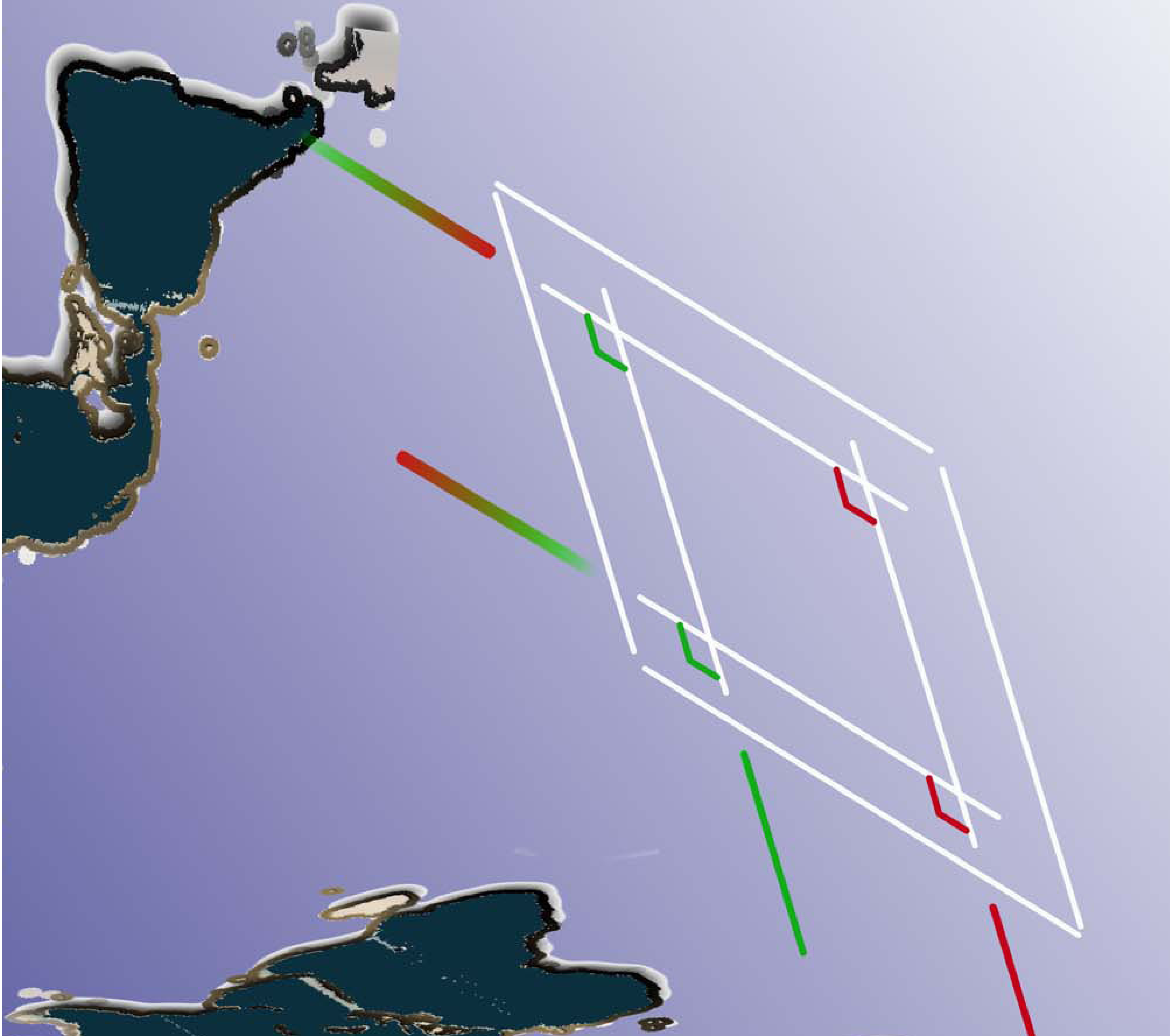
Scheduling in High Performance Buffered Crossbar Switches



Lotfi Mhamdi

Stellingen behorende bij het proefschrift

Scheduling in High Performance Buffered Crossbar Switches

van

Lotfi Mhamdi

Delft, 19 October 2007

1. The creation of the Internet is one step further for humankind to reach the speed of light.

2. The choice between circuit-switching and packet-switching boils down to which is “bandwidth-wise” economically worth it: using more or wasting more.

3. The advantage of packet-switching, over circuit-switching, is statistical multiplexing. It is also the source of all its challenges.

4. Optimal switching performance cannot be obtained through distributed scheduling algorithms only; some sort of centralized knowledge is required.

5. The answer to: “I want a packet-switch that is scalable, has low latency and achieves high throughput” is: “Choose two”.

6. If someone is considering to have telesurgery over the Internet, he is strongly advised to look elsewhere.

7. It is not because things are difficult that we do not dare, it is because we do not dare that they are difficult.

8. Knowledge is one of few resources on earth that multiplies when shared.

9. Vision without action is a daydream; action without vision is a nightmare.

10. Only a fool expects to be happy all the time; happiness, per se, does not exist, there are moments of happiness instead.k

These propositions are considered defendable and opposable and as such have been approved by Prof. dr. K. Goossens.

1. Met het creëren van het Internet is de mensheid één stap dichter bij het bereiken van de lichtsnelheid.

2. De keuze tussen circuit-switching en packet-switching komt neer op welke qua bandbreedte economisch waardevoller is: meer gebruiken of meer verbruiken.

3. Het voordeel van packet-switching ten opzichte, van circuit-switching, is statistisch multiplexen. Dit is tevens de bron van alle uitdagingen.

4. Optimale switching prestaties kunnen niet worden behaald met enkel gedistribueerde algoritmen; een bepaalde vorm van centrale kennis is altijd vereist.

5. Het antwoord op: "Ik wil een packet-switch die schaalbaar is, weinig vertraging heeft en een hoge doorvoersnelheid kan halen" is: "Kies twee".

6. Als iemand overweegt een operatie op afstand over het Internet te ondergaan, wordt diegene ten strengste geadviseerd elders te kijken.

7. Het is niet omdat dingen moeilijk zijn dat we er bang voor zijn, maar omdat we er bang voor zijn lijken dingen moeilijk.

8. Kennis is een van de weinige bronnen op aarde die vermenigvuldigt als ze gedeeld wordt.

9. Visie zonder actie is als een dagdroom; actie zonder visie is een nachtmerrie.

10. Enkel een dwaas verwacht altijd blij te zijn; blijheid, per se, bestaat niet; er zijn echter momenten van blijheid.

Deze stellingen worden verdedigbaar en opponeerbaar geacht en zijn zodanig goedgekeurd door Prof. dr. K. Goossens

Scheduling in High Performance Buffered Crossbar Switches

Scheduling in High Performance Buffered Crossbar Switches

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof.dr.ir. J.T. Fokkema,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen

op vrijdag 19 oktober 2007 om 12:30 uur

door

Lotfi MHAMDI

Master of Philosophy
The Hong Kong University of Science and Technology
geboren te Sidi Bouzid, Tunesïe

Dit proefschrift is goedgekeurd door de promotor:
Prof. dr. S. Vassiliadis

Samenstelling promotiecommissie:

Rector Magnificus, voorzitter	Technische Universiteit Delft, The Netherlands
Prof. dr. S. Vassiliadis, promotor	Technische Universiteit Delft, The Netherlands
Prof. dr. K. Goossens	Technische Universiteit Delft, The Netherlands
Prof. dr. M. Katevenis	The University of Crete and FORTH, Greece
Prof. dr. M. Hamdi	The Hong Kong Univ. of Sci. & Tech., Hong Kong
Prof. dr. N. Ansari	New Jersey Institute of Technology, USA
Dr. K. Bertels	Technische Universiteit Delft,The Netherlands
Dr. C. Minkenberg	IBM Zurich Research Laboratory, Switzerland
Prof. dr. C.Witteveen, reservelid	Technische Universiteit Delft, The Netherlands

CIP-DATA KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Lotfi Mhamdi

Scheduling in High Performance Buffered Crossbar Switches
Delft: TU Delft, Faculty of Elektrotechniek, Wiskunde en Informatica - III
Thesis Technische Universiteit Delft. – With ref. –
Met samenvatting in het Nederlands.

ISBN 978-90-807957-2-3

Subject headings: high performance switching, buffered crossbar fabrics, scheduling, multicast, design, performance.

Copyright © 2007 Lotfi MHAMDI
All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without permission of the author.

Printed in The Netherlands

In memory of Prof. dr. Stamatis Vassiliadis

Scheduling in High Performance Buffered Crossbar Switches

Lotfi Mhamdi

Abstract

Numerous proposals for identifying suitable architectures for high performance packet switches (high speed IP routers and ATM switches) have been investigated and implemented by both academia and industry. These architectures can be classified based on various attributes such as queueing schemes, scheduling algorithms and/or switch fabric topology. Most high performance switches and Internet routers built today use a bufferless crossbar fabric topology. Designing crossbar-based routers that are scalable and provide performance guarantees is challenging with current technology. This is attributed to the high computational complexity of the centralized crossbar scheduler and to the nature of the crossbar-based switching architecture.

This dissertation studies the scheduling problem in buffered crossbar switches, i.e. crossbars with a small amount of internal buffering per crosspoint. The independent scheduling of unicast and multicast traffic flows as well as their integration is considered. A set of distributed and parallel scheduling algorithms, along with appropriate switching architectures, is described. These algorithms are designed to be practical and scalable with router port count and line rate.

A class of unicast scheduling algorithms, where the arbitration process is fully based on the internal buffers information, is described. A switching architecture is proposed, where the schedulers are all embedded within the buffered crossbar fabric chip, resulting in scalable switching and efficient scheduling. The proposed architecture is further shown to provide performance guarantees. With a speedup of two, the proposed architecture is capable of emulating an ideal output queued switch.

The problem of scheduling multicast traffic flows is also studied. A buffered crossbar switching architecture based on input multicast FIFO queues along with appropriate scheduling is proposed and shown to outperform existing architectures. The multicast switching architecture is further improved by using a small number of multicast queues per input port of the switch. A multicast cell assignment algorithm that maps incoming traffic to input queues is devised. The proposed algorithm is shown to assign traffic more efficiently, fairly and quickly than existing algorithms. The study shows an interesting trade off between the number of input multicast queues and the size of internal buffers. This provides enhanced switching performance as well as reduced scheduling complexity, resulting in faster and more scalable switching.

