

In addition, we also define the following.

1. A positive-definite diagonal matrix, T_{ocf}, whose diagonal elements are $\{\lambda_{1,1}, \ldots\lambda_{1,N}, \ldots\lambda_{N,1}, \ldots\lambda_{N,N}\}$.

2. $[\underline{a} \cdot \underline{b} \cdot \underline{c}]$ denotes a vector in which each element is a scalar product of the corresponding elements of vector $\underline{a}$, $\underline{b}$, and $\underline{c}$, i.e., the i, j th element is equal to $a_{i,j} \cdot b_{i,j} \cdot c_{i,j}$.

2 Main Theorem

Theorem 1.1: *Under the OCF algorithm, the queue occupancies are stable for all admissible and independent arrival processes, i.e.,* $E[\|\underline{L}(n)\|] \leq C < \infty$.

3 Proof

The proof is carried out in two steps. First, we prove the stability of the waiting times. Then, we show that the stability of the waiting time implies the stability of the queue occupancies. The proof requires the following lemmas and theorem.

Lemma 1: *Under the OCF algorithm,* $\underline{W}^T(n)\underline{\lambda} - \underline{W}^T(n)\underline{S}^*(n) \leq 0, \forall \underline{W}(n), \underline{\lambda}$, *where* $\underline{S}^*(n)$ *is s.t.* $\underline{W}^T(n)\underline{S}^*(n) = max\left(\underline{W}^T(n)\underline{S}(n) \right)$.

Proof: Consider the linear programming problem:

$$max\left(\underline{W}^T(n)\underline{\lambda} \right) \tag{5}$$

$$\text{s.t.} \sum_{i=1}^{N} \lambda_{i,j} \leq 1, \sum_{j=1}^{N} \lambda_{i,j} \leq 1, \lambda_{i,j} \geq 0 \tag{6}$$

Λ is a doubly stochastic matrix and forms a convex set, C, with the set of extreme points equal to permutation matrices, S [51]. Therefore, the following are true.

$$max\left(\underline{W}^T(n)\underline{\lambda} \right) \leq \underline{W}^T(n)\underline{S}^*(n) . \square \tag{7}$$

Lemma 2: *Under the OCF algorithm,*

$$E\left[\underline{\tilde{W}}^T(n+1)T_{ocf}\underline{\tilde{W}}(n+1) - \underline{W}^T(n)T_{ocf}\underline{W}(n)\middle|\underline{W}(n)\right] \le K,\ \forall\underline{\lambda},$$

where K is a constant.

Proof:

By expansion,

$$\begin{aligned}
&\underline{\tilde{W}}^T(n+1)T_{ocf}\underline{\tilde{W}}(n+1) \\
&= (\underline{W}(n) + \underline{1} - [\underline{S}^*(n)\cdot\underline{\tau}(n)])^T T_{ocf}(\underline{W}(n) + \underline{1} - [\underline{S}^*(n)\cdot\underline{\tau}(n)]) \\
&= \underline{W}^T(n)T_{ocf}\underline{W}(n) + 2\underline{W}^T(n)\underline{\lambda} - 2\underline{W}^T(n)[\underline{S}^*(n)\cdot\underline{\tau}(n)\cdot\underline{\lambda}] \\
&\quad + \sum_{i,j}\lambda_{i,j} - 2\sum_{i,j}S^*_{i,j}(n)\cdot\tau_{i,j}(n)\cdot\lambda_{i,j} + \sum_{i,j}S^*_{i,j}(n)\cdot\tau^2_{i,j}(n)\cdot\lambda_{i,j}
\end{aligned} \tag{8}$$

Subtracting $\underline{W}^T(n)T_{ocf}\underline{W}(n)$ from both sides:

$$\begin{aligned}
&\underline{\tilde{W}}^T(n+1)T_{ocf}\underline{\tilde{W}}(n+1) - \underline{W}^T(n)T_{ocf}\underline{W}(n) \\
&= 2\underline{W}^T(n)\underline{\lambda} - 2\underline{W}^T(n)[\underline{S}^*(n)\cdot\underline{\tau}(n)\cdot\underline{\lambda}] \\
&\quad + \sum_{i,j}\lambda_{i,j} - 2\sum_{i,j}S^*_{i,j}(n)\cdot\tau_{i,j}(n)\cdot\lambda_{i,j} + \sum_{i,j}S^*_{i,j}(n)\cdot\tau^2_{i,j}(n)\cdot\lambda_{i,j}
\end{aligned} \tag{9}$$

Then, taking the conditional expected value given $\underline{W}(n)$:

$$\begin{aligned}
&E\left[\underline{\tilde{W}}^T(n+1)T_{ocf}\underline{\tilde{W}}(n+1) - \underline{W}^T(n)T_{ocf}\underline{W}(n)\middle|\underline{W}(n)\right] \\
&= 2\left(\underline{W}^T(n)\underline{\lambda} - \underline{W}^T(n)\underline{S}^*(n)\right) + \sum_{i,j}\lambda_{i,j} - 2\sum_{i,j}S^*_{i,j}(n) + \sum_{i,j}\frac{S^*_{i,j}(n)}{\lambda_{i,j}}
\end{aligned} \tag{10}$$

After imposing the admissibility constraints and the scheduling algorithm properties, we obtain the following inequalities:

$$\sum_{i,j}\lambda_{i,j} \le N, \sum_{i,j} S^*_{i,j}(n) \ge 0, \sum_{i,j}\frac{S^*_{i,j}(n)}{\lambda_{i,j}} \le L < \infty, \tag{11}$$

where L is a non-negative constant.

From equation 10, we obtain:

$$\begin{aligned} &E\left[\underline{\tilde{W}}^T(n+1)T_{ocf}\underline{\tilde{W}}(n+1) - \underline{W}^T(n)T_{ocf}\underline{W}(n)\middle|\underline{W}(n)\right] \\ &\le 2\left(\underline{W}^T(n)\underline{\lambda} - \underline{W}^T(n)\underline{S}^*(n)\right) + L + N. \end{aligned} \tag{12}$$

Recall Lemma 1, $\underline{W}^T(n)\underline{\lambda} - \underline{W}^T(n)\underline{S}^*(n) \le 0$. Hence, we prove Lemma 2, where $K = L + N > 0$. ❐

Lemma 3: *Under the OCF algorithm,*
$E\left[\underline{\tilde{W}}^T(n+1)T_{ocf}\underline{\tilde{W}}(n+1) - \underline{W}^T(n)T_{ocf}\underline{W}(n)\middle|\underline{W}(n)\right] \le -\varepsilon\|\underline{W}(n)\| + K,$
$\forall\underline{\lambda} \le (1-\beta)\,\underline{\lambda}_m, 0 < \beta < 1$*, where* $\underline{\lambda}_m$ *is any rate vector such that* $\|\underline{\lambda}_m\|^2 = N$ *and* $\varepsilon > 0$.

Proof:

$$\underline{W}^T(n)\underline{\lambda} - \underline{W}^T(n)\underline{S}^*(n) \le \underline{W}^T(n)\,(1-\beta)\,\underline{\lambda}_m - \underline{W}^T(n)\underline{S}^*(n). \tag{13}$$

Applying Lemma 1,

$$\underline{W}^T(n)\underline{\lambda} - \underline{W}^T(n)\underline{S}^*(n) \le -\beta\underline{W}^T(n)\underline{\lambda}_m. \tag{14}$$

$$\underline{W}^T(n)\underline{\lambda} - \underline{W}^T(n)\underline{S}^*(n) \le -\beta\|\underline{W}(n)\| \cdot \|\underline{\lambda}_m\| \cdot \cos\theta, \tag{15}$$

where θ is the angle between $\underline{W}(n)$ and $\underline{\lambda}_m$.

Any non-zero waiting time, $W_{i,j}(n) > 0$, can occur iff the corresponding rate is non-zero, $\lambda_{i,j} > 0$. Hence, it is sufficient to say that $\cos\theta > 0$. Furthermore, since $\|\underline{\lambda}\| \le \sqrt{N}$,

$$\cos\theta = \frac{\underline{W}^T(n)\,\underline{\lambda}}{\|\underline{W}(n)\| \cdot \|\underline{\lambda}\|} \ge \frac{W_{max}(n)\,\lambda_{min}}{\|\underline{W}(n)\|\sqrt{N}}, \tag{16}$$

where $W_{max}(n) = max(W_{i,j}(n))$ and $\lambda_{min} = min(\lambda_{i,j})$.

Since $\|\underline{W}(n)\| \le NW_{max}$,

$$\cos\theta \ge \frac{\lambda_{min}}{N\sqrt{N}} \tag{17}$$

Using equations 12, 15, and 17.

$$\begin{aligned} &E\left[\underline{\tilde{W}}^T(n+1)T_{ocf}\underline{\tilde{W}}(n+1) - \underline{W}^T(n)T_{ocf}\underline{W}(n)\middle|\underline{W}(n)\right] \\ &\le -2\beta\frac{\lambda_{min}}{N\sqrt{N}}\|\underline{W}(n)\| + K, \end{aligned} \tag{18}$$

where $\varepsilon = 2\beta\dfrac{\lambda_{min}}{N\sqrt{N}}$. ❐

Lemma 4: *Under the OCF algorithm,*

$$E\left[\underline{W}^T(n+1)T_{ocf}\underline{W}(n+1) - \underline{W}^T(n)T_{ocf}\underline{W}(n)\middle|\underline{W}(n)\right] \le -\varepsilon\|\underline{W}(n)\| + K,$$

$\forall\underline{\lambda} \le (1-\beta)\,\underline{\lambda}_m$, $0 < \beta < 1$, *where* $\underline{\lambda}_m$ *is any rate vector such that* $\|\underline{\lambda}_m\|^2 = N$.

Proof: We can draw the following relationship between the two waiting times:

$$W_{i,j}(n+1) = \begin{cases} \tilde{W}_{i,j}(n+1), & \tilde{W}_{i,j}(n+1) \ge 0 \\ 0, & \tilde{W}_{i,j}(n+1) < 0 \end{cases}. \tag{19}$$

Since T_{ocf} is a positive definite matrix, equation 19 implies:

$$\underline{W}^T(n+1)T_{ocf}\underline{W}(n+1) \le \underline{\tilde{W}}^T(n+1)T_{ocf}\underline{\tilde{W}}(n+1),\ \forall n\,. \tag{20}$$

Hence,

$$\begin{aligned} &E\left[\underline{W}^T(n+1)T_{ocf}\underline{W}(n+1) - \underline{W}^T(n)T_{ocf}\underline{W}(n)\middle|\underline{W}(n)\right] \\ &\le E\left[\underline{\tilde{W}}^T(n+1)T_{ocf}\underline{\tilde{W}}(n+1) - \underline{W}^T(n)T_{ocf}\underline{W}(n)\middle|\underline{W}(n)\right] \end{aligned}. \tag{21}$$

This proves Lemma 4. ❐

Lemma 5: *Under the OCF algorithm, there exists a quadratic Lyapunov function, $V(\underline{W}(n))$ such that:*

$$E\,[\,V(\underline{W}(n+1)) - V(\underline{W}(n))\,|\,\underline{W}(n)]\ \le -\varepsilon\|\underline{W}(n)\| + K\,, \tag{22}$$

where K is a constant and $\varepsilon > 0$.