Next, the scheduling of more realistic traffic flows is studied: the combination of unicast and multicast traffic. A buffered crossbar based switching architecture, along with appropriate scheduling that efficiently supports both unicast and multicast traffic flows, is described. The proposed scheduler, while based on a fanout splitting policy, tends not to overload the serial links between the line cards and the fabric core when servicing multicast traffic. The proposed architecture is shown to outperform existing architectures.

Finally, a variation to the buffered crossbar switching architecture is studied. A partially buffered crossbar switching architecture is proposed. It is designed to be a good compromise between the two extreme cases of unbuffered crossbars and fully buffered crossbars. The proposed partially buffered crossbar is based on few internal buffers per fabric output, making its cost comparable to unbuffered crossbars. It also overcomes the centralized crossbar scheduling bottleneck by means of distributed and pipelined schedulers, as in fully buffered crossbars, making it a practical and low cost architecture for such ultra high capacity networks.

Acknowledgements

This dissertation would not have been possible without the help, guidance and encouragement of many people that made my PhD years a joyful journey. First and foremost, I would like to thank my supervisor Prof. Stamatis Vassiliadis. For me, Stamatis was not just a promoter, but also a father, a brother and a dear friend, ultimately a beautiful person. He had a significant influence on me, not just as a student but in my personal life as well. Throughout my technical interactions with him, he continuously surprised me with his ability to quickly change my uncertainty to confidence and vision that made difficult problems easier to solve. It was always a pleasure to work and interact with him, not least for his unique sense of both focus and fun. For his care, love and parental responsibility, I always felt welcome, protected and safe in his presence. I feel privileged to have known him and honored to have carried out my PhD under his guidance. For what seems like an all too brief period, before Prof. Stamatis passed away. I shall remember him always.

Dr. Koen Bertels and dr. Georgi Gaydajiev have worked hard and done a sterling job maintaining the momentum in the CE group before and after the passing of Stamatis. I would like to sincerely thank them for their help, support and encouragement. Prof. dr. Kees Goossens helped me a great deal during the last mile of my PhD work and offered all the guidance I needed. I am sincerely grateful for his many detailed and valuable comments that significantly helped me to shape the thesis into its final form. Prof. Mounir Hamdi was the first to introduce me to the world of research during my Master studies in the Hong Kong University of Science and Technology (HKUST). I would like to deeply thank him for his valuable support and encouragement. During the hardest days of my PhD, Prof. Manolis Katevenis kindly invited me to visit his research group at FORTH, Crete. I would like to thank him for the valuable time he dedicated to me during my visit and for his help and support. I would also like to thank Dr. Cyriel Minkenberg (IBM) for his valuable comments on my thesis draft that enhanced its quality.

My PhD years would not have been joyful without the right work and social environment. I would like to thank my friends and colleagues: Carlo, Christos, Dado, Daniele, Pepijn, Said, Sebastian, Yannis and the rest. Thanks to Carlo for his hospitality and the many delicious Italian dinners we had in his place. I will always remember the special dinners, especially those prepared by Stamatis and followed by various discussions spanning all nature of subjects. Thanks to Daniele for helping with the thesis cover and for his top class amatriciana. Thanks to Pepijn for his support in numerous ways, including the Dutch translation and the thesis cover. My thanks also go to Bert for his technical support and to Lidwina for her administrative assistance throughout my years at TU Delft.

I would like to express my appreciation to Prof. Ghidaoui (HKUST) for his friendship, help and support over the years. Thanks Moe for being always there. I would also like to express my deep thanks and gratitude to Sheena for her friendship throughout these years. Thank you for your unconditional help and support.

Finally, I would like to express my deepest love and gratitude to my parents. My PhD, and this thesis, is a tribute to them for affording me the gift of life and the riches of their love.

Lotfi   Delft, The Netherlands, 2007

Contents

Abstract i

Acknowledgments iii

List of Tables xi

List of Figures xiii

List of Acronyms xix

1 Introduction **1**
- 1.1 Background 1
- 1.2 Motivation 3
- 1.3 Dissertation Contributions and Organization 4

2 High Performance Packet Switches **7**
- 2.1 Why Packet Switching? 7
  - 2.1.1 Statistical Multiplexing 9
  - 2.1.2 Packet Switching Technologies 9
- 2.2 The Architecture of Internet Routers 10
  - 2.2.1 Categories of Routers 10
  - 2.2.2 Basic Architectural Components 12
- 2.3 Packet Switching Architectures 13
  - 2.3.1 Fabric Losslessness and Number of Stages 14

2.3.2 Typical Switch Fabrics 15
2.3.3 Output Queued Switches 17
2.3.4 Input Queued Switches 17
2.3.5 CIOQ Switches 19
2.4 The VOQ Switching Architecture 21
2.4.1 Scheduling in VOQ Switches 22
2.4.2 Maximum Matching Algorithms 23
2.4.3 Practical Maximal Size Matching Algorithms 24
2.5 Buffered Crossbar (CICQ) Switches 26
2.5.1 CICQ Switch Architecture 26
2.5.2 Scheduling in CICQ Switches 28
2.6 Summary 30

3 The Embedded CICQ Scheduling Architecture 31
3.1 Introduction 31
3.2 Conventional CICQ Architecture 33
3.2.1 Scheduling in Conventional CICQ Switches 34
3.2.2 Flow Control in Conventional CICQ Switches 35
3.3 Embedded Scheduling Architecture 36
3.3.1 Reference Architecture 36
3.3.2 The Dynamics of The Switch 37
3.3.3 The Most Critical Buffer First Algorithm (MCBF) . . 39
3.3.4 MCBF Properties 40
3.4 The MCBF Implementation 41
3.4.1 First Approximation: α-MCBF 41
3.4.2 Second Approximation: β-MCBF 45
3.4.3 Extension to Wider Range of Algorithms 49
3.5 Implementation and Performance Results 49
3.5.1 Implementation Results 49
3.5.2 Performance Results 51
3.6 Summary 61

4 Scheduling Multicast Traffic 63

4.1 Introduction 63
4.2 The Multicasting Problem 65
  4.2.1 The Multicast FIFO Architecture 67
  4.2.2 Algorithms For The Multicast FIFO Architecture . . . 68
  4.2.3 The Multicast k FIFO Queues Architecture 69
  4.2.4 Algorithms For The Multicast k FIFO Queues Architecture 70
4.3 The Multicast CICQ Switching Architecture 71
  4.3.1 Switch Model 72
  4.3.2 The Multicast Crosspoint Round Robin Algorithm: MXRR 73
4.4 The Multicast K FIFOs CICQ Switch Architecture 74
  4.4.1 Multicast Cell Assignment 75
  4.4.2 The Modulo Cell Assignment Algorithm 77
  4.4.3 The Multicast k FIFOs Algorithm: MXRR_k 79
4.5 Performance Results 80
  4.5.1 Performance of the Multicast FIFO Architecture . . . 81
  4.5.2 Performance of the Multicast k FIFOs Architecture . . 83
4.6 Summary 87

5 Integrated Unicast and Multicast Scheduling 88

5.1 Introduction 88
5.2 Background 90
5.3 The Integrated CICQ Switching Architecture 92
  5.3.1 Integrated Scheduling 93
5.4 Performance Results 96
  5.4.1 MURS_mix vs. Eslip 98
  5.4.2 The Effect of MQs Number, k 100
  5.4.3 The Number of MQs vs. The XP Size 102
5.5 Hardware Implementation 104

5.6 Summary 106

6 Partially Buffered Crossbar Switches 109

6.1 Introduction 109

6.2 The Partially Buffered Crossbar Architecture (PBC) 112

6.2.1 Switch Model 112

6.2.2 Scheduling Process 113

6.3 Scheduling in PBC Switches 116

6.3.1 The Distributed Round Robin (DRR) Algorithm . . . 116

6.3.2 The Credit Release Delay 118

6.3.3 The DROP Algorithm 119

6.3.4 The Prioritized DROP Algorithm 120

6.4 Performance Results 122

6.4.1 Uniform Traffic 123

6.4.2 Unbalanced Traffic 127

6.5 Summary 128

7 Conclusions 129

7.1 Summary 130

7.2 Contributions 131

7.3 Future Research Directions 133

A Output Queued Switch Emulation 135

A.1 Switch Model and Definitions 136

A.2 FIFO Output Queueing Emulation 139

B Performance Simulation Environment 147

B.1 Simulation Environment 147

B.2 Traffic Scenarios 149

B.2.1 Uniform Traffic 149

B.2.2 Non-Uniform Traffic 150

B.3 Performance Indices 151

Bibliography **153**

List of Publications **163**

Samenvatting **165**

Curriculum Vitae **167**

List of Tables

3.1 Encoding of the number of ‘1’s. 43
3.2 α-MCBF area results. 50
3.3 β-MCBF area results. 51
3.4 Percentage of resource allocation. 51

4.1 The Modulo scheme implementation results for different k. . . 78

5.1 Hardware implementation results. 106

List of Figures

1.1 Growth trends for Internet traffic and computers. 2

2.1 Packet Switches in the Internet. 8
2.2 Basic architectural components of a router. 12
2.3 A bufferless crossbar fabric switch. 15
2.4 Output Queued Switching. 18
2.5 Input Queued Switching. 19
2.6 Combined Input and Output Queued (CIOQ) Switch. 20
2.7 Virtual Output Queueing (VOQ) Switch. 21
2.8 Bipartite graph matching. 22
2.9 Early Buffered Crossbar Switches. 27
2.10 CICQ Switch architecture. 28

3.1 The CICQ Switching architecture. 34
3.2 The CICQ Switching architecture with embedded schedulers. . 37
3.3 Flow control signals usage for different switch sizes. 38
3.4 The buffers occupancy table controller. 42
3.5 α-MCBF input arbiter micro-architecture. 43
3.6 α-MCBF output arbiter micro-architecture. 45
3.7 β-MCBF input arbiter micro-architecture. 47
3.8 The minimum index function. 48
3.9 β-MCBF output arbiter micro-architecture. 48
3.10 Packets flow. 50