Proof: From Lemma 4, $V(\underline{W}(n)) = \underline{W}^T(n)T_{ocf}\underline{W}(n)$, $\varepsilon = 2\beta\dfrac{\lambda_{min}}{N\sqrt{N}} > 0$ and $K = L + N > 0$. ❐

Theorem 1.2: *Under the OCF algorithm, the waiting times are stable for all admissible and independent arrival processes, i.e., $E\,[\|\underline{W}(n)\|] \le C < \infty$.*

Proof: Since there exists a quadratic Lyapunov function $V(\underline{W}(n))$ that satisfies equation 22, according to Kumar [32][38], the sum of all waiting times is stable-in-the-mean, i.e.,

$$\frac{1}{N+1}\sum_{n=0}^{N}\sum_{i,j}E\,[\,W_{i,j}(n)] < \infty,\ \forall N \tag{23}$$

Furthermore, since the arrivals are independent, the waiting vector forms a Markov chain, and equation 23 guarantees that the chain is positive recurrence [32][38]. Hence,

$$E[\|\underline{W}(n)\|] \le C < \infty . \qquad (24)$$

Now, we are ready to prove the main theorem.

Proof of the Main Theorem: From Fact 3, $W_{i,j}(n) \ge L_{i,j}(n), \forall i, j, n$. Thus,

$$E[\|\underline{L}(n)\|] \le E[\|\underline{W}(n)\|] \le C < \infty . \qquad (25)$$

Hence, $E[\|\underline{L}(n)\|]$ is also bounded above by C. ❐

APPENDIX 2

Stability of NxN Switch under LPF with Independent Arrivals

1 Definitions

In addition to the definitions defined in Chapter 1, the proofs in this appendix also require the following definitions:

1. A positive-definite and symmetric transformation matrix, T_{lpf}.

   For an $N \times N$ switch, T_{lpf} is an $N^2 \times N^2$ matrix whose elements are defined as follows:

$$T_{lpf(i,j)} = \begin{cases} 2, & i = j \\ 1, & \left\lfloor \frac{i}{N} \right\rfloor = \left\lfloor \frac{j}{N} \right\rfloor \\ 1, & i \bmod N = j \bmod N \\ 0, & \text{otherwise.} \end{cases}$$

   For instance, for a 2×2 switch, T_{lpf} is

$$T_{lpf} = \begin{bmatrix} 2 & 1 & 1 & 0 \\ 1 & 2 & 0 & 1 \\ 1 & 0 & 2 & 1 \\ 0 & 1 & 1 & 2 \end{bmatrix}.$$

2 Stability without the Pipeline Delay

2.1 Main Theorem

Theorem 2.1: *Under the LPF algorithm, the queue occupancies are stable for all admissible and independent arrival processes, i.e.,* $E[\|\underline{L}(n)\|] \le C < \infty$.

2.2 Proof

Consider a quadratic Lyapunov function $V(\underline{L}(n)) = \underline{L}(n) T_{lpf} \underline{L}(n)$, the next state occupancy vector:[1]

$$\underline{L}(n+1) \equiv \underline{L}(n) - \underline{S}(n) + \underline{A}(n) \tag{1}$$

and an LPF request vector, $\underline{L}'(n)$, whose each element is a function of queue occupancies defined as:

$$L'_{i,j}(n) = \begin{cases} R_i + C_j, & L_{i,j}(n) > 0 \\ 0, & \text{otherwise,} \end{cases} \tag{2}$$

where $R_i = \sum_{j}^{N} L_{i,j}(n)$ and $C_j = \sum_{i}^{N} L_{i,j}(n)$.

Unlike the proofs in Appendix 1, we can directly consider the actual next state occupancy vector because Fact 1 below guarantees that the vector contains no negative element. As shown by Figure A2.1 that LPF considers current arrivals when making a scheduling decision, it is trivial to verify the following fact, which is critical for the proofs that follow.

1. See [49] for detailed definition.

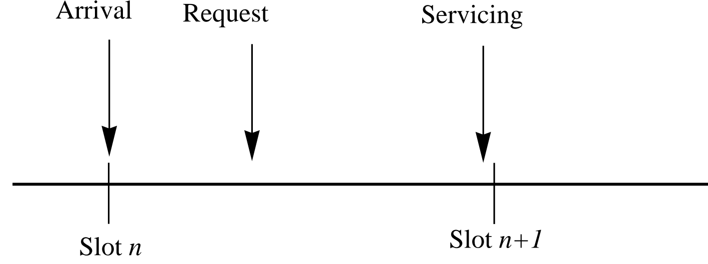


Figure A2.1 A time line showing events at a VOQ. During every slot, an arrival at the queue, if any, takes place at the beginning of the slot. Shortly after the beginning of the slot, every non-empty queue makes a request. LPF then takes all requests into consideration and arrives at its scheduling decision before the end of the slot. Each selected queue is then serviced before a new arrival occurs.

Fact 1 A queue with zero occupancy and no arrival cannot make a request. Hence, a queue which makes a request must have either non-zero occupancy, an arrival or both.

Lemma 1: *Under the LPF algorithm,* $E[\underline{L}'^T(n)(\underline{A}(n) - \underline{S}^*(n)) \,|\, \underline{L}(n)] \le 0$ *for* $\sum_{i=1}^{N} \lambda_{i,j} \le 1$, $\sum_{j=1}^{N} \lambda_{i,j} \le 1$, $\lambda_{i,j} \ge 0$*, where* $\underline{S}^*(n)$ *is an LPF service matrix such that* $\underline{L}'^T(n)\underline{S}^*(n) = \max(\underline{L}'^T(n)\underline{S}(n))$.

Proof: Consider the following two cases. The first case is when the match size is N, $|\underline{S}^*(n)| = N$.

For this case, Property 1 in Chapter 3 implies that

$$\underline{L}'^T(n)\underline{S}^*(n) = 2\sum_{i,j} L_{i,j}(n) . \tag{3}$$

This is because all inputs and all outputs are selected (matched). Also, for the match size of N,

$$E[\underline{L}'^T(n)\underline{A}(n) \mid \underline{L}(n), |\underline{S}^*(n)| = N] = \underline{L}^T(n) T_{lpf}\underline{\lambda} . \tag{4}$$

For the admissibility constraint: $\sum_{i=1}^{N} \lambda_{i,j} \le 1, \sum_{j=1}^{N} \lambda_{i,j} \le 1, \lambda_{i,j} \ge 0$,

$$T_{lpf}\underline{\lambda} \le \underline{2} . \tag{5}$$

Substituting equation 5 into equation 4, we obtain the following inequality,

$$E[\underline{L}'^T(n)\underline{A}(n) \mid \underline{L}(n), |\underline{S}^*(n)| = N] \le 2\sum_{i,j} L_{i,j}(n) . \tag{6}$$

By equations 3 and 6, we prove the first case.

$$E[\underline{L}'^T(n)(\underline{A}(n) - \underline{S}^*(n)) \mid \underline{L}(n), |\underline{S}^*(n)| = N] \le 0 . \tag{7}$$

For the second case when the match size is $k < N$, we use Konig-Egervary's theorem, which states that *the size of a maximum size match is equal to the minimum number of rows plus columns which can contain all requests*[1] [36][45]. In this case, rows and columns are inputs and outputs. Consider any non-weighted request matrix, $R(n)$, from which LPF finds a match of size k. Consequently, as a result of Theorem 3.1 (which says that a match found by the LPF is also a maximum size match), there exists a minimum set of k rows or columns or rows and columns combined which contains all requests in the matrix.

These rows and/or columns may not necessarily be consecutive. In order for the ease of visualizing the proofs, we can rearrange the request matrix so that the columns are con-

1. "Containing," in this context, means that all requests belong to the interested set of either rows or columns.

secutively moved to the left of the matrix, and so that the rows are also consecutively moved to the top of the matrix. By permuting the rows and the columns, we obtain such a matrix, $R'(n)$, whose first l rows and $k-l$ columns belong to the minimum set and therefore contain all requests, where $0 \le l \le k$, as shown in Figure A2.2. The shaded area, which is not covered by the first l rows or $k-l$ columns, contains no request.

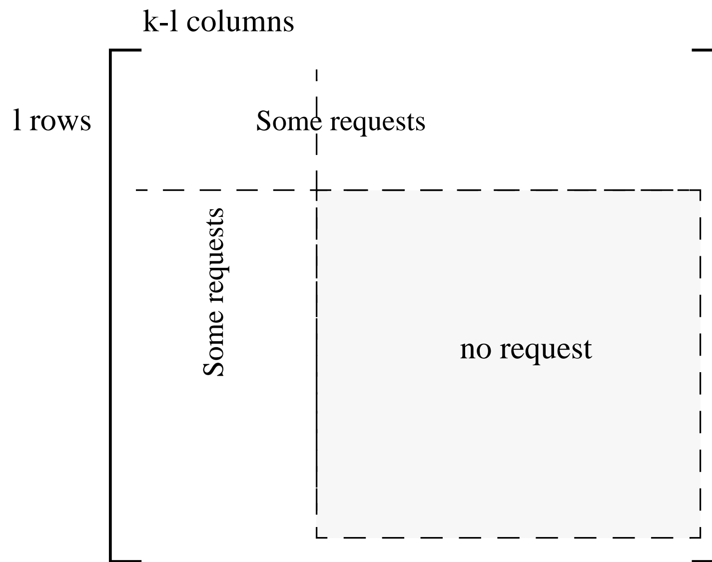


Figure A2.2 A rearranged request matrix. From the original request matrix, row permutation was performed to move all l rows of the minimum set to the top, and column permutation was performed to move all $k-l$ columns of the same set to the left. Together these rows and columns contain all requests, leaving the shaded area of the new request matrix with no request.

For a given set I containing the original indices of the first l rows and a given set J containing the original indices of the first $k-l$ columns, let:

$$\tilde{\underline{\lambda}} \equiv E[\underline{A}(n) | I, J] \tag{8}$$

be a conditional arriving rate vector. Since the shaded area has no arrival,

$$\tilde{\lambda}_{i,j} = \begin{cases} \lambda_{i,j}, i \in I, i \in J \\ 0, \text{otherwise.} \end{cases} \tag{9}$$

With the traffic admissibility constraint: $\sum_{i=1}^{N} \lambda_{i,j} \le 1, \sum_{j=1}^{N} \lambda_{i,j} \le 1, \lambda_{i,j} \ge 0$,

$$\sum_{i,j} \tilde{\lambda}_{i,j} \le k. \tag{10}$$

Now, consider a linear programming problem:

$$\begin{aligned} &\max(\underline{L}'^{T}(n)\tilde{\underline{\lambda}}) \\ &\text{s.t.} \quad \sum_{i=1}^{N} \tilde{\lambda}_{i,j} \le 1, \sum_{j=1}^{N} \tilde{\lambda}_{i,j} \le 1, \tilde{\lambda}_{i,j} \ge 0 \\ &\sum_{i,j} \tilde{\lambda}_{i,j} \le k. \end{aligned} \tag{11}$$

Because Λ is a doubly stochastic matrix and forms a convex set with a set of $S(n)$ that satisfies the following constraint as its set of extreme points [49],

$$\begin{aligned} &\text{s.t.} \quad \sum_{i=1}^{N} S_{i,j}(n) = 1, \sum_{j=1}^{N} S_{i,j}(n) = 1, S_{i,j}(n) = 0, 1 \\ &\sum_{i,j} S_{i,j}(n) = k \end{aligned} \tag{12}$$

hence,

$$E[\underline{L}'^{T}(n)\tilde{\underline{\lambda}}] \le max[\underline{L}'^{T}(n)\underline{S}(n)], \tag{13}$$

$$E[\underline{L}'^{T}(n)\underline{A}(n)|I, J] \le max[\underline{L}'^{T}(n)\underline{S}(n)]. \tag{14}$$

Since the above is true for all I and J satisfying $|I| + |J| = k$, according to Konig-Egervary's theorem, it follows that:

$$E\,[\underline{L}'^T(n)\,(\underline{A}(n) - \underline{S}^*(n))\,\big|\underline{L}(n), \big|\underline{S}^*(n)\big| = k] \le 0\,, \tag{15}$$

where $\underline{S}^*(n)$ is such that $\underline{L}'^T(n)\underline{S}^*(n) = max\,[\underline{L}'^T(n)\underline{S}(n)]$. This proves the second case when $\big|\underline{S}^*(n)\big| < N$.

From both cases, it can be concluded that:

$$E\,[\underline{L}'^T(n)\,(\underline{A}(n) - \underline{S}^*(n))\,\big|\underline{L}(n)] \le 0\,. \;\square \tag{16}$$

Note: whenever $S_{i,j}(n)$ cannot be one because there is no request, Fact 1 implies that $A_{i,j}(n)$ must be zero, and hence, $\tilde{\lambda}_{i,j} = 0$.

Lemma 2: *Under the LPF algorithm,* $E\,[\underline{L}'^T(n)\,(\underline{A}(n) - \underline{S}^*(n))\,\big|\underline{L}(n)] \le -2\beta\sum_{i,j} L_{i,j}(n)$ $\forall\underline{\lambda} \le (1-\beta)\,\underline{\lambda}_m,\ 0 < \beta < 1$ *,where* $\underline{\lambda}_m$ *is any admissible rate vector such that* $\sum_{i,j}\underline{\lambda}_{m_{i,j}} = N$.

Proof: Following the same steps as done in the proof of Lemma 1 and using the fact that $T\underline{\lambda}_m = \underline{2}$, it is can be shown that for every $\big|\underline{S}^*(n)\big| = k$:

$$E\,[\underline{L}'^T(n)\,(\underline{A}(n) - \underline{S}^*(n))\,\big|\underline{L}(n), \big|\underline{S}^*(n)\big| = k] \le -2\beta\sum_{i,j} L_{i,j}(n)\,. \;\square \tag{17}$$

Lemma 3: *Under the LPF algorithm,*

$$\mathrm{E}\Big[\underline{L}^T(n+1)T_{lpf}\underline{L}(n+1) - \underline{L}^T(n)T_{lpf}\underline{L}(n)\Big|\ \underline{L}(n)\Big] \le -\varepsilon\sum_{i,j} L_{i,j}(n) + N^2$$

$\forall\underline{\lambda} \le (1-\beta)\,\underline{\lambda}_m,\ 0 < \beta < 1$, *where* $\underline{\lambda}_m$ *is any rate vector such that* $\sum_{i,j}\underline{\lambda}_{m_{i,j}} = N$ *and* $\varepsilon > 0$.

Proof: Since the LPF algorithm always finds a maximum weight match, $\underline{S}^*(n)$, using equation 1, we obtain the following equality by expansion.

$$\underline{L}^T(n+1)T_{lpf}\underline{L}(n+1) - \underline{L}^T(n)T_{lpf}\underline{L}(n)$$
$$= 2\underline{L}^T(n)T_{lpf}(\underline{A}(n) - \underline{S}^*(n)) + (\underline{S}^*(n) - \underline{A}(n))^T T_{lpf}(\underline{S}^*(n) - \underline{A}(n)) . \tag{18}$$

Since $|\underline{S}^*(n) - \underline{A}(n)| \le N$, $(\underline{S}^*(n) - \underline{A}(n))^T T_{lpf}(\underline{S}^*(n) - \underline{A}(n)) \le 2N^2$. Using Lemma 2,

$$\mathrm{E}\left[\underline{L}^T(n+1)T_{lpf}\underline{L}(n+1) - \underline{L}^T(n)T_{lpf}\underline{L}(n) \middle| \underline{L}(n)\right] \le -\varepsilon\sum_{i,j} L_{i,j}(n) + 2N^2 \text{, where}$$

$\varepsilon = 4\beta$. ❒ (19)

Now, we are ready to prove the main theorem.

Proof of Main Theorem:

There exists a *quadratic Lyapunov function,* $V(\underline{L}(n)) = \underline{L}(n)T_{lpf}\underline{L}(n)$, such that:

$$\mathrm{E}\,[V(\underline{L}(n+1)) - V(\underline{L}(n)) |\, \underline{L}(n)] \le -\varepsilon\sum_{i,j} L_{i,j}(n) + 2N^2 . \tag{20}$$

According to Kumar [32][38], the sum of all queue occupancies is stable-in-the-mean, i.e.,

$$\frac{1}{N+1}\sum_{n=0}^{N}\sum_{i,j} E\,[L_{i,j}(n)] < \infty, \forall N . \tag{21}$$

Furthermore, if the arrivals are independent, the queue occupancy vector forms a Markov chain, which equation 21 guarantees to be positive recurrence. Hence,

$$E\,[\|\underline{L}(n)\|] \le C < \infty . \tag{22}$$

3 Stability with the Pipeline Delay

3.1 Main Theorem

Theorem 2.2: *Using k slot old weights, the LPF algorithm is stable for all admissible independent arrival processes,* $0 < k < \infty$*, i.e.,* $E[\|\underline{L}(n)\|] \le C < \infty$

3.2 Proof

The proof requires the subsequent lemma and is given after the proof of the lemma. The following lemma states the existence of an upper bound on the difference between the total weight of the optimum match found by non-pipelined LPF and the total weight of the actual match found by pipelined LPF.

Lemma 4: $\underline{L}'(n)\underline{S}^*(n) - \underline{L}'(n)\tilde{\underline{S}}(n) \le \left(N^2 + 3N\right)k$ *where* $\underline{S}^*(n)$ *is the optimum service vector if LPF had been given the correct weights, and* $\tilde{\underline{S}}(n)$ *is the actual service vector which optimizes on the incorrect weight.*

Proof: Let $\tilde{\underline{L}}'(n)$ be the incorrect weight vector given to LPF as a result of the pipeline delay in the sorters, and by definition:

$$\tilde{\underline{L}}'(n) = \underline{L}'(n-k) \ . \tag{23}$$

Furthermore, we can establish an upper bound and a lower bound of $\tilde{\underline{L}}'(n)$ with respect to the correct vector as follows. Consider the fact that there can be up to k arrivals at any input and Nk arrivals for any output, and that there can be no departure from any input or for any output during a period of k slots. This fact implies that a correct LPF weight can increase by at most $(N+1)k$ from its previous k slot value:

$$\underline{L}'(n) \le \underline{L}'(n-k) + \underline{(N+1)k} \ . \tag{24}$$

From the above equation and equation 23, a lower bound of $\tilde{L}'(n)$ is:

$$\underline{L}'(n) - \underline{(N+1)\,k} \le \tilde{\underline{L}}'(n). \tag{25}$$

For an upper bound, consider the case when there is no arrival but one departure from every input and to every output during every slot in a period of k slots. An upper bound of $\tilde{\underline{L}}'(n)$ with respect to $\underline{L}'(n)$ is therefore:

$$\underline{L}'(n) \ge \underline{L}'(n-k) - \underline{2k}, \tag{26}$$

$$\tilde{\underline{L}}'(n) \le \underline{L}'(n) + 2k. \tag{27}$$

With the two bounds, we are now ready to derive an upper bound on the difference between the two total weights. For the following derivation, it is worth noting that $\underline{S}^*(n)$ maximizes the total weight on $\underline{L}'(n)$ while $\tilde{\underline{S}}(n)$ maximizes the total weight on $\tilde{\underline{L}}'(n)$, and that the numbers of connections made by $\underline{S}^*(n)$ and $\tilde{\underline{S}}(n)$ are equal, i.e., $|\tilde{\underline{S}}(n)| = |\underline{S}^*(n)|$.

Using equation 27, consider $\tilde{\underline{S}}(n)$.

$$\underline{L}'(n)\tilde{\underline{S}}(n) \ge \tilde{\underline{L}}'(n)\tilde{\underline{S}}(n) - \underline{2k}\tilde{\underline{S}}(n). \tag{28}$$

But,

$$\underline{2k}\tilde{\underline{S}}(n) \le 2kN. \tag{29}$$

Hence,

$$\underline{L}'(n)\tilde{\underline{S}}(n) \ge \tilde{\underline{L}}'(n)\tilde{\underline{S}}(n) - 2kN. \tag{30}$$

Now consider $\underline{S}^*(n)$ using equation 25.

$$\underline{L}'(n)\underline{S}^*(n) - \underline{(N+1)\,k}\underline{S}^*(n) \le \tilde{\underline{L}}'(n)\underline{S}^*(n). \tag{31}$$

Since $\underline{(N+1)\,k}\underline{S}^*(n) \le (N+1)\,kN$,

$$L'(n)\underline{S}^*(n) - (N+1)\,kN \le \tilde{\underline{L}}'(n)\underline{S}^*(n)\ . \tag{32}$$

Furthermore, $\tilde{\underline{L}}'(n)\underline{S}^*(n) \le \tilde{\underline{L}}'(n)\tilde{\underline{S}}(n)$ because $\tilde{\underline{S}}(n)$ is the optimum match with respect to $\tilde{\underline{L}}'(n)$. Thus,

$$L'(n)\underline{S}^*(n) - (N+1)\,kN \le \tilde{\underline{L}}'(n)\tilde{\underline{S}}(n)\ . \tag{33}$$

Substituting equation 33 back into equation 30 yields:

$$L'(n)\tilde{\underline{S}}(n) \ge L'(n)\underline{S}^*(n) - (N+1)\,kN - 2kN\,. \tag{34}$$

Finally, an upper bound on the difference between the two total weights is:

$$L'(n)\underline{S}^*(n) - L'(n)\tilde{\underline{S}}(n) \le \left(N^2 + 3N\right)k\ .\ \square \tag{35}$$

With the lemma proved, we can now prove the main theorem.

Proof of Main Theorem:

Similarly to the proof of Theorem 2.1, by taking the same steps as done in Lemma 1, Lemma 2 and Lemma 3 and by using the relationship stated by Lemma 4, we can show that there exists a *quadratic Lyapunov function,* $V(\underline{L}(n)) = \underline{L}(n)T_{lpf}\underline{L}(n)$, such that:

$$\mathrm{E}\,[V(\underline{L}(n+1)) - V(\underline{L}(n))|\ \underline{L}(n)] \le -\varepsilon\sum_{i,j} L_{i,j}(n) + 2N^2 + \left(N^2 + 3N\right)k\,. \tag{36}$$

And likewise, according to Kumar [32][38], the sum of all queue occupancies is stable-in-the-mean, i.e.,

$$\frac{1}{N+1}\sum_{n=0}^{N}\sum_{i,j} E\,[L_{i,j}(n)] < \infty,\ \forall N\,. \tag{37}$$

Furthermore, if the arrivals are independent, the queue occupancy vector forms a Markov chain, which equation 37 guarantees to be positive recurrence. Hence,

$$E[\|\underline{L}(n)\|] \le C' < \infty . \tag{38}$$

APPENDIX 3

LPF Theorems

1 LPF Match is a Maximum Size Match

Theorem 3.1: *The maximum weight match found by LPF is also a maximum size match.*

Proof:

We prove the theorem by contradiction. Let M be a maximum weight match but not a maximum size match found by LPF; that is, there exists another larger size match M' that can be found by any maximum size matching algorithm.