3.11 Average delay comparison between using a speedup of 2 and internal buffer size per cross point of 36 cells, under diagonal traffic. 52

3.12 Average delay comparison between using a speedup of 2 and internal buffer size per cross point of 36 cells, under non-uniform unbalanced traffic, ω = 0.5. 53

3.13 Delay performance comparison between the MCBF and α-MCBF schedulers with different internal buffer sizes under Diagonal traffic. 54

3.14 Delay performance comparison between the MCBF and β-MCBF schedulers with different internal buffer sizes under Diagonal traffic. 55

3.15 Delay performance comparison between the MCBF and α-MCBF schedulers with different internal buffer sizes under Unbalanced traffic. 55

3.16 Delay performance comparison between the MCBF and β-MCBF schedulers with different internal buffer sizes under Unbalanced traffic. 56

3.17 Delay performance of the MCBF scheduler with different internal buffer sizes under Diagonal traffic. 57

3.18 Average cell delay performance under uniform traffic. 58

3.19 The Input queues occupancies under uniform traffic. 58

3.20 Stability under unbalanced traffic with internal buffer size of 1 cell. 59

3.21 Stability under unbalanced traffic with internal buffer size of 4 cells. 60

3.22 Stability under unbalanced traffic with internal buffer size of 8 cells. 60

3.23 Average cell latency for different internal buffer settings under unbalanced traffic, $\omega = 0.5$. 61

4.1 Multicast traffic support in core routers. 66

4.2 A 2×4 FIFO multicast crossbar switch. 67

4.3 $N \times M$ multicast CICQ Switch. 71

4.4 A 2×4 multicast CICQ Switch. 72

4.5 An $N \times M$ multicast k FIFO queues CICQ switch. 75
4.6 The Modulo cell placement scheme. 78
4.7 Average cell delay of 8×8 multicast FIFO switch under Bernoulli uniform traffic. 81
4.8 Average cell delay of a multicast FIFO switch under Bursty uniform traffic. 81
4.9 Average cell delay of 16×16 multicast FIFO switch under Bernoulli uniform traffic. 82
4.10 Average cell delay of MXRR with different internal buffer settings. 83
4.11 Throughput comparison between Modulo and Majority cell placement schemes under Bernoulli uniform traffic. 83
4.12 Input queues occupancies of Modulo and Majority under Bernoulli uniform traffic. 84
4.13 Average cell delay of 16×16 multicast k FIFO switch with different numbers of input queues, k = 2, 4. 85
4.14 Average cell delay of MXRR_k with different MQ numbers and XP sizes. 85
4.15 Input queues occupancies with different MQ numbers and XP sizes. 86

5.1 The integrated CICQ Switching architecture. 92
5.2 Average cell delay of MURS_mix and Eslip under Bernoulli uniform unicast traffic ($f_m = 0$). 97
5.3 Average cell delay of MURS_mix and Eslip under Bernoulli uniform mixed traffic, ($f_m = 0.5$). 97
5.4 Average cell delay of MURS_mix and Eslip under Bernoulli uniform multicast traffic ($f_m = 1$). 98
5.5 Throughput performance of MURS_mix and Eslip_4 under different switch sizes and different multicast fractions. 99
5.6 Average cell delay of an 8×8 CICQ switch running MURS with different numbers of MQs per input and mixed input traffic ($f_m = 0.5$). 100

5.7 Average cell delay of a 16×16 CICQ switch running MURS with different numbers of MQs, k, per input and mixed input traffic ($f_m = 0.5$). 101

5.8 Average delay of MURS_mix with different switch sizes and different MQ numbers. 101

5.9 Average cell delay of MURS_mix as a function of the numbers of MQs, the XP sizes and input traffic combinations. 102

5.10 Input queues occupancies of MURS_mix as a function of the numbers of MQs, the XP sizes and input traffic combinations. 103

5.11 The MURS_mix input scheduler algorithm. 104

6.1 Crossbar Fabric variants: (a) Unbuffered Crossbar Fabric. (b) Buffered Crossbar Fabric, with N^2 Internal Buffers. 111

6.2 The Partially Buffered Crossbar (PBC) Switching architecture. 113

6.3 The iSLIP scheduling algorithm. 114

6.4 A PBC Scheduling cycle, 4×4 PBC switch with $B = 2$. . . . 114

6.5 Grant probability as function of switch size, N, and different internal buffers settings. 115

6.6 A DRR scheduling phase for a 4×4 PBC switch with $B = 2$. 118

6.7 Average cell delay of the PBC algorithms under Bernoulli uniform traffic. 122

6.8 Average cell delay of the PBC algorithms under Bursty uniform traffic. 123

6.9 PBC Performance under Bernoulli uniform arrivals. 124

6.10 PBC Performance under Bursty uniform arrivals. 125

6.11 Performance under Bernoulli uniform arrivals. 126

6.12 Performance under Bursty uniform arrivals. 126

6.13 Throughput performance under Unbalanced traffic. 127

A.1 CICQ Switching architecture with embedded schedulers and output queues. 136

A.2 Scheduling phases in an embedded CICQ Switch. 137

A.3 Scheduling phases in embedded CICQ Switch with parallel arrival and input scheduling phases. 145

B.1 The dynamics of a generic switch. 148

List of Acronyms

ATM	Asynchronous Transfer Mode
BOT	Buffer Occupancy Table
CCF	Critical Cell First
CICQ	Combined Input and Crosspoint Queued
CIOQ	Combined Input and Output Queued
CQ	Credit Queue
CXPB	Column of Crosspoint Buffers
DRAM	Dynamic Random Access Memory
DRR	Distributed Round Robin
DSL	Digital Subscriber Line
DTC	Delay Till Critical
DWDM	Dense Wavelength Division Multiplexing
EDF	Early Deadline First
EF	Eligible Fanout
FCFS	First Come First Serve
FIFO	First In First Out
FIRM	FIFO In Round robin Matching
FPGA	Field Programmable Gate Array
FR	Fanout Residue
GBVOQ	Group By Virtual Output Queue
GQ	Grant Queue
GS	Grant Scheduler
HoL	Head of Line
I/O	Input Output
IBT	Input Buffer Table
IBV	Input Buffer Vector
IP	Internet Protocol
IPL	Input Priority List
IPTV	IP Television
IQ	Input Queued
IS	Input Scheduler
ISP	Internet Service Provider
IT	Input Thread
LAN	Local Area Network
LIHP	Last In High Priority
LPF	Longest Port First
LQF	Longest Queue First

LUT	Look Up Table
LXPB	Line of Crosspoint Buffers
MAN	Metropolitan Area Network
MCBF	Most Critical Buffer First
MP	Multicast Pointer
MPE	Masked Priority Encoder
MQ	Multicast Queue
MQF	Multicast Queue Fanout
MRRM	Multicast Round Robin Matching
MSM	Maximum Size Matching
MURS	Multicast and Unicast Round robin Scheduling
MUSF	Most Urgent Cell First
MWM	Maximum Weight Matching
MXRR	Multicast Round Robin
NEC	Nippon Electric Company
OC	Output Cushion
OCF	Oldest Cell First
OPL	Output Priority List
OQ	Output Queued
OS	Output Scheduler
PBC	Partially Buffered Crossbar
PE	Priority Encoder
PIFO	Push In First Out
PIM	Parallel Iterative Matching
PPE	Programmable Priority Encoder
PoP	Point of Presence
QoS	Quality of Service
RGA	Request Grant Accept
RR	Round Robin
RRM	Round Robin Matching
SERDES	Serializer/Deserializer
SRAM	Static Random Access Memory
SRR	Static Round Robin
TTL	Time To Leave
UP	Unicast Pointer
VLSI	Very Large Scale Integration
VOQ	Virtual Output Queueing
WAN	Wide Area Network
WDM	Wavelength Division Multiplexing

WWW	World Wide Web
XP	Internal Crosspoint Buffer

Chapter 1

Introduction

This introductory chapter provides a minimal background of the work presented in this dissertation. The motivation and objectives of the dissertation are discussed. Finally, the chapter overviews the main contributions and outlines the remaining content of the dissertation.

1.1 Background

The concept of computing has progressively shifted from desktop to distributed systems in recent years. The Internet is perhaps the most typical example of a distributed system. While telephone, TV and radio devices have multiplied the power of communication methods, their limited reach combined with their requirements of synchronization in both space and time had left much to be done. The glory of ubiquitous Internet today gives the impression that there are no longer any restrictions on communication. The Internet is a well established worldwide communications medium for the entire spectrum of communication modes (data, voice and video) both real-time and non-real time, affecting every aspect of our lives, economically, politically and culturally. A critical mass of individuals have access to computers and these computers can all talk to each other whether as a global World-Wide-Web (WWW) or peer-to-peer systems. As a consequence, users critically depend on the reliability of the underlying communication network.

Since its conception in the early 1960s, the Internet has changed much. Starting as a research and university network, providing basic services such as e-mail and file transfer, the Internet has grown to be a commercial success with

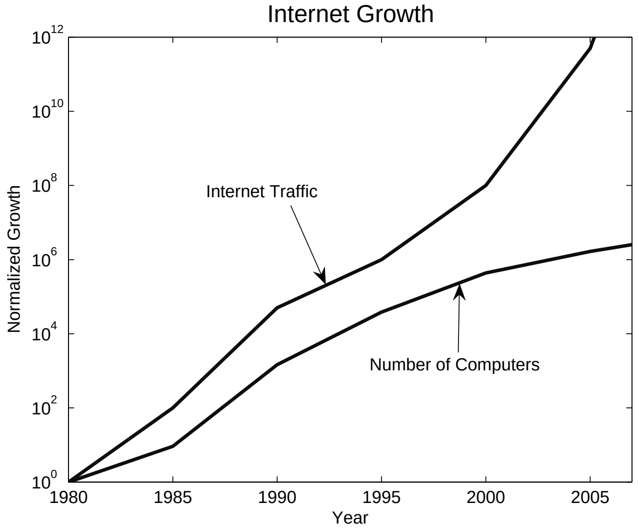


Figure 1.1: Growth trends for Internet traffic and computers.

billions of dollars of annual investment. The Internet, today, consists of thousands of networks. What used to be the “Backbone” in the 1980s is now the interconnection of multiple backbone networks, belonging to large telecommunications providers. Numerous studies have shown that Internet traffic is growing by a factor of 30% per year [1] [2]. According to [3], the number of computers (hosts) on the Internet has exceeded 433 million by the beginning of 2007. Figure 1.1 gives an overview on the growth trends of the Internet over the last two decades.