Consider the Ford-Fulkerson method [13][67]. With this method, if M was not a maximum size match, a larger size match M' could be found by augmenting the flow produced by M on the corresponding flow network similar to the one shown in Figure 3.4. Since this augmentation process does not remove any matched input or output of the previous flow (match) [13][67], M' would contain all matched inputs and outputs in M plus some newly matched inputs and outputs. As a result of the set of inputs and outputs in M being a subset of that in M', Property 1 in Chapter 3 implies that, had it existed, M' would be a

larger weight match than M. Hence, M could not be the match found by LPF because it was not a maximum weight match. This contradicts the assumption above. Therefore, M' does not exist, and an LPF match must be both a maximum size and maximum weight match. ❐

2 Modified Edmonds-Karp Algorithm is Equivalent to LPF

Theorem 3.2: *For the request weighting defined in equation 3.1, there exists a maximum size matching algorithm which can find a match that is of both maximum size and maximum weight.*

Proof: proved by Theorem 3.3. ❐

Theorem 3.3: *A match found by the modified Edmonds-Karp algorithm is a maximum weight match whose weights are as defined in equation 3.1.*

Proof: Before we prove the theorem, we first prove the following lemma needed for proving the theorem.

Let M be a match found by the modified Edmonds-Karp algorithm, F be the associated flow of M and R be a residual graph produced by F. Based on the principle that a maximum weight match can be found by solving for a flow of a minimum cost, the strategy for the proof is therefore to show that F is a minimum cost flow on the associated flow network.[1] To show that, we need the following theorem [67].

Theorem 3.4: *A flow F is minimum cost if and only if its residual graph R has no negative cost cycle.*

1. A transformation from a request graph to a flow network is outlined in Section 4.1 in Chapter 3.

Based on this theorem, the following proves that F is a minimum cost flow by showing that R contains no negative cycle. As an example, Figure A3.1 shows a residual graph

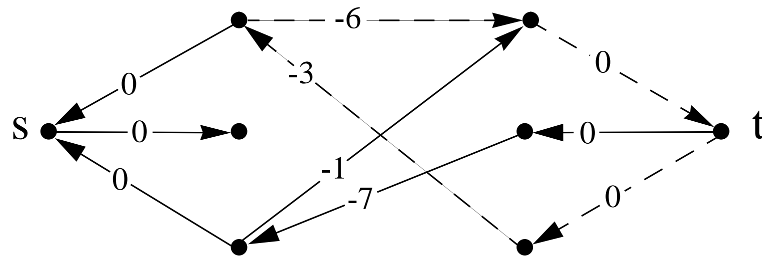


Figure A3.1 A residual network containing a negative cost cycle shown by the dotted edges.

of a match which is of only maximum size, not a maximum weight, because the residual graph contains one negative cost cycle (shown by the dashed lines). In principle, the cycle in this figure indicates that the total cost of the flow can be lowered by replacing the matched edges (input and output pairs) in the cycle with the unmatched edges [67]. Before we proceed with the proof, it is necessary to state the following properties, which are required for the proof and can be easily verified.

Property 1 Flow augmentation[1] does not remove any input or output from the matched sets but adds one or more new inputs and outputs to the sets. Once matched, every input or output remains matched throughout the augmentation process.

1. The Ford-Fulkerson method.

Property 2 In R, an input is never adjacent to another input, and an output is never adjacent to another output; i.e., no edge exists between any two inputs or between any two outputs.

Property 3 F cannot be increased since it is already a maximum flow (M is a maximum size match).

Property 4 In R, all edges from unmatched inputs are to only matched outputs; otherwise, F is not a maximum flow (an additional flow can be trivially added).

Property 5 In R, all edges to unmatched outputs must be from matched inputs; otherwise, F is not a maximum flow (an additional flow can be trivially added).

We are now ready to begin the proof by considering the following three cases in which a negative cost cycle can occur. Shown in Figure A3.3, Figure A3.4 and Figure A3.5 (on page 150 and page 151) are residual graphs containing three types of negative cost cycles: an input cycle, an output cycle and a compound cycle, respectively. These residual graphs are constructed from the corresponding flow network by reversing the directions of all matched edges including the ones connecting s to matched inputs and t with matched outputs as described in Chapter 3. All matched edges are represented by the shaded lines, and all unmatched edges are represented by the solid lines. In these graphs, the inputs are rearranged and divided into two non-overlapping sets: matched and unmatched, and similarly, the outputs are also rearranged and divided. The set of matched inputs and the set of matched outputs are shown by the shaded dots and are connected by two sets of edges: matched and unmatched. These sets are shown by the thicker lines.

Through these sets of edges, there exists at least one path which begins at a matched output and ends at a matched input connecting a subset of matched inputs and matched

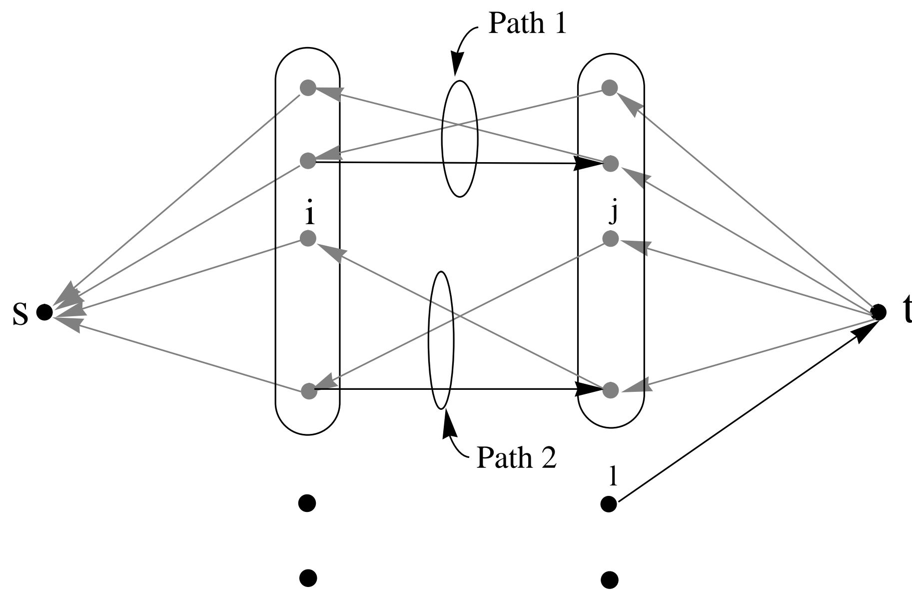


Figure A3.2 A residual graph showing two reverse flow paths between two pairs of inputs and outputs.

outputs. Each of the paths is formed by matched and unmatched edges lying alternately along the path.[1] On such paths, the inputs and outputs also lie alternately [67]. Moreover, every matched input and matched output must belong to one and only one path [67]. Figure A3.2 shows examples of such paths. We shall refer to these paths as *reverse flow* paths.

When considering the three cases, we assume that there exists an unmatched edge from some matched input i to some unmatched output l, and that there exists an unmatched edge from some unmatched input k to some matched output j. However, according to Property 4 and 5, we cannot assume the existence of any edge connecting an unmatched input to an unmatched output in these graphs.

1. See alternate path theory in [67].

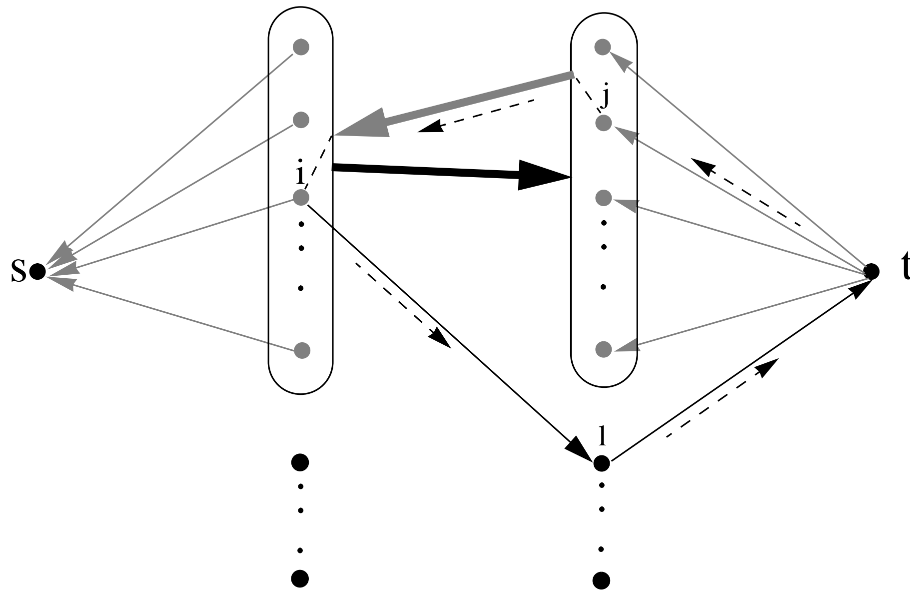


Figure A3.3 A residual graph showing an output cycle as indicated by the dotted lines.

Case 1: Output cycle: Shown in Figure A3.3, the cycle starts from t to some matched output j, goes through a reverse flow path connecting j to some input i, and finally returns from i to t without visiting s via some unmatched output l. When the occupancy of l is greater than that of j, property 3.1 infers that the cycle has a negative cost.

Case 2: Input cycle: Shown in Figure A3.4, the cycle begins at s and goes to some unmatched input k that connects to some matched output j. From j the cycle reaches matched input i from where it can return to s. Likewise, property 3.1 infers that the cycle is a negative cost cycle if the occupancy of i less than that of k.

Case 3: Compound cycle: Unlike an input or an output cycle, a compound cycle visits both s and t. As shown in Figure A3.5, the cycle starts from s and goes to some unmatched input k which has a request for some matched output j. The request leads the cycle to j. From j to some matched input i, there exists a reverse flow path which leads

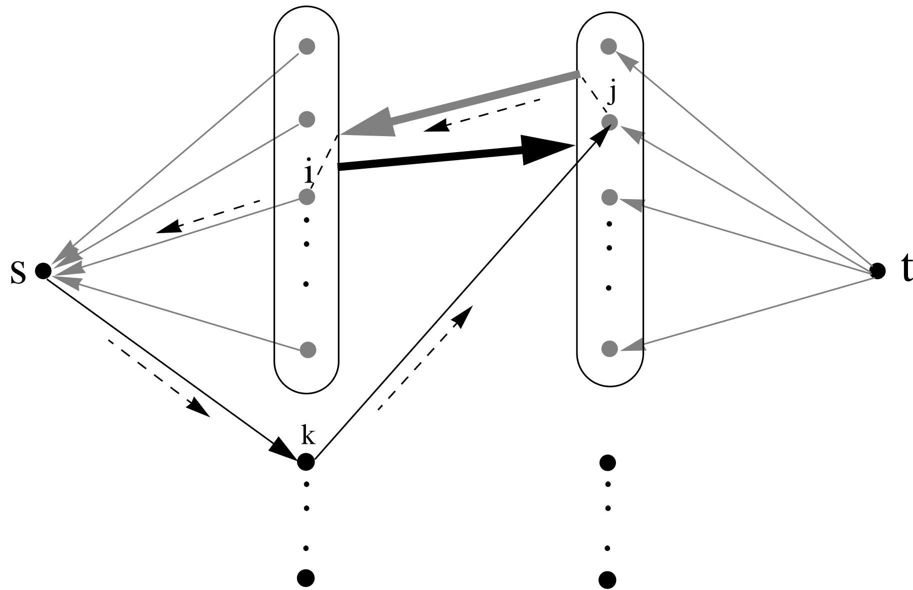


Figure A3.4 A residual graph showing an input cycle as indicated by the dotted lines.

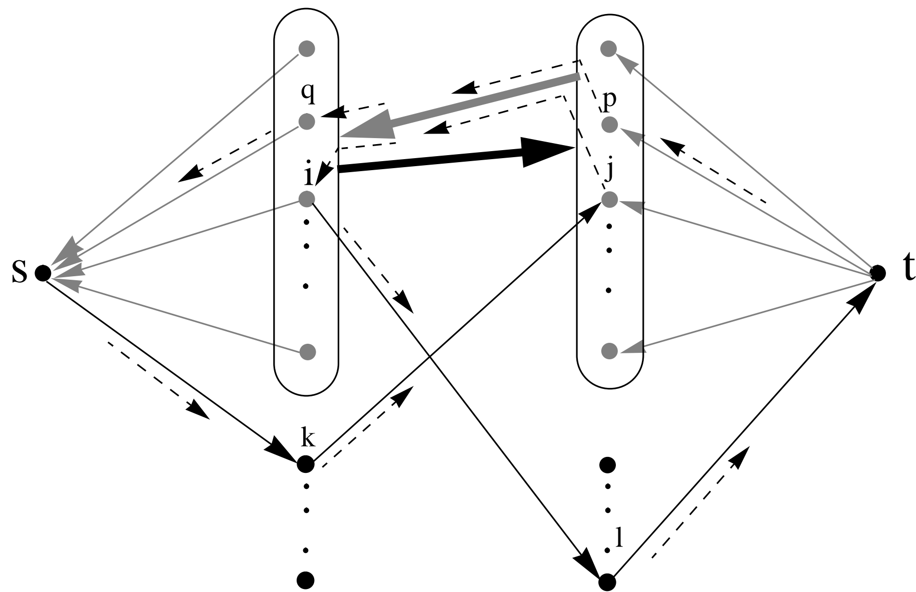


Figure A3.5 A residual graph showing a compound cycle as indicated by the dotted lines.

the cycle to i. From i, the cycle reaches t via some unmatched output l. The cycle then return to s by first visiting some matched output p that is connected to another matched input q by some reverse flow path. From q, the cycle reaches s where it began. According to property 3.1, the cycle has a negative cost if the combined occupancy of input i and output j is less than the combined occupancy of input k and output l.

With the above definitions of negative cycles, we are ready to prove that no negative cycle exists in a residual graph of a match found by the modified algorithm.

Lemma 1: *No negative cost output cycle exists in* R.

Proof:

We prove this Lemma by contradiction. First, we assume that such a cycle, as shown in Figure A3.6, exists. This implies that somehow a reverse flow path P_o exists in R from some output j to input i that directly leads to output l by an unmatched edge. Of all outputs on the path, j must be the smallest; otherwise, we can form a more negative cost cycle by replacing j with a smaller output on the path.

To show that such a path cannot possibly exist, consider the formation of P_o in Figure A3.6 when the modified algorithm matches input i' to j. When it was the turn of input i' to be matched, the modified algorithm performed LPFS to find the largest unmatched output for i'. For j to be appended to P_o, i' must have at least two requests: one to j which is subsequently matched and another one to some output on P_o which is later never unmatched. Hence, there would have been at least two augmenting paths through i', leading i' to more than one unmatched output: (a) j, (b) some output on P_o or l that has not been matched then. Since j is smaller than l and all outputs on the path, this creates a contradiction because LPFS would have chosen to match i' not with j but with the largest unmatched outputs on the path or l then.

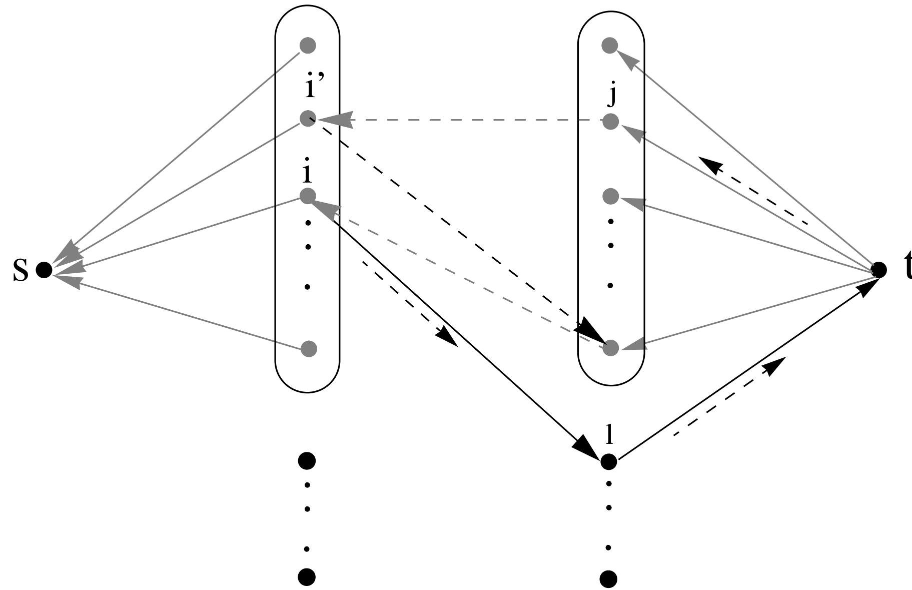


Figure A3.6 A residual graph for the proof of Lemma 1, depicting an assumed reverse flow path P_o from j to i by the dotted edges.

Given the fact that j is matched to i', P_o and hence the cycle do not exist. ❐

Lemma 2: *No negative cost input cycle exists in* R.

Proof:

We prove this lemma by employing the same strategy as that used for the previous lemma. We first assume that the cycle exists and then contradict the assumption by using the property of LPFS. As shown in Figure A3.7, the assumed cycle visits some matched input i, some matched output j and unmatched input k. Based on LPF request weighting, the occupancy of k must be greater than that of i for the cycle to be of a negative cost. But most important of all, in order for the cycle to exist, there must be a reverse flow path, P_i, from j to i in R.

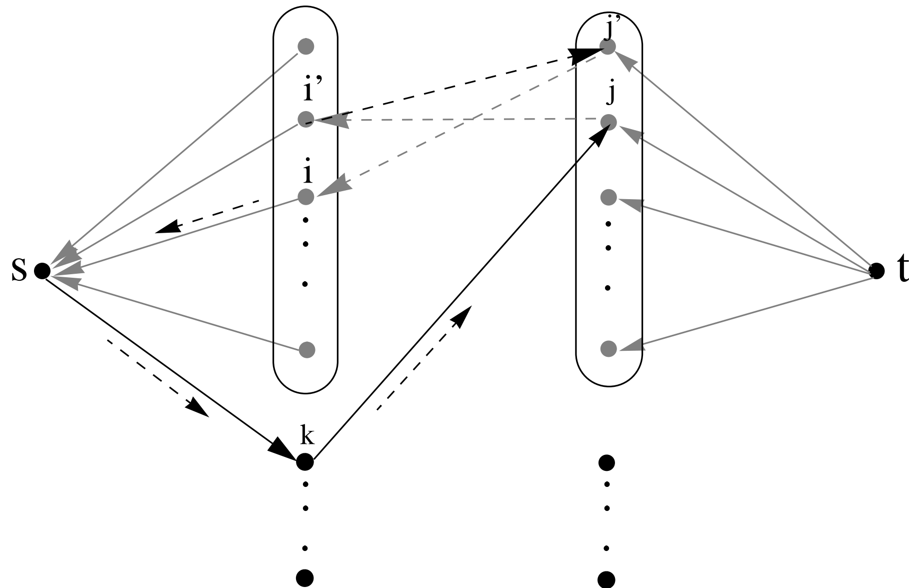


Figure A3.7 A residual graph for the proof of Lemma 2, depicting an assumed reverse flow path P_i from to by the dotted edges.

The following, however, contradicts the existence of the path. Consider that the modified algorithm would attempt to match k before i because k is larger than i. With the existence of the path, there would be two possibilities to match k. One possibility was that k discovered that output j had not been matched at that time, and thus k would have been matched to j. This possibility contradicts the fact that k is never matched.

The other possibility is that j is already matched to i', which must be larger than k when it is the turn of k to be matched. Consequently, the matching of j appends j to P_i. However, from j, P_i would have led k some unmatched output on P_i which had not been matched at that time. In the worst case, P_i would have led k to i' and consequently to output j', which is supposed to be matched to i but remained unmatched at that time since it was not yet the turn of i to be matched. In any case, had P_i existed, k would be matched.

Given the fact that k is unmatched, P_i and hence the cycle do not exist. ❐

Lemma 3: *No negative cost compound cycle exists in R.*

Proof:

As defined and as illustrated in Figure A3.8, in order for such a cycle to exist, there must be a path from s to t via k and l. However, this path is exactly an augmenting path on which more flow can be carried. The existence of the path contradicts Property 3, which says that the flow is already a maximum flow. Therefore, such a path from s to t and hence a negative cost compound cycle cannot exist. ❐

Lemma 4: *No negative cost cycle of any kind exists in R.*

Proof: By Lemma 1, Lemma 2 and Lemma 3. ❐

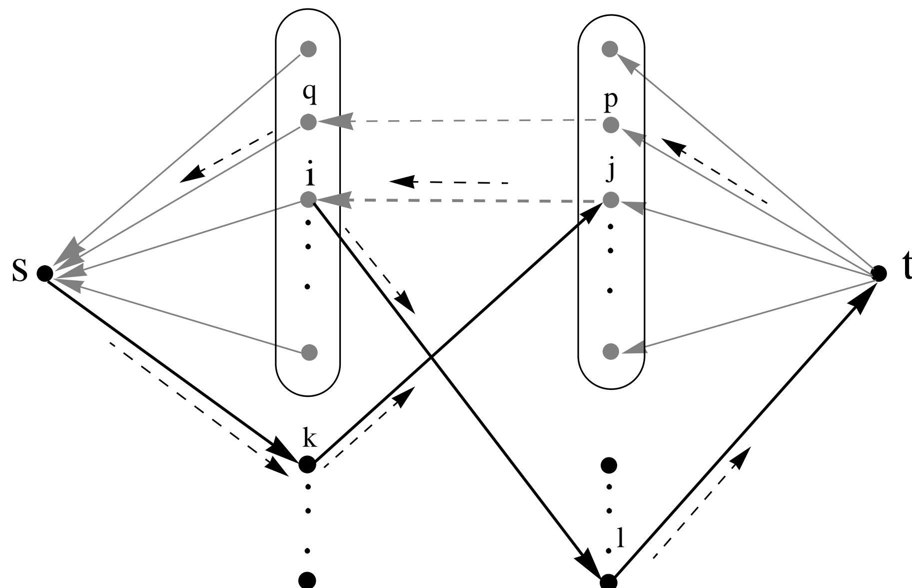


Figure A3.8 A residual graph for the proof of Lemma 3, showing a forward flow path from s to t which can increase the total flow from s to t.