The Internet is a packet-switched network based on the “statistical multiplexing” paradigm, which means that resources are shared among users rather than dedicated. It is comprised of a mesh of end-hosts, links and routers. Nodes on the Internet, both end hosts and routers, communicate using the Internet Protocol (IP). IP packets travel over links from one router to the next on their way towards their final destinations. A router consists of several processing stages each of which performs a specific task. Amongst others, a router performs two main tasks: *routing* and *switching*. During the routing stage, the router checks the packet header and decides where the packet should be sent next. In the switching stage, the router transfers the packet from its arriving input port to the destination output port in preparation to depart. This dissertation focusses on the switching stage of the router.

1.2 Motivation

The explosive growth in number of users and traffic per user on the Internet is coupled with the same growth in transmission links capacity due to the advances in fiber optic bandwidth. The deployment of wavelength-division multiplexing (WDM) and dense WDM (DWDM) transmission technology has resulted in an abundance of raw bandwidth, already reaching the multi-terabit per second (Tbps) range. Consequently, the total data rate of a single fiber is increasing at a faster rate than the switching and routing equipment that terminates and switches traffic at a carrier's central office or point of presence (PoP). As a result, switches and routers are becoming the true bottleneck of the network. To exacerbate this, the emergence of new applications on the Internet today, such as packetized voice (voice over IP), Internet Television (IPTV) and video multicast streams, require a minimum level of quality of service such as latency and jitter. This results in increased data switching time and can only further widen the gap between transmission links and switching capacities. Therefore, in order to keep up with the Internet growth, backbone, metro and local area networks are facing major engineering challenges of scale, capacity and speed, which will in turn drive their respective network architecture and node design.

Although several switching architectures for high-performance routers have been investigated and implemented, the most prominent and commercially available architecture today is the crossbar-based switch with input queues. The performance of a crossbar-based router critically depends on a centralized and complex *scheduler*, which determines when packets are to cross the switch fabric. Due to the scheduler bottleneck, it is difficult to build a crossbar-based router that meets the aforementioned engineering challenges using current technology. A slight variant of the crossbar switching architecture, a buffered crossbar fabric switch, has recently been shown to overcome the scheduling bottleneck and to have a scalability potential. However, the buffered crossbar architecture has, so far, used a simple mapping of earlier algorithms initially proposed for the unbuffered architecture. These algorithms are distributed over the inputs and the outputs of the switch and require an expensive flow control mechanism, which limits the scalability of the system. Additionally, little has been done to address the increasing number of new applications such as multicast.

The objective of this dissertation is to study the single-stage buffered crossbar switching architecture and solve problems associated with:

- The design of scalable buffered crossbar switches, using appropriate and simple scheduling.
- Providing performance guarantees using scalable buffered crossbars and simple unicast scheduling.
- The scheduling of multicast traffic in buffered crossbars and the integration of multicast and unicast flows.
- The design of “partially” buffered crossbars that benefit from the best of both the unbuffered crossbar and the fully buffered crossbar switching architectures.

1.3 Dissertation Contributions and Organization

The contributions of the dissertation are organized in chapters. Before presenting the contributions, Chapter 2 first provides the necessary background of the work in this dissertation. It surveys packet switching design and scheduling with a focus on single-stage crossbar switches by describing their advantages and limitations. Finally, it summarizes the shortcomings of the buffered crossbar switching architecture to be addressed in subsequent chapters.

Chapter 3 presents our first contribution, the design and implementation of a set of embedded schedulers within the buffered crossbar fabric chip. This stems from the observation that the switching fabric chip is I/O pin count constrained, implying the existence of extra area on the chip. Embedding the schedulers inside the crossbar results in optimized flow control (in terms of pin count) between the crossbar fabric chip and the input line cards. This has the benefit of speeding up the scheduling time while using a *limited* number of control signals, resulting in more scalable crossbar switches. It also improves the performance of the scheduling algorithms, since there are many algorithms that base their decisions on the internal buffers and, if embedded within the crossbar chip, would have faster decisions and cheaper access to resources. Although our devised embedded schedulers were shown to provide high performance under a wide range of unicast traffic patterns, they do not provide performance guarantees. We, then, propose a set of scheduling algorithms, for a buffered crossbar fabric running twice as fast as the external line rate, that can mimic an ideal output queued switch. Our results apply to the class of output queued switches that use a First-In-First-Out (FIFO) output scheduling discipline. We divert the output queueing emulation work to Appendix A.

We study the problem of multicast traffic flows scheduling in Chapter 4. We describe the multicast problem and review prior and related work. We propose an internally buffered multicast switching architecture based on input FIFO queues along with appropriate scheduling. We show that our architecture performs better than existing architectures. We further improve our multicast switching architecture by adding a small number of input queues per port of the switch. We devise a multicast cell assignment algorithm to map incoming traffic to input queues. Our algorithm is shown to assign traffic more efficiently, fairly and quickly than existing algorithms. Our study shows an interesting trade off between the number of input multicast queues and the size of internal buffers. This results in enhanced switching performance as well as reduced scheduling complexity, providing faster and more scalable switching.

In Chapter 5, we proceed to scheduling more realistic traffic flows: the combination of unicast and multicast traffic. We propose a buffered crossbar based switching architecture, along with appropriate scheduling, that efficiently supports both unicast and multicast traffic flows. We propose an integrated scheduler capable of servicing unicast and multicast flows simultaneously. Our proposed scheduler, while based on a fanout splitting policy, tends to not exhaust the serial links between the line cards and the fabric core when servicing multicast traffic. The proposed architecture is shown to outperform existing architectures.

Chapter 6 describes a novel variation to the buffered crossbar switching architecture. We propose a *partially* buffered crossbar switching architecture that is designed to be a good compromise between the two extreme cases of unbuffered crossbars and fully buffered crossbars. The proposed partially buffered crossbar is based on few internal buffers per fabric output, making its cost comparable to unbuffered crossbars. It overcomes the centralized crossbar scheduling bottleneck by using distributed and pipelined schedulers as in fully buffered crossbars, making it a practical and low cost architecture for such ultra high capacity networks.

Finally, Chapter 7 provides concluding remarks on the work presented. The chapter summarizes the dissertation, outlines its contributions and proposes future research directions.

Chapter 2

High Performance Packet Switches

R**outers** constitute the basic, and main, building blocks of the Internet. The design of routers has evolved over the last two decades and different packet-switch architectures have been studied and implemented. These architectures can be classified based on various attributes such as queueing schemes, scheduling algorithms and the switch core topology. This chapter begins with explaining the reasons for using packet-switches rather than circuit-switches. Then, it describes the architectural components of routers. It provides an overview of existing packet-switch architectures and discusses the advantages and drawbacks of each of them. Finally, it summarizes the shortcomings of the buffered crossbar switching architecture to be addressed in subsequent chapters.

2.1 Why Packet Switching?

Communication networks fall into two broad categories: packet-switching and circuit-switching. Within the circuit-switching paradigm, deployed in telephone and telegraph networks more than a century ago, users establish a dedicated connection (also called circuit or channel) with a fixed amount of bandwidth between the source and the destination for the duration of their communication. The channel remains open for the entire duration of the call, irrespective of whether the channel is actually used or not. This approach is efficient for traffic such as telephone voice calls which transmit data at a constant bit

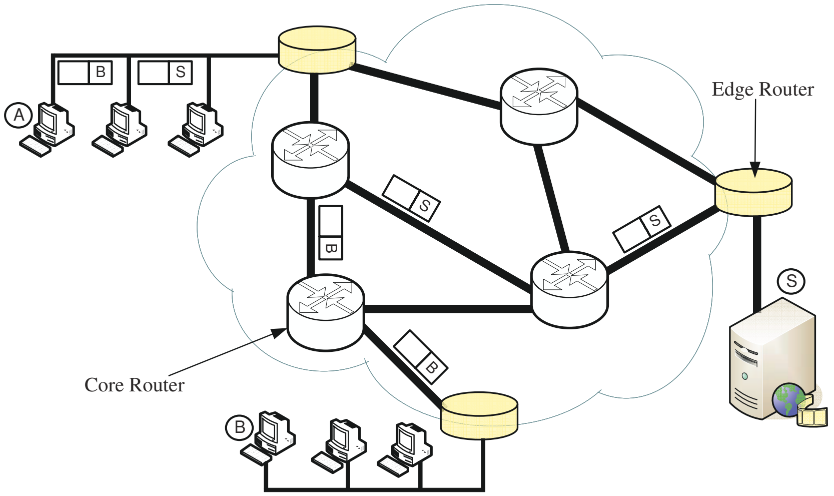


Figure 2.1: Packet Switches in the Internet.

rate wherein connection duration is longer than the amount of time required to establish the connection.

Data and computer communication networks are, however, designed to handle a variety of different types of applications, including applications with time-varying data rates. Early studies have shown that communication between computers could be achieved with packets instead of circuits and that the circuit-switched telephone system was totally inadequate for computer communications [4] [5]. The paradigm of packet-switching networks has first been studied in [6]. In these networks information is carried by packets. Each packet is switched and transmitted through the network, traversing one or more routers, based on the information contained in the packet header. Figure 2.1 depicts packet switches in the Internet and shows how packets are sent from source to destination. Upon reaching their final destination, packets are reassembled to reconstruct the original information. Unlike circuit-switching where no one can use an open channel if its endpoints do not use it, with packet switching, active sources can use any excess capacity made available by the inactive sources.

2.1.1 Statistical Multiplexing

The most important advantage of packet-switching over circuit-switching is its ability to exploit *statistical multiplexing*. To make most efficient use of network bandwidth, connections are statistically multiplexed (shared), to take advantage of their rate variations. In data network environments, carrying traffic such as bursty, sharing network resources can significantly increase the effective capacity of the network. Recent studies have shown a ratio between the peak and average rates as high as 15:1 for data traffic [2]. The bandwidth gain by statistical multiplexing comes, however, at the expense of a serious problem, namely network *contention*. Contention arises when more than one packet contend for the same link at the same time. Since only one packet can be transmitted at a time, the remaining packets need to wait, therefore introducing the requirement for *queues*. As we shall explain later, the queueing discipline employed in a packet-switch is key to its performance. Long-term contention leads to network *congestion*[1]. Network congestion management is important and has been studied since the early days of packet switching [7] [8] [9]. By contrast, because circuit-switching uses resource reservations and dedicated connections for data transfer, there is no requirement for queueing. This is a key difference between the two concepts. The absence of queueing in circuit-switched networks have led a belief to, possibly, enable all-optical switches. Recent studies show that optical transmission links will, soon, reach a saturation point and therefore packet-switches will remain economically cheaper [2]. For the above reasons, the focus in this dissertation will be on electronic packet switching architectures.

2.1.2 Packet Switching Technologies

The two widely known and used packet switching architectures are Asynchronous Transfer Mode (ATM) [10] [11] and Internet Protocol (IP) [12] [5]. ATM is a packet switching technology that uses fixed-size packets (called cells) as the basic transmission unit. Small fixed-size cells allow fast switching and easy, yet efficient, hardware implementation. ATM was designed to be a unifying technology, transporting voice, data, and video and providing sophisticated services such as bandwidth and delay guarantees. The ATM is a connection-oriented technology, based on virtual connection identifiers (VCIs), making the lookup phase simple and fast. However, ATM connections require an overhead of circuit setup and teardown as in circuit-switching, ren-

---

[1] Also called network saturation, overload or oversubscription

dering them less appealing. IP, in contrast, is a connection-less packet switching paradigm. It uses variable-size packets, and supports only one basic service: best effort packet delivery, which does not provide any timeliness or reliability guarantees. Despite the advantages of ATM in terms of quality of service, IP has recently emerged as the dominant architecture and the bearer service for the global information infrastructure. This is mainly due to advances in routing and *scheduling algorithms* for variable-length packets and to the dominance of IP at the endpoints [13]. For more details about circuit- and packet-switching, the reader is referred to books such as [14] [10] and references such as [4] [5] [15].

This dissertation focusses on the switching stage in packet switching architectures in general, irrespective of whether the underlying technology is ATM, IP or proprietary. When the switching architecture is implemented in hardware at very high speed, it is usually tailored towards data units of fixed sizes. Throughout this dissertation, unless otherwise stated, we use the terms cell and packet to refer to the same entity, namely fixed-size data unit. Variable length packets are segmented, into fixed-size units, on their entry to the router and reassembled back to their original lengths at the outputs before being sent out to the outgoing links.

2.2 The Architecture of Internet Routers

Router architectures have evolved over time and in performance. Over the years, several architectures have been used for routers. The choice of a particular architecture is based on several factors such as number of ports, required performance and currently available technology.

2.2.1 Categories of Routers

Routers belong to three broad categories [13], namely access routers, edge routers and core routers. Figure 2.1 shows core and edge routers in the network. Access routers connect end-users from home and the office to Internet Service Providers (ISPs). Owing to the advances in broadband access technologies such as Digital Subscriber Line (DSL), cable modem and gigabit Ethernet, transmission speeds are growing and so is the variety of applications. As a result, the main design factors for access routers are the number of ports and flexibility, to connect more users and to be able to support different protocols. Edge routers (also referred to as enterprise routers) connect endpoints or segments of endpoints, such as Local Area Networks (LANs) or a set of access routers. Edge routers have higher speeds than access routers, usually with high numbers of ports. The main design issues for edge routers include packet classification and filtering for quality of service (QoS) requirements and security reasons. Some of the routers of this category are called “flow-aware” routers [16].

The last category is the core routers (also called backbone routers). As the name suggests, core routers are used in the Internet core. They connect networks, such as Wide Area Networks (WANs). In the Internet backbone, the traffic is aggregated from low speed links. Hence, backbone routers are built to connect few links at very high speed, like OC-192 (10 Gbps) and up to OC-768 (40 Gbps). As the link speed increases, the per-packet processing time (at least table lookup and switching) decreases, making it challenging to design such routers. That is why the datapath of these routers is often implemented in hardware. The main issues in their design are their reliability and their speed. The speed of this category is limited by many obstacles, such as routing, memory bandwidth and switching.

- The routing operation performs a table lookup to match the header of an arriving packet to one of the router output ports. It is often challenging to implement table lookup operation at the line speed.
- The second speed limiting factor is the memory bandwidth. Packets are transmitted over optical links, however they are queued in electronic buffers inside the routers. The wide gap between the optical transmission speed and the electronic memories speed makes it difficult to maintain high routing speeds. Solutions to address this problem have been proposed, such as the combination of Static Random Access Memories (SRAMs) with Dynamic RAMs (DRAMs) [17].
- The third and most severe bottleneck is the *switching fabric and scheduling algorithm*. This component is of utmost importance for the design of a high speed router, as it has significant impact on its overall performance. Since the focus of this dissertation is on the switching stage of high performance routers, Section 2.3 will discuss the switching architectures and scheduling in much more details.

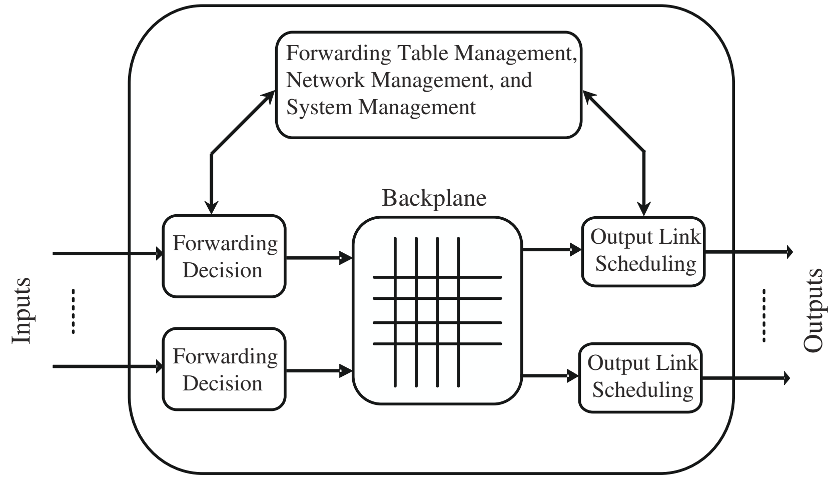


Figure 2.2: Basic architectural components of a router.

2.2.2 Basic Architectural Components

All routers, irrespective of their performance and capacity profile, possess a number of common attributes and perform a set of common tasks. Figure 2.2 illustrates a generic router architecture [18]. The tasks performed by a router can be divided into two types, namely the control path and the datapath.

The control path functionalities are performed and implemented by routing and signaling protocols. They are performed relatively infrequently and are often implemented in software. These functions include routing table construction, maintenance and update as well as system configuration and management. The control path consists of all functions and operations performed by the network to set up and maintain the state required by the data path.

The datapath functions represent the set of operations performed by routers on a per-packet basis. Because of their critical role, the datapath functions are most often implemented in hardware, and include forwarding decision, backplane and output link scheduler. Therefore, scaling the performance of a router implies improving its datapath. The operation of every block of the datapath is as follows:

- **The Forwarding Decision:** It is commonly located in interface cards, which consist of adapters that perform inbound and outbound packet forwarding. On the arrival of a packet, its destination IP address is parsed and looked up. The result of this operation could imply a unicast delivery or a multicast delivery. The packet lifetime is controlled by this component by adjusting the time-to-live field (TTL). This TTL field is used to avoid any indefinite routing (loop) of the same packet. Advanced routers today perform additional tasks, such as packet classification and filtering.

- **The Backplane:** The router backplane (switch fabric) is responsible for transferring packets between the input ports and the output ports. Depending on the backplane, a *scheduler* may be required to make the configuration, or matching, between the input and output interface cards. While waiting its turn to be served across the backplane, a packet may need to be queued. Forwarding a packet through the backplane of a router might seem to be a relatively simple process. But a closer look at this task, performed for each packet, reveals quite a lot has to be done. As we shall see later, the *queueing and scheduling* strategies have an important impact on the performance of the router and on its implementation feasibility. *The main focus of this dissertation is on the design and performance of the backplane of high performance routers.*

- **The Output Link Scheduler:** Once a packet reaches the output port, it is again queued before it can be transmitted to the output link. In most routers today, a single FIFO queue is maintained at each output port and packets are transmitted in the same order of their arrivals. However, advanced routers use different queues to distinguish different flows, or priority classes and schedule the departure time of each packet in order to meet a set of specific QoS guarantees.

2.3 Packet Switching Architectures

A packet-switch[2] (or simply a switch fabric) is a multi-input, multi-output device that connects the input ports of a router to its output ports. The task of the switch fabric is to transfer as many packets as possible from the inputs to the appropriate outputs. The important considerations for the switch fabric design are: throughput, packet loss, packet delay and the complexity of the implementation. Switch fabrics come in different flavors and many architectures have existed over the past. They can be categorized based on different factors,

---

[2] A packet switch is also called backplane, switch fabric or just fabric for short

such lossless vs. lossy, single-stage vs. multi-stage, etc. Substantial research work has been directed at switching architectures [11] [19] [20] [21] [22].

2.3.1 Fabric Losslessness and Number of Stages

Due to the adverse effects of packet loss, the vast majority of switch fabrics are lossless[3]. In order to avoid packet loss, a packet-switch must contain some sort of queueing. Simultaneous arrival of packets, to different inputs, destined to the same output gives rise to a phenomenon called *output contention* (assuming that the output reception capacity is one packet, at most, at a time). When a packet loses contention, it has to be queued. Therefore, in addition to *switching* packets from inputs to outputs, a packet-switch also performs *queueing*. The placement of the queueing function, with respect to the switching function, in a packet-switch is extremely important (see Section 2.3.2). This placement, not only determines the architecture class of a packet-switch, but also has a significant impact on its performance, hardware cost and implementation feasibility.