With Lemma 4, we can prove Theorem 3.3. According to Theorem 3.4, F must be a minimum cost flow because, as proved by Lemma 4, its residual graph contains no negative cost cycle. Therefore, M is a maximum weight match. ❒

APPENDIX 4

Stability of NxN Switch under OPF with i.d. Arrivals

1 Definitions

Most definitions required for the proofs in this appendix are already defined in Chapter 1 and Appendix 1. This appendix uses the same waiting time definitions derived from Figure A1.1. The following are new definitions.

1. A positive-definite transformation matrix,[1] T_{opf}:

$$T_{opf} = T_{lpf} T_{ocf}. \tag{1}$$

For example, for a 2×2 switch, T_{opf} is

$$T_{opf} = \begin{bmatrix} 2 & 1 & 1 & 0 \\ 1 & 2 & 0 & 1 \\ 1 & 0 & 2 & 1 \\ 0 & 1 & 1 & 2 \end{bmatrix} \begin{bmatrix} \lambda_{1,1} & 0 & 0 & 0 \\ 0 & \lambda_{1,2} & 0 & 0 \\ 0 & 0 & \lambda_{2,1} & 0 \\ 0 & 0 & 0 & \lambda_{2,2} \end{bmatrix} \tag{2}$$

---

1. T_{ocf} is defined in Appendix 1, and T_{lpf} is defined in Appendix 2.

2. An OPF request weight vector, $\underline{W}'(n)$, whose elements are defined by equation 4.1.

3. An approximate OPF request weight vector at slot n, $\underline{\tilde{W}}'(n)$:

$$\underline{\tilde{W}}'^{T}(n) = W^{T}(n)T_{lpf}. \tag{3}$$

4. A vector $\underline{e}$ whose elements take a value of one if the corresponding VOQ makes a request or otherwise take a value of zero.

In addition, we also define the following property in order to handle the case that a queue makes no request.

Property 1 When a queue has no occupancy and no arrival in the current slot, i.e., makes no request, it is assumed to have a virtual cell with $\tau_{i,j}(n) = 0$, which overrides Property 2 in Appendix 1.

Also, in this appendix, we assume that the arrivals and the scheduling events take place as shown in Figure A2.1.

Fact 1 A queue with zero occupancy and no arrival cannot make a request. Hence, a queue which makes a request must have either non-zero occupancy or an arrival or both.

The derivation of Fact 1 is identical to that of Fact 1 in Appendix 2.

2 Main Theorem

Theorem A4.1: *Under the OPF algorithm, the queue occupancies are stable for all admissible and independent arrival processes, i.e.,* $E[\|\underline{L}(n)\|] \le C < \infty$.

3 Proof

Since the proof for OCF in Appendix 1 has already established that the stability of the waiting times implies the stability of the occupancies, the proof of the main theorem involves only establishing the stability of the waiting times under OPF.

Consider (i) a quadratic Lyapunov function, $V(\underline{W}(n)) = \underline{W}(n)T_{opf}\underline{W}(n)$, (ii) the approximate next waiting time vector:

$$\tilde{\underline{W}}(n+1) \equiv \underline{W}(n) + \underline{e} - [\underline{S}(n) \cdot \underline{\tau}(n)] \tag{4}$$

and (iii) an OPF request weight vector $W'(n)$ whose each individual element is a function of the waiting times of HOL cells defined as:

$$w'_{i,j}(n) = \begin{cases} R_i + C_j, & L_{i,j}(n) > 0 \\ 0, & \text{otherwise,} \end{cases} \tag{5}$$

where $R_i = \sum_{j}^{N} W_{i,j}(n)$ and $C_j = \sum_{i}^{N} W_{i,j}(n)$.

Lemma 1: *Under the OPF algorithm,* $E[\underline{W}'^T(n)(\underline{A}(n) - \underline{S}^*(n)) \,|\, \underline{W}(n)] \le 0$ *for*

$$\sum_{i=1}^{N} \lambda_{i,j} \le 1,\ \sum_{j=1}^{N} \lambda_{i,j} \le 1,\ \lambda_{i,j} \ge 0,\ \textit{where } \underline{S}^*(n) \textit{ is such that}$$

$$\underline{W}'^T(n)\underline{S}^*(n) = \max(\underline{W}'^T(n)\underline{S}(n)).$$

Proof: Similar to the proof of Lemma 1 in Appendix 2, consider the following two cases. The first case is when the match size is N, $|\underline{S}^*(n)| = N$.

For the match size of N, Property 1 in Chapter 4 implies that

$$\underline{W}'^{T}(n)\underline{S}^{*}(n) = 2\sum_{i,j} W_{i,j}(n) . \tag{6}$$

This is because all inputs and all outputs are selected (matched). Also, for the match size of N,

$$E[\underline{W}^{T}(n)\underline{A}(n) | \underline{W}(n), |\underline{S}^{*}(n)| = N] = \underline{W}^{T}(n)T_{lpf}\underline{\lambda} . \tag{7}$$

Similar to the case of LPF, with the admissibility constraint:
$\sum_{i=1}^{N} \lambda_{i,j} \le 1,\ \sum_{j=1}^{N} \lambda_{i,j} \le 1,\ \lambda_{i,j} \ge 0$,

$$T_{lpf}\underline{\lambda} \le \underline{2} . \tag{8}$$

Substituting equation 8 into equation 7, we obtain,

$$E[\underline{W}^{T}(n)\underline{A}(n) | \underline{W}(n), |\underline{S}^{*}(n)| = N] \le 2\sum_{i,j} W_{i,j}(n) . \tag{9}$$

From equation 6 and equation 9, we prove the first case.

$$E[\underline{W}^{T}(n)\,(\underline{A}(n) - \underline{S}^{*}(n))\,|\underline{W}(n), |\underline{S}^{*}(n)| = N] \le 0 . \tag{10}$$

For the second case when the match size is $k < N$, we follow the same procedure as for LPF: we use Konig-Egervary’s theorem and rely on the same requests permutation shown in Figure A2.2.

As in the case of LPF, for a given set I containing the original indices of the first l rows and a set J containing the original indices of the first $k - l$ columns,
let

$$\underline{\tilde{\lambda}} \equiv E[\underline{A}(n) | I, J] , \tag{11}$$

be a conditional arriving rate vector. From Fact 1, there are only arrivals to the unshaded area (see Figure A2.2). The shaded area has no arrival.

Therefore,

$$\tilde{\lambda}_{i,j} = \begin{cases} \lambda_{i,j},\ i \in I, i \in J \\ 0, \text{otherwise.} \end{cases} \tag{12}$$

With the traffic admissibility constraint: $\sum_{i=1}^{N} \lambda_{i,j} \le 1,\ \sum_{j=1}^{N} \lambda_{i,j} \le 1,\ \lambda_{i,j} \ge 0$,

$$\sum_{i,j} \tilde{\lambda}_{i,j} \le k\,. \tag{13}$$

Now, consider a linear programming problem:

$$\begin{aligned} & \max(\underline{W}'^{T}(n)\underline{\tilde{\lambda}}) \\ & \text{s.t.} \quad \sum_{i=1}^{N} \tilde{\lambda}_{i,j} \le 1,\ \sum_{j=1}^{N} \tilde{\lambda}_{i,j} \le 1,\ \tilde{\lambda}_{i,j} \ge 0 \\ & \sum_{i,j} \tilde{\lambda}_{i,j} \le k \end{aligned} \tag{14}$$

which has a set of $S(n)$ as its extreme points that satisfy the following constraints.

$$\begin{aligned} & \text{s.t.} \quad \sum_{i=1}^{N} S_{i,j}(n) = 1,\ \sum_{j=1}^{N} S_{i,j}(n) = 1,\ S_{i,j}(n) = 0, 1 \\ & \sum_{i,j} S_{i,j}(n) = k \end{aligned} \tag{15}$$

Hence,

$$E\,[\,\underline{W}^{T}(n)\underline{\tilde{\lambda}}\,] \le max\,[\,\underline{W}^{T}(n)\underline{S}(n)\,]\ , \tag{16}$$

$$E\,[\,\underline{W}^{T}(n)\underline{A}(n) | I, J\,] \le max\,[\,\underline{W}'^{T}(n)\underline{S}(n)\,]\ . \tag{17}$$

Since the above is true for all I and J satisfying $|I| + |J| = k$, according to Konig-Egervary's theorem, it follows that:

$$E[\underline{W}^T(n)(\underline{A}(n) - \underline{S}^*(n)) \mid \underline{W}(n), |\underline{S}^*(n)| = k] \leq 0, \quad (18)$$

where $\underline{S}^*(n)$ is such that $\underline{W}^T(n)\underline{S}^*(n) = max[\underline{W}^T(n)\underline{S}(n)]$. This proves the second case when $|\underline{S}^*(n)| < N$.

From both cases, it can be concluded that:

$$E[\underline{W}^T(n)(\underline{A}(n) - \underline{S}^*(n)) \mid \underline{W}(n)] \leq 0. \ \square \quad (19)$$

Note: whenever $S_{i,j}(n)$ cannot be 1 because there is no request, Fact 1 implies that $A_{i,j}(n)$ must be zero, and hence, $\tilde{\lambda}_{i,j} = 0$.

Lemma 2: *Under the OPF algorithm,* $E[\underline{W}^T(n)(\underline{A}(n) - \underline{S}^*(n)) \mid \underline{W}(n)] \leq -2\beta \sum_{i,j} W_{i,j}(n)$ $\forall \underline{\lambda} \leq (1-\beta)\underline{\lambda}_m, 0 < \beta < 1$, *where* $\underline{\lambda}_m$ *is any rate vector such that* $\sum_{i,j} \underline{\lambda}_{m_{i,j}} = N$ *and* $\varepsilon > 0$.