Switch fabrics can be implemented in a single-stage or in a multi-stage fashion. Single-stage fabrics exhibit strong performance characteristics over multi-stage fabrics. They are non-blocking and connect a set of inputs to a set of outputs through a fast and single path (crosspoint). Single-stage fabrics are easy to build, easy to comprehend and analyze. However, single-stage switches are not scalable, compared to their multi-stage counterparts, as their cost grows quadratically with their input-output port count. Multi-stage fabrics, on the other hand, are built out of a set of single-stage fabrics. Their strong advantage over their single-stage counterparts is their scalability to large port numbers. Examples of multi-stage fabric switches include [24] [25] [26] [27]. A multi-stage fabric is a cascade of single-stage fabrics operating in tandem and in parallel. Therefore, designing a multi-stage fabric reduces to designing single-stage[4] fabrics. We conjecture that the results presented in this dissertation, for single-stage fabrics, will also be useful in the design of high performance multi-stage fabrics. *For the reasons above, this dissertation focusses only on single-stage fabrics*[5].

---

[3] Although lossy architectures have been proposed, such as [23]. This architecture suffers severe packets loss as high as 37% under uniform traffic arrivals.

[4] Normally the transition from single-stage to multi-stage entails many issues. According to [21], these issues can be summarized in network topology, performance, fabric-internal routing, flow control and multicast support.

[5] From now on, we will be using the term fabric to refer to single-stage fabrics

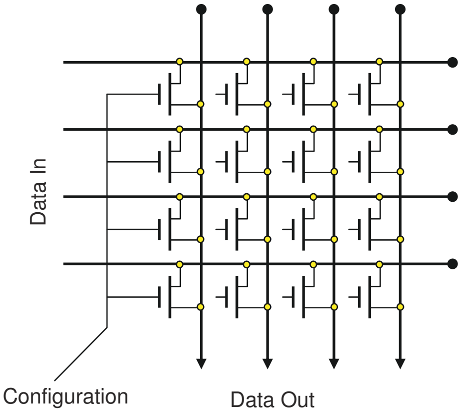


Figure 2.3: A bufferless crossbar fabric switch.

2.3.2 Typical Switch Fabrics

The most common switch fabric architectures in use today are bus-based, shared memory, and crossbar. In this section we present these architectures in turn.

- **Bus:** The simplest switch fabric is the bus. Bus-based routers implement a monolithic fabric comprising a single medium over which all inter-module traffic must flow. The bus architecture is strictly non-blocking, but it allows at most one packet to be transferred at the same time, hence it requires a coordination among the ports. A bus is limited in capacity by its capacitance and by the arbitration overhead for sharing this critical resource. The challenge is that it is almost impossible to build a bus arbitration scheme fast enough to provide non-blocking performance at MultiGigabit speeds. An example of this architecture is the ATOM switch developed by NEC [28].

- **Shared Memory:** The switch fabric can be implemented as shared memory. Incoming packets share a common “shared” buffer memory. Sharing a common buffer pool has the advantage of minimizing the amounts of buffers required to achieve a specified packet loss rate. The idea is that a central buffer is most capable of taking advantage of statistical sharing. If the rate of traffic to one output port is high, it can draw upon more buffer space until the common buffer pool is partially, or completely, filled. However, this may lead to buffer hogging problems, where a flow of packets monopolizes the shared buffers and prevents other packets from accessing it. The major disadvantage of this architecture is the high-speed at which the memory must operate. If the router port number is N and the link speed is S, then a single port shared memory must run as fast as $2NS$. Moreover, as the access time of random access memories is physically limited, this speedup factor limits the ability of this approach to scale up to large sizes and high speeds and thus become its bottleneck. The Prelude switch [29] is an example of this architecture.

- **Crossbar:** A crossbar fabric switch consists of a two-dimensional array of crosspoint switches, one for each input-output pair, as depicted in Figure 2.3. It is one of the most popular interconnection networks used for building input buffered switches because of its low cost, good scalability and non-blocking properties. For an $N \times N$ switch, there are up to N^2 crosspoints. The connection between input i and output j is made by closing the $(i,j)^{th}$ crosspoint in the two-dimensional array. Many commercial routers use crossbar switch fabrics, such as Cisco Systems [30] and Lucent Technology [31].

The crossbar-based fabric architecture is the dominant architecture for today's high-performance packet-switches (IP routers, ATM switches, Ethernet switches) for at least three reasons. First, the crossbars are more scalable than their direct competitors, shared bus and shared memory. This is due to the limitation in bus transfer bandwidth and/or the limitation in the memory access bandwidth. Second, they provide simple point-to-point connections, allowing them to operate at very high speed (up to 10 Gbps). Third, they can support multiple input/output (I/O) transactions simultaneously. This can increase the aggregate bandwidth of the system, which can be in the hundreds of Gbps.

Based on the aforementioned advantages, from here forward, we will assume a single-stage, non-blocking fabric switch such as the crossbar. Since a crossbar switch is lossless, queueing is needed in addition to switching (refer to Section 2.2.2). Fabric switch architectures are classified based on the placement of the *switching* and the *queueing* functions. A crossbar switch belongs to the class of Input Queued (IQ) switches if the queueing takes place before the switching, at the input of the switch. If the queueing is performed after the switching, or at the outputs, the switch is termed Output Queued (OQ). A Combined Input and Output Queued (CIOQ) switch is one where the queueing is performed before and after the switching, in the inputs as well as the outputs of the switch. The last architecture, which is the focus of this dissertation, is the buffered crossbar switch. A buffered crossbar switch is an IQ switch where there is a small amount of limited buffering in each crosspoint. When there is buffering at the inputs, a buffered crossbar switch is also known as the Combined Input and Crosspoint Queued (CICQ) switch or the Combined Input and Crossbar Queued (CICQ) switch [32]. In what follows, we will present each of these architectures and discuss their advantages and limitations.

2.3.3 Output Queued Switches

The output queued (OQ) switch is the ideal switching architecture due to its optimal performance. When a packet arrives at an OQ switch, it is immediately placed in a queue dedicated to its outgoing link. Because no obstacle can prevent an output queue from keeping the outgoing link busy whenever it has a packet, an OQ is known to be *work conserving*. A work conserving switch, such as OQ, has the highest throughput of all switches. Switches and routers have, traditionally, been most often designed with output queueing strategy. It has advantages in that guaranteed QoS can be provided, such as allocating bandwidth to different flows of packets and controlling their delays [33] [34].

Since an OQ switch has no queues at the inputs, all arriving cells must be immediately delivered to their outputs. A major disadvantage is that simultaneous delivery of all arriving cells to the outputs requires too much internal interconnection bandwidth and memory bandwidth. Figure 2.4 depicts an OQ switch with N input ports. There can be up to N cells, one from each input, arriving for the same output simultaneously. In this case, each output memory must perform N write operations (to queue the N packets) and one read operation (to send one packet out). If each external link runs at a rate R, then the memory must run at a speed of $(N + 1)R$. This requirement is known as the internal speedup of a switch [35]. Nowadays, the demand for bandwidth is growing rapidly and with switch sizes continuing to increase, memory bandwidth will be insufficient for output queueing to be practical. As a result the OQ switching architecture is often used as a theoretical reference architecture to assess the performance of alternative, practical, switches.

2.3.4 Input Queued Switches

Figure 2.5 illustrates an input queued (IQ) switching architecture. The IQ has the crossbar running at the same speed as the line rate, R. Queues at the inputs need not receive or send more than one packet simultaneously because

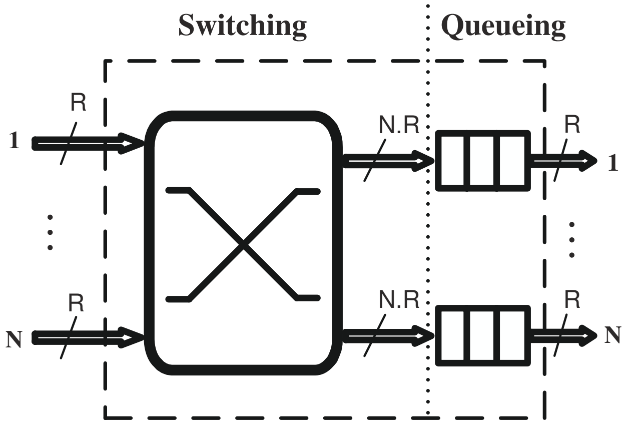


Figure 2.4: Output Queued Switching.

no more than one packet can arrive at or depart from each input in one packet-time[6]. Therefore the memory needs only to operate twice as fast as the line rate, a speed of $2R$ to write-in and read-out a packet. This helps build high-bandwidth IQ switches at low cost and with high scalability features, making them highly appealing. Unfortunately, an IQ switch adopting FIFO queueing at each input has low performance due to the so called head-of-line (HoL) blocking problem [36], described next.

The HoL Blocking Problem

In a FIFO IQ switch, all the cells waiting in an input port are maintained in the same queue. In every time-slot, the HoL cell of each FIFO is considered for scheduling. Since each input cannot receive or send more than one cell in a cell-time, therefore at most only one cell can leave the FIFO of each input. Consider the example in Figure 2.5. The HoL cells of input 1 and input N have the same output port, 1, for which they contend. This implies that only one cell will win the contention for output 1 and will be selected by the scheduler. Let

---

[6] A packet-time is the time duration it takes a packet to go through the switch (back-to-back), which is equal to the time between the arrivals of two consecutive packets to the switch. This, equality, is required in order for the switch to run at the same speed as the external lines. A packet-time is also called time-slot or cell-time. Refer to Section B.1 for more details.

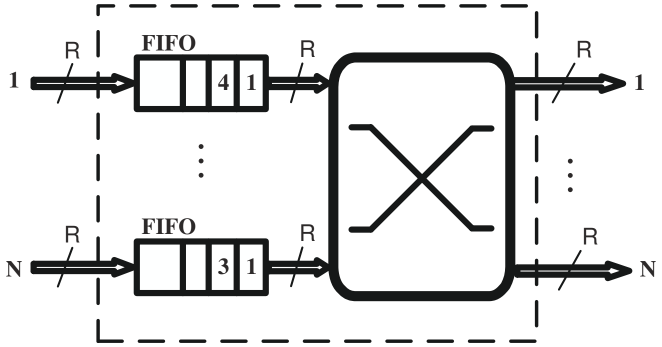


Figure 2.5: Input Queued Switching.

us assume that the scheduler selects the HoL cell of input number N. In this case, the HoL cell of input 1 will remain in the queue and will block the cell behind it (the cell destined to output 4) resulting in output 4 remaining idle despite the existence of cell destined to it. The cell destined to output 4 is prevented from being transferred due the HoL blocking phenomenon, in this case caused by cell 1. It was shown in [36] that under uniform Bernoulli traffic, the HoL blocking problem reduces the achievable throughput to only 58.6%. Worst performance is achieved when the arrival traffic pattern is bursty [37]. Considerable research work has been done to overcome the HoL problem, and different solutions have been proposed, such as the use of a speedup [37] [38]. The HoL problem can, fortunately, be completely eliminated by the use of a simple queueing structure called virtual output queueing (VOQ) [39] [40] [41]. We will discuss the VOQ architecture in Section 2.4.

2.3.5 CIOQ Switches

One of the proposed solutions to overcome the HoL problem is the use of speedup (denoted S, see Figure 2.6) —defined as the ratio at which the internal fabric must operate in comparison to the external links. In [37] and [38], it has been shown that a crossbar switch with a single FIFO at the input can achieve nearly 99% throughput under certain assumptions on the input traffic statistics for speedup range between four and five. When the internal speedup is higher than one, buffering is required at both the inputs and outputs. Thus, a

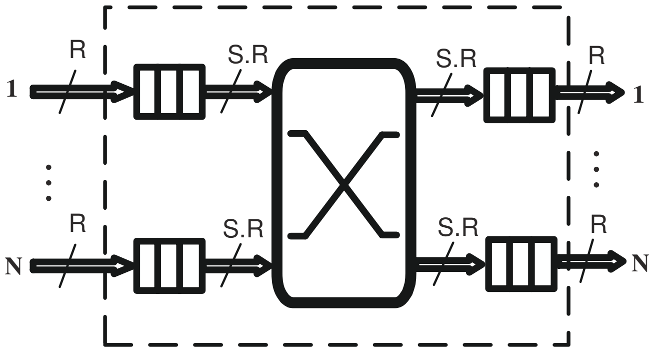


Figure 2.6: Combined Input and Output Queued (CIOQ) Switch.

combination of an input buffered and an output buffered switch is required, i.e., Combined Input and Output Queued (CIOQ) switch as depicted in Figure 2.6.

Some of the designs and the research work carried out on CIOQ was aimed at finding the minimum speedup required to emulate an output queued (OQ) switch. It has been shown in [42] and [43] that a CIOQ is work conserving. In [44], it was proven that a speedup of four is sufficient for a CIOQ to exactly emulate an OQ switch using the Most Urgent Cell First (MUSF) algorithm. An improved result was proposed by [35], with a speedup of just two, a CIOQ can behave identically to an OQ and a speedup of $2 - \frac{1}{N}$ is sufficient to mimic a FIFO-OQ switch. This means that the departure time of each cell is exactly as if an OQ is used and therefore QoS can be guaranteed. The cost of this important result was the use of a more complex scheduling policy called Critical Cell First (CCF). This algorithm requires a push-in queueing structure (PIFO) along with an insertion policy called Last In Highest Priority (LIHP). An attempt to reduce the complexity of this algorithm was based on Delay Till Critical (DTC) strategy, to reduce the number of iterations from N^2 to N, along with an algorithm called Group-By-Virtual-Output Queue (GBVOQ), to reduce the information complexity. Unfortunately, these two solutions cannot be combined, since they are mutually exclusive. Therefore, these results remained of theoretical nature.

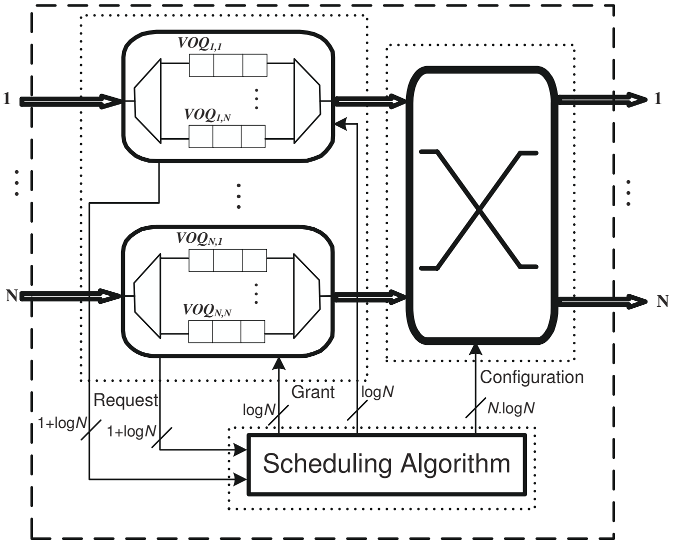
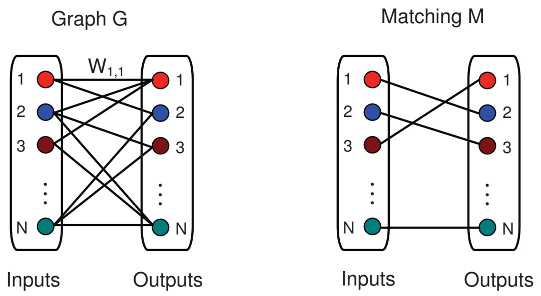


Figure 2.7: Virtual Output Queueing (VOQ) Switch.

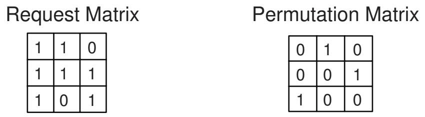
2.4 The VOQ Switching Architecture

Instead of maintaining one FIFO for each input, the Virtual Output Queueing (VOQ) structure is employed. Rather than maintaining a single FIFO queue for all cells, each input maintains a separate queue for each output as shown in Figure 2.7. Thus there are a total of N^2 input queues. Each separate queue is called a VOQ and operates according to the FIFO discipline. The scheduler selects among the HoL cells of each VOQ and transmits them. HoL blocking is eliminated because no cell can be held up by a cell ahead of it that is destined to a different output. When virtual output queueing is employed, the performance of the switch critically depends on the scheduling algorithm used. The scheduling algorithm decides which cells should be transmitted during a cell time under the condition that only one cell can depart from each input and only one cell can arrive at each output.

The scheduler maintains the state of all VOQs in the system, as depicted in Figure 2.7. It does this by keeping N^2 bits, called the state of the VOQs. In every time slot, each input notifies the scheduler whether or not it has cell(s) to be transmitted to the output(s). The input performs this operation by sending a *request* to the scheduler. The request contains the index of the VOQ (log$N$



a) Scheduling as Instance of Bipartite Graph Matching


Request Matrix

1	1	0
1	1	1
1	0	1

Permutation Matrix

0	1	0
0	0	1
1	0	0

b) Scheduling as a Permutation Matrix

Figure 2.8: Bipartite graph matching.

bits) and one additional bit to indicate it state (transition from empty to non empty or vise-versa). Depending on the scheduling policy used, the scheduler may express its willingness to accept the cell. It may do this by sending a *grant* back to the requesting input. The grant contains the index of the destination output, log$N$ bits. Simultaneously, the scheduler sends $N$log$N$ bits to the crossbar fabric to configure the input-output matrix. With suitable scheduling algorithms, an input queued switch using virtual output queueing can increase the throughput from 58.6% [36] to 100% for both uniform and non-uniform traffic [40] [45] [46].

2.4.1 Scheduling in VOQ Switches

The task of the scheduling algorithm is to connect (match) the set of inputs to the set of outputs of a VOQ switch. The matching of input-output pairs must be conflict-free, since each input can send at most one cell and each output can receive at most one cell in every time-slot. Ideally, the scheduler finds the largest possible matching within each cycle to make the most effective possible use of the crossbar. The problem to be solved by the crossbar scheduler is an instance of the bipartite graph matching, as depicted in Figure 2.8 (a).

The scheduling problem can take a matrix representation. Figure 2.8 (b) depicts the equivalent matrix representation for a 3×3 switch (considering only the first 3 input-output pairs of the graph in Figure 2.8 (a)). A request matrix, $\mathcal{R}$, can be used to represent the graph containing the VOQs requests. Each row, i, of the matrix represents an input and each entry, j, in the row represents an output. $\mathcal{R} \equiv [r_{i,j}]$, where $r_{i,j}$ equals to 1 if there are cells in input i destined to output j, 0 otherwise. Finding a one-to-one matching is equivalent to finding a service matrix $\mathcal{S} \equiv [s_{i,j}]$. $\mathcal{S}$ is a permutation matrix, where $s_{i,j} = 1$ indicates that input i is connected to output j, resulting in a cell being transmitted from input i to output j.

Unlike the service matrix, where the entries can take only the values 0 and 1, the entries of the request matrix can take either $\{0,1\}$ values or other values. Depending on the scheduling algorithm used, it is also possible for $r_{i,j}$ to take values such as the number of cells in input i destined to output j, namely the request weight and denoted $w_{i,j}$. Different classes of scheduling algorithms have been proposed and can broadly be categorized into weighted or non-weighted algorithms [40] [45] [47]. The next section discusses these families of algorithms.

2.4.2 Maximum Matching Algorithms

This class of algorithms use weights for the arbitration process. The weight is defined in two different ways, the maximum weight matching or the maximum size matching.

A Maximum Weight Matching (MWM) scheduling algorithm assigns weights to requests. The weight, $w_{i,j}$, of a request from input i to output j, can be, for instance, the number of cells queued in $VOQ_{i,j}$, the age of the HoL cell of $VOQ_{i,j}$ or any other quantity. A MWM algorithm is one that finds the maximum weight matching. In other words, finding the matrix, S^*, that maximizes the total weight, where:

$$\mathcal{S}^* = \arg\max_{\mathcal{S}} (\sum_{i,j} s_{i,j} w_{i,j})$$

The class of MWM scheduling algorithms includes algorithms such as the Longest Queue First (LQF), the Oldest Cell First (OCF) [40] and the longest port first (LPF) [45]. These scheduling algorithms achieve 100% throughput and are stable under any admissible traffic pattern. However, the major problem of these algorithms lies in their high computational complexity. They require $O(N^3 \log N)$ time complexity, making them too complex and too slow for high bandwidth switches.