Proof: Following the same steps as done in the proof of Lemma 1 and using the fact that $T_{lpf}\underline{\lambda}_m = \underline{2}$, it can be shown that for every $|\underline{S}^*(n)| = k$:

$$E[\underline{W}^T(n)(\underline{A}(n) - \underline{S}^*(n)) \mid \underline{W}(n), |\underline{S}^*(n)| = k] \leq -2\beta \sum_{i,j} W_{i,j}(n). \ \square \quad (20)$$

Lemma 3: *Under the OPF algorithm,*

$$\mathrm{E}\left[\underline{\tilde{W}}^T(n+1)T_{opf}\underline{\tilde{W}}(n+1) - \underline{W}^T(n)T_{opf}\underline{W}(n) \mid \underline{W}(n)\right] \leq -\varepsilon \sum_{i,j} W_{i,j}(n) + K$$

$\forall \underline{\lambda} \leq (1-\beta)\underline{\lambda}_m, 0 < \beta < 1$, *where* $\underline{\lambda}_m$ *is any rate vector such that* $\sum_{i,j} \underline{\lambda}_{m_{i,j}} = N$, $\varepsilon > 0$ *and K is a finite positive constant.*

Proof: Consider

$$\begin{aligned}&\underline{\tilde{W}}^T(n+1)T_{opf}\underline{\tilde{W}}(n+1)\\&=(\underline{W}(n)+\underline{e}-[\underline{S}^*(n)\cdot\underline{\tau}(n)])^T T_{opf}(\underline{W}(n)+\underline{e}-[\underline{S}^*(n)\cdot\underline{\tau}(n)]).\end{aligned}\tag{21}$$

By expansion,

$$\begin{aligned}&\underline{\tilde{W}}^T(n+1)T_{opf}\underline{\tilde{W}}(n+1)\\&=\underline{W}^T(n)T_{opf}\underline{W}(n)+2\underline{W}^T(n)T_{opf}\underline{e}-2\underline{W}^T(n)T_{opf}[\underline{S}^*(n)\cdot\underline{\tau}(n)]\\&\quad-[\underline{S}^*(n)\cdot\underline{\tau}(n)]T_{opf}\underline{e}+\underline{e}^T T_{opf}\underline{e}\\&\quad+[\underline{S}^*(n)\cdot\underline{\tau}(n)]T_{opf}[\underline{S}^*(n)\cdot\underline{\tau}(n)].\end{aligned}\tag{22}$$

Note that $\underline{W}^T(n)T_{opf}[\underline{S}^*(n)\cdot\underline{\tau}(n)]=[\underline{S}^*(n)\cdot\underline{\tau}(n)]^T T_{opf}\underline{W}(n)$ and that $\underline{W}^T(n)T_{opf}\underline{e}=\underline{e}^T T_{opf}\underline{W}(n)$. Subtracting $\underline{W}^T(n)T_{opf}\underline{W}(n)$ from both sides:

$$\begin{aligned}&\underline{\tilde{W}}^T(n+1)T_{opf}\underline{\tilde{W}}(n+1)-\underline{W}^T(n)T_{opf}\underline{W}(n)\\&=2\underline{W}^T(n)T_{opf}\underline{e}-2\underline{W}^T(n)T_{opf}[\underline{S}^*(n)\cdot\underline{\tau}(n)]\\&\quad-[\underline{S}^*(n)\cdot\underline{\tau}(n)]T_{opf}\underline{e}+\underline{e}^T T_{opf}\underline{e}\\&\quad+[\underline{S}^*(n)\cdot\underline{\tau}(n)]T_{opf}[\underline{S}^*(n)\cdot\underline{\tau}(n)].\end{aligned}\tag{23}$$

Taking the conditional expected value given $\underline{W}(n)$:

$$\begin{aligned}&E\left[\underline{\tilde{W}}^T(n+1)T_{opf}\underline{\tilde{W}}(n+1)-\underline{W}^T(n)T_{opf}\underline{W}(n)|\underline{W}(n)\right]\\&=E\Big[2\underline{W}^T(n)T_{opf}\underline{e}-2\underline{W}^T(n)T_{opf}[\underline{S}^*(n)\cdot\underline{\tau}(n)]\\&\quad-[\underline{S}^*(n)\cdot\underline{\tau}(n)]^T T_{opf}\underline{e}+\underline{e}^T T_{opf}\underline{e}\\&\quad+[\underline{S}^*(n)\cdot\underline{\tau}(n)]^T T_{opf}[\underline{S}^*(n)\cdot\underline{\tau}(n)]\,|\,\underline{W}(n)\Big].\end{aligned}\tag{24}$$

Then evaluate each term on the left hand side of equation 24. For the first term,

$$\underline{W}^T(n)T_{opf}\underline{e} = \underline{W}^T(n)T_{lpf}T_{ocf}\underline{e}\,. \tag{25}$$

But $\underline{W}^T(n)T_{lpf} = \underline{\tilde{W}}'^T(n)$, and $T_{ocf}\underline{e} = \underline{\tilde{\lambda}}$ (see equation 12). Therefore,

$$E\left[2\underline{W}^T(n)T_{opf}\underline{e}|\underline{W}(n)\right] = 2\underline{\tilde{W}}'^T(n)\underline{\tilde{\lambda}}\,. \tag{26}$$

For the second term,

$$\begin{aligned} &E\left[\underline{W}^T(n)T_{opf}[\underline{S}^*(n)\cdot\underline{\tau}(n)]\,|\underline{W}(n)\right] \\ &= \underline{W}^T(n)T_{lpf}E\left[T_{ocf}[\underline{S}^*(n)\cdot\underline{\tau}(n)]\,|\underline{W}(n)\right]. \end{aligned} \tag{27}$$

However, $E\left[T_{ocf}[\underline{S}^*(n)\cdot\underline{\tau}(n)]\,|\underline{W}(n)\right] = \underline{S}^*(n)$, and $\underline{W}^T(n)T_{lpf} = \underline{\tilde{W}}'^T(n)$. Thus,

$$E\left[2\underline{W}^T(n)T_{opf}[\underline{S}^*(n)\cdot\underline{\tau}(n)]\,|\underline{W}(n)\right] = 2\underline{\tilde{W}}'^T(n)\underline{S}^*(n)\,. \tag{28}$$

For the third term,

$$E\left[[\underline{S}^*(n)\cdot\underline{\tau}(n)]^T T_{opf}\underline{e}|\underline{W}(n)\right] = E\left[[\underline{S}^*(n)\cdot\underline{\tau}(n)]^T\middle|\underline{W}(n)\right]T_{opf}\underline{e}\,. \tag{29}$$

Considering the fact that $\underline{0} \le T_{opf}\underline{e} \le \underline{2N}$, then

$$E\left[[\underline{S}^*(n)\cdot\underline{\tau}(n)]^T T_{opf}\underline{e}|\underline{W}(n)\right] \le 2N\sum_{i,j}\frac{S_{i,j}(n)}{\lambda_{i,j}}\,. \tag{30}$$

Using Property 5 in Appendix 1,

$$0 \le \sum_{i,j}\frac{S^*_{i,j}(n)}{\lambda_{i,j}} \le L < \infty\,. \tag{31}$$

Thus,

$$0 \le E\left[[\underline{S}^*(n)\cdot\underline{\tau}(n)]^T T_{opf}\underline{e}|\underline{W}(n)\right] \le 2NL\,. \tag{32}$$

For the forth term,

$$\underline{e}^T T_{opf} \underline{e} = \underline{e}^T T_{lpf} T_{ocf} \underline{e} . \tag{33}$$

Similarly, it can be shown that $\underline{e}^T T_{lpf} \le 2N\left(\underline{1}^T\right)$ and that $T_{ocf}\underline{e} \le \underline{\lambda}$. Consider the traffic admissibility constraint that $\sum_{i,j} \lambda_{i,j} \le N$. Hence,

$$E\left[\underline{e}^T T_{opf} \underline{e}\right] \le 2N\left(\underline{1}^T\right)\underline{\lambda} \le 2N \sum_{i,j} \lambda_{i,j} \le 2N^2 . \tag{34}$$

Now, we consider the last term.

$$\begin{aligned} &[\underline{S}^*(n) \cdot \underline{\tau}(n)]^T T_{opf} [\underline{S}^*(n) \cdot \underline{\tau}(n)] \\ &= [\underline{S}^*(n) \cdot \underline{\tau}(n)]^T T_{lpf} T_{ocf} [\underline{S}^*(n) \cdot \underline{\tau}(n)] \\ &= [\underline{S}^*(n) \cdot \underline{\tau}(n)]^T T_{lpf} [\underline{S}^*(n) \cdot \underline{\tau}(n) \cdot \underline{\lambda}] . \end{aligned} \tag{35}$$

If we expand $T_{lpf}[\underline{S}^*(n) \cdot \underline{\tau}(n) \cdot \underline{\lambda}]$, each i, j th term of this vector is as follows:

$$\sum_k \underline{S}_{i,k}^*(n) \cdot \tau_{i,k}(n) \cdot \lambda_{i,k} + \sum_l \underline{S}_{l,j}^*(n) \cdot \tau_{l,j}(n) \cdot \lambda_{l,j} . \tag{36}$$

Considering the constraint that $\sum_{i=1}^{N} S_{i,j}(n) = 1$, $\sum_{j=1}^{N} S_{i,j}(n) = 1$, $S_{i,j}(n) = 0 \text{ or } 1$. It can be verified that $S_{i,j}(n) S_{i,j'}(n) = 0, \forall j' \ne j$ and that $S_{i,j}(n) S_{i',j}(n) = 0, \forall i' \ne i$.

Therefore,

$$\begin{aligned} &S_{i,j}(n)\tau_{i,j}(n)\left(\sum_k \underline{S}_{i,k}^*(n) \cdot \tau_{i,k}(n) \cdot \lambda_{i,k} + \sum_l \underline{S}_{l,j}^*(n) \cdot \tau_{l,j}(n) \cdot \lambda_{l,j}\right) \\ &= 2S^2_{i,j}(n)\tau^2_{i,j}(n)\lambda_{i,j}. \end{aligned} \tag{37}$$

Since $S_{i,j}(n) = 0 \text{ or } 1$, $S^2_{i,j}(n) = S_{i,j}(n)$, and $E\left[\tau^2_{i,j}(n)\right] = \frac{1}{\lambda_{i,j}^2}$. Hence,

$$E\Big[\,[\underline{S}^*(n)\cdot\underline{\tau}(n)]^T T_{opf}[\underline{S}^*(n)\cdot\underline{\tau}(n)]\,\big|\underline{W}(n)\Big] = 2\sum_{i,j}\frac{S_{i,j}(n)}{\lambda_{i,j}}. \qquad (38)$$

Thus,

$$E\Big[\,[\underline{S}^*(n)\cdot\underline{\tau}(n)]^T T_{opf}[\underline{S}^*(n)\cdot\underline{\tau}(n)]\,\big|\underline{W}(n)\Big] \le 2L. \qquad (39)$$

From equations 26, 28, 30, 34 and 39, we simplify equation 24 as follows:

$$\begin{aligned}&E\Big[\underline{\tilde{W}}^T(n+1)T_{opf}\underline{\tilde{W}}(n+1) - \underline{W}^T(n)T_{opf}\underline{W}(n)\Big|\underline{W}(n)\Big]\\ &\le 2\Big(\underline{\tilde{W}'}^T(n)\underline{\tilde{\lambda}} - \underline{\tilde{W}'}^T(n)\underline{S}^*(n)\Big) + 2N^2 + 2L.\end{aligned} \qquad (40)$$

Furthermore, it can be shown that $\underline{\tilde{W}'}^T(n)\underline{\tilde{\lambda}} = \underline{W'}^T(n)\underline{\tilde{\lambda}} = E[\underline{W'}^T(n)\underline{A}(n)|\underline{W}(n)]$, and that

$$\underline{\tilde{W}'}^T(n)\underline{\tilde{\lambda}} - \underline{\tilde{W}'}^T(n)\underline{S}^*(n) = E[\underline{W'}^T(n)(\underline{A}(n)-\underline{S}^*(n))\,|\underline{W}(n)]\,. \qquad (41)$$

Recall Lemma 2 that $E[\underline{W'}^T(n)(\underline{A}(n)-\underline{S}^*(n))\,|\underline{W}(n)] \le -2\beta\sum_{i,j}W_{i,j}(n)$. We prove Lemma 3:

$$\mathrm{E}\Big[\underline{\tilde{W}}^T(n+1)T_{opf}\underline{\tilde{W}}(n+1) - \underline{W}^T(n)T_{opf}\underline{W}(n)\Big|\ \underline{W}(n)\Big] \le -\varepsilon\sum_{i,j}W_{i,j}(n) + 2N^2 + 2L,$$

where $\varepsilon = 4\beta$ *and* $K = 2N^2 + 2L$. ❐

Lemma 4: *Under the OPF algorithm,*

$$\mathrm{E}\Big[\underline{W}^T(n+1)T_{opf}\underline{W}(n+1) - \underline{W}^T(n)T_{opf}\underline{W}(n)\Big|\ \underline{W}(n)\Big] \le -\varepsilon\sum_{i,j}W_{i,j}(n) + K$$

$\forall\underline{\lambda} \le (1-\beta)\,\underline{\lambda}_m,\ 0<\beta<1$, *where* $\underline{\lambda}_m$ *is any rate vector such that* $\sum_{i,j}\underline{\lambda}_{m_{i,j}} = N,\ \varepsilon > 0$ *and K is a finite positive constant.*

Proof: Similar to the proof of Lemma 4 in Appendix 1, we can draw the following relationship between the two waiting times:

$$\underline{W}^T(n+1)T_{opf}\underline{W}(n+1) \le \tilde{\underline{W}}^T(n+1)T_{opf}\tilde{\underline{W}}(n+1), \forall n. \quad (42)$$

Hence,

$$\begin{aligned} &E\left[\underline{W}^T(n+1)T_{opf}\underline{W}(n+1) - \underline{W}^T(n)T_{opf}\underline{W}(n) \middle| \underline{W}(n)\right] \\ &\le E\left[\tilde{\underline{W}}^T(n+1)T_{opf}\tilde{\underline{W}}(n+1) - \underline{W}^T(n)T_{opf}\underline{W}(n) \middle| \underline{W}(n)\right] \end{aligned}. \quad (43)$$

This proves Lemma 4. ❒

Now, we are ready to prove the main theorem.

Proof of Main Theorem:

There exists a *quadratic Lyapunov function,* $V(\underline{W}(n)) = \underline{W}(n)T_{opf}\underline{W}(n)$, such that:

$$\mathrm{E}\left[V(\underline{W}(n+1)) - V(\underline{W}(n)) \middle| \underline{W}(n)\right] \le -\varepsilon\sum_{i,j} W_{i,j}(n) + K. \quad (44)$$

According to Kumar [32][38], the sum of the waiting times is stable-in-the-mean, i.e.,

$$\frac{1}{N+1}\sum_{n=0}^{N}\sum_{i,j} E\left[W_{i,j}(n)\right] < \infty, \forall N. \quad (45)$$

Furthermore, if the arrivals are independent, the waiting time vector forms a Markov chain, which equation 45 guarantees is a positive recurrence. Hence,

$$E\left[\|\underline{W}(n)\|\right] \le C_1 < \infty. \quad (46)$$

and so,

$$E\left[\|\underline{L}(n)\|\right] \le C_2 < \infty \quad (47)$$

as a result of the proof from Appendix 1.

4 Stability with the Pipeline Delay

4.1 Main Theorem

Theorem A4.2: *Using k slot old weights, the OPF algorithm is stable for all admissible independent arrival processes,* $0 < k < \infty$, *i.e.,* $E[\|\underline{L}(n)\|] \le C < \infty$

4.2 Proof

Similar to the case of LPF, the approach is to establish an upper bound on the expected value of the difference between the total weight of an optimum match and the total weight of a non-optimum match (as a result of the pipeline delay), and then use this upper bound to show that pipelined OPF still maintains the negative single-step drift of the Lyapunov function.

Let $\underline{S}^*(n)$ be the optimum service vector if OPF had been given the correct weights and $\tilde{\underline{S}}(n)$ be the actual service vector which optimizes on the k slot old weights.

Lemma 5: $E[\underline{W}'(n-k)|\underline{W}(n)]\,\underline{S}^*(n) - E[\underline{W}'(n-k)|\underline{W}(n)]\,\tilde{\underline{S}}(n) \le 0$

Proof: Because $\tilde{\underline{S}}(n)$ is a maximum weight match on $\underline{W}'(n-k)$, it follows that

$$\underline{W}'(n-k)\tilde{\underline{S}}(n) \ge \underline{W}'(n-k)\underline{S}^*(n)\,, \tag{48}$$

and so,

$$E[\underline{W}'(n-k)\tilde{\underline{S}}(n)\big|\underline{W}(n), \tilde{\underline{S}}(n)] \ge E[\underline{W}'(n-k)\underline{S}^*(n)\big|\underline{W}(n), \tilde{\underline{S}}(n)]\ . \tag{49}$$

Since $\underline{W}'(n-k)$ and $\underline{S}^*(n)$ are independent of $\tilde{\underline{S}}(n)$, and $\underline{S}^*(n)$ is deterministic given $\underline{W}(n)$, equation 49 becomes

$$E[\underline{W}'(n-k)|\underline{W}(n)]\,\tilde{\underline{S}}(n) \ge E[\underline{W}'(n-k)|\underline{W}(n)]\,\underline{S}^*(n)\,.\ ❒ \tag{50}$$

Lemma 6: $E[\underline{W}'(n)\underline{S}^*(n) - \underline{W}'(n)\tilde{\underline{S}}(n) | \underline{W}'(n)] \leq 2N^2k + \frac{2Nk}{\lambda_{min}}$, *where* λ_{min} *is the smallest non-zero arrival rate.*

Proof: Since $\underline{S}^*(n)$ maximizes $\underline{W}'(n)\underline{S}^*(n)$ and $\tilde{\underline{S}}(n)$ maximizes $\underline{W}'(n-k)\tilde{\underline{S}}(n)$, in order to prove the lemma, we need to find some relationship between $\underline{W}'(n)$ and $\underline{W}'(n-k)$, which can be derived as follows. Consider the weight of a request from input i to output j during a period of k slots, where k is the number of pipeline stages. At the end of the period, the current weight $W'_{i,j}(n)$ can increase from its previous k value by at most $2Nk$ when every queue at i and every queue for j are non-empty and have not been serviced during every slot in the period (i.e. the waiting times of these $2N$ queues all increase by k). Thus,

$$\underline{W}'(n) \leq \underline{W}'(n-k) + \underline{2Nk}\ , \tag{51}$$

which implies that

$$\underline{W}'(n) \leq E[\underline{W}'(n-k) | \underline{W}(n)] + \underline{2Nk}\,. \tag{52}$$

For a lower bound of $\underline{W}'(n)$ with respect to $E[\underline{W}'(n-k) | \underline{W}(n)]$, we consider the case when $Q_{i,j}$ becomes an only non-empty queue at i and an only non-empty queue for j, especially, when $Q_{i,j}$ has no arrival but a departure during every slot in the period. From equation 4, we obtain the following

$$W_{i,j}(n) = W_{i,j}(n-k) + 2k - 2\sum_{l=n-k}^{n} \tau_{i,j}(l)\,. \tag{53}$$

Take the conditional expectation of equation 53 given $\underline{W}(n)$.

$$W_{i,j}(n) = E[W_{i,j}(n-k) | \underline{W}(n)] + 2k - \frac{2k}{\lambda_{i,j}}\,. \tag{54}$$

Hence,

$$\underline{W}'(n) \geq E[\underline{W}'(n-k)|\underline{W}(n)] - \frac{2k}{\lambda_{min}}\underline{1}\,. \tag{55}$$

Combine the two bounds.

$$E[\underline{W}'(n-k)|\underline{W}(n)] - \frac{2k}{\lambda_{min}}\underline{1} \leq \underline{W}'(n) \leq E[\underline{W}'(n-k)|\underline{W}(n)] + \underline{2Nk}\,. \tag{56}$$

Now we are ready to consider the total matching weights. First, we multiply both sides of equation 55 by $\tilde{\underline{S}}(n)$.

$$E[\underline{W}'(n-k)|\underline{W}(n)]\tilde{\underline{S}}(n) - \frac{2k}{\lambda_{min}}\underline{1}\tilde{\underline{S}}(n) \leq \underline{W}'(n)\tilde{\underline{S}}(n)\,. \tag{57}$$

Then, we multiply both sides of equation 52 by $\underline{S}^*(n)$.

$$\underline{W}'(n)\underline{S}^*(n) \leq E[\underline{W}'(n-k)|\underline{W}(n)]\underline{S}^*(n) + \underline{2Nk}\underline{S}^*(n)\,. \tag{58}$$

By subtracting equation 58 by equation 57, we obtain the following,

$$\begin{aligned}\underline{W}'(n)\underline{S}^*(n) - \underline{W}'(n)\tilde{\underline{S}}(n) \leq\ & E[\underline{W}'(n-k)|\underline{W}(n)]\underline{S}^*(n) - E[\underline{W}'(n-k)|\underline{W}(n)]\tilde{\underline{S}}(n) \\ & + \underline{2Nk}\underline{S}^*(n) + \frac{2k}{\lambda_{min}}\underline{1}\tilde{\underline{S}}(n).\end{aligned} \tag{59}$$

Using Lemma 5, we simplify equation 59.

$$\underline{W}'(n)\underline{S}^*(n) - \underline{W}'(n)\tilde{\underline{S}}(n) \leq \underline{2Nk}\underline{S}^*(n) + \frac{2k}{\lambda_{min}}\underline{1}\tilde{\underline{S}}(n)\,. \tag{60}$$

Since $\underline{2Nk}\underline{S}^*(n) \leq 2N^2k$ and $\underline{1}\tilde{\underline{S}}(n) \leq N$, it follows that

$$\underline{W}'(n)\underline{S}^*(n) - \underline{W}'(n)\tilde{\underline{S}}(n) \leq 2N^2k + \frac{2Nk}{\lambda_{min}}\,. \;\square \tag{61}$$

With Lemma 6, we can now prove the main theorem.

Proof of Main Theorem:

Similarly to proof of Theorem 4.1, by taking the same steps as done in Lemma 1, Lemma 2 and Lemma 3 and by using the relationship stated by Lemma 6, we can show that there exists a *quadratic Lyapunov function,* $V(\underline{W}(n)) = \underline{W}(n)T_{opf}\underline{W}(n)$, such that:

$$\mathrm{E}\,[V(\underline{W}(n+1)) - V(\underline{W}(n))|\ \underline{W}(n)] \le -\varepsilon\sum_{i,j} W_{i,j}(n) + K + 2N^2k + \frac{2Nk}{\lambda_{min}}. \qquad (62)$$

And likewise according to Kumar [32][38], the sum of all queue occupancies is stable-in-the-mean, i.e.,

$$\frac{1}{N+1}\sum_{n=0}^{N}\sum_{i,j} E\,[W_{i,j}(n)] < \infty, \forall N. \qquad (63)$$

Furthermore, if the arrivals are independent, the queue occupancy vector forms a Markov chain, which equation 63 guarantees is a positive recurrence. Thus,

$$E\,[\underline{W}(n)] \le C' < \infty \qquad (64)$$