The Maximum Size Matching (MSM) in a bipartite graph is one that maximizes the number of edges. When the weight of a request takes only the value of either 0 or 1 (indicating the state of a VOQ), finding a maximum size matching is equal to finding the largest size matching between inputs and outputs. This matching maximizes the number of connections made in each time slot, hence maximizing the *instantaneous* throughput of the switch. The MSM for bipartite graph can be found by solving an equivalent network flow problem [48]. Many MSM algorithms exist and the most efficient one known currently has a $O(N^{2.5})$ time complexity [49]. In addition to its high computational complexity, the MSM algorithm is undesirable as it leads to instability and unfairness under non-uniform traffic arrivals [40]. As a result, practical algorithms that approximate the above complex algorithms have been proposed and implemented, such as the class of maximal size matching algorithms.

2.4.3 Practical Maximal Size Matching Algorithms

Although the performance of MWM and MSM algorithms is very good, their high computational complexity prohibits them from being suitable for high bandwidth switches. The alternative was to design algorithms that approximate the optimal solution. These algorithms belong to the class of *maximal* size matching. The difference between a *maximum* size matching and a *maximal* size matching is that, while the former finds the maximum matching, the latter is not guaranteed to do so because once an edge is added to its matching it cannot be removed, even if it does not belong to the maximum matching.

A plethora of maximal size matching algorithms have been proposed over the last two decades [39] [50] [51] [52] [53] [54]. These algorithms iterate over the set of inputs, in parallel, in order to match them to the set of outputs. They perform their matching in a three step process, known as the *Request-Grant-Accept* (RGA) handshaking protocol. The first proposed RGA-based algorithm is the Parallel Iterative Matching (PIM) [39] and was developed by DEC Systems Research Center for a 16×16 switch. The most well known algorithm is iSLIP [53], used by Cisco routers. Although all of these algorithms run a similar RGA protocol, each performs a different set of scheduling criteria. Below we highlight their differences in each step of the RGA protocol.

- **Step 1 (*Request*):** Each unmatched input sends a request to every output for which it has a queued cell. Algorithms that approximate the

MWM have weighted requests equivalent to the associated queue length (iLQF [45]) or waiting time of HoL cell (iOCF [45]). Approximation algorithms for MSM, such as PIM [39], iSLIP [53] and FIRM [51] have requests of weight equal to 1 if the associated queue is not empty.

- **Step 2 (*Grant*):** Each output grants one of the requests received. The granting mechanism depends on the algorithm used. Algorithms that approximate MWM grant to the request with the heaviest weight (either the longest queue or the oldest cell). Grants for MSM approximations are based on a rotating priority scheme, known as highest priority pointer. The pointer movement has a significant consequence on the performance of the algorithm. PIM grants requesting inputs randomly. iSLIP updates its highest priority in a round robin fashion. However, the grant pointer does not move (slips) unless the grant is accepted in the third step. This is very important since it reduces pointers synchronization[7]. FIRM updates its pointer as in iSLIP, except that the pointer moves to the granted input if the grant is dropped in step 3. SRR [54] uses a fully desynchronized round robin updating scheme, which totally overcomes the synchronization effect.

- **Step 3 (*Accept*):** Each input accepts a grant amongst the received ones. Similar to the grant step, the input accepts a grant based either on weights or on a pointer updating scheme.

All the above algorithms have a time complexity of $O(N^2)$ and can be readily implemented in hardware by means of priority encoders. The only complex algorithm amongst them is PIM due to the randomness it uses. Additionally, PIM has low throughput (63%) with one iteration and uniform traffic [53]. To improve the performance of these algorithms, multiple iterations are usually performed. In every iteration, the three RGA steps are executed and the matched input-output pairs are excluded from further iterations. Almost all the above algorithms converge to a MSM match in $O(\log N)$. However, in practice, they usually achieve close to 100% throughput after a few iterations. The implemented algorithms often use speedup between 1.5 and 2 to achieve acceptable performance. The main drawback of these algorithms is their inability to perform well under real traffic patterns, such as non-uniform.

Modified versions of some of these algorithms were devised in order to support

---

[7] The pointer synchronization occurs when the pointers move in a synchronized way, therefore granting always to the same input(s) while only one grant will be accepted. This leads to poor performance, as low as 50% throughput [40]

multicast traffic flows and the combination of unicast and multicast scheduling [18]. Multicast traffic scheduling as well as the combination of unicast and multicast are studied in Chapters 4 and 5, respectively.

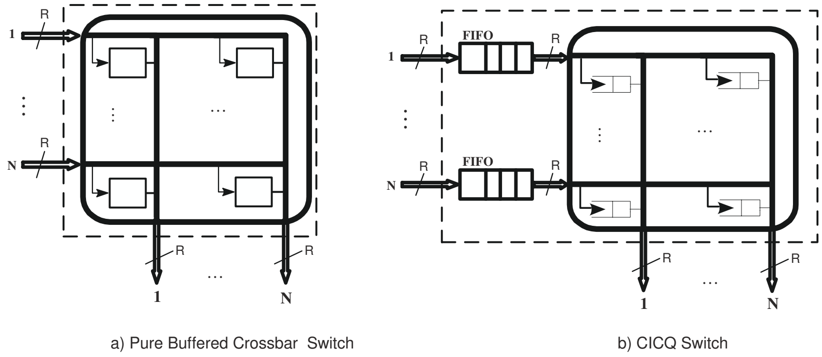
In summary, MWM algorithms are optimal, however they are complex to run at high speeds. Practical algorithms are readily implemented, however they have low performance. This is mainly due to the centralized nature of the bufferless crossbar switching architecture. As a result, alternative switching architectures have been studied to overcome the scheduling problem. A promising alternative is the combined input and crosspoint queued (CICQ) switch architecture, described next.

2.5 Buffered Crossbar (CICQ) Switches

The buffered crossbar fabric is simply a crossbar, where limited buffers exist in each crosspoint. Buffered crossbar switches have been studied for over two decades. The first pure buffered crossbar appeared in 1982 by [55], where buffering exists only inside the crossbar fabric. This architecture is depicted in Figure 2.9 (a) and was implemented by Fujitsu [56]. At that time, it was not possible to embed enough and sufficient buffering on chip and this early architecture was therefore unable to comply with the required cell loss rate. In order to overcome the on-chip memory high requirement, buffered crossbar switches with input queues were proposed [9] [57] [58]. This architecture is based on input queueing and small buffers at the crosspoints, as depicted in Figure 2.9 (b), and is called the combined input crosspoint queued (CICQ) switch. A recent research result showed that a CICQ employing FIFO queueing at the inputs can achieve 100% throughput under uniform traffic arrivals [59]. Additionally, this result showed that the throughput of CICQ switches increases with the switch size. This is in sharp contrast to IQ switches, where the throughput decreases with the switch size, N.

2.5.1 CICQ Switch Architecture

The most widely used CICQ architecture is based on input VOQs that was first proposed by [60]. In the remainder of this dissertation, we will be using the terms buffered crossbar switch and CICQ switch to refer to a buffered crossbar switch using input VOQs. The CICQ has attracted a lot of interest in recent years and different designs have been proposed [61] [62] [63]. Figure 2.10 depicts an $N \times N$ CICQ switch. There are N input line cards, each con-



a) Pure Buffered Crossbar Switch

b) CICQ Switch

Figure 2.9: Early Buffered Crossbar Switches.

sisting of N logically separated VOQs (one per output) and an arbiter (input scheduler). The input scheduler selects a cell to be transmitted next from the input card to the buffered crossbar fabric. Before performing its arbitration, every input scheduler must first check the availability of space inside the internal buffers. This is accomplished by means of a *flow control* mechanism. The buffered crossbar sends up to N bit signals (flow control) to each input scheduler (one per internal buffer in a row belonging to the input scheduler), a total of N^2 flow control signals. The buffered crossbar fabric contains buffers at each crosspoint, a total of N^2 internal crosspoint buffers (denoted as XP). There are N arbiters (output schedulers) inside[8] the buffered crossbar, one per output.

The presence of internal buffers significantly improves the overall performance of the switch due to the advantages it offers. The adoption of internal buffers makes the scheduling totally distributed, hence reducing the arbitration complexity and makes it linear. Consequently, there is no longer any requirement for synchronized decision among the inputs and the outputs as is the case with IQ bufferless switches. This is particularly important for variable length packet scheduling [61]. Moreover, the internal buffers reduce (or avoid) the output contention by allowing the inputs to send cells to an output irrespective of simultaneous cell transfer to the same output.

---

[8] Some researchers assume that the output arbiters are placed outside the buffered crossbar [60] [64] [65]. Refer to Section 3.2.2 for more details.

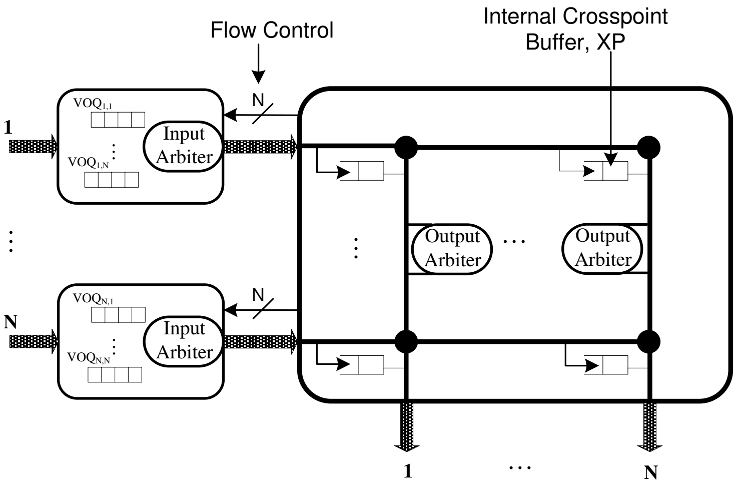


Figure 2.10: CICQ Switch architecture.

2.5.2 Scheduling in CICQ Switches

The appeal of the CICQ architecture is due to its simple and distributed scheduling process. A scheduling cycle consists of three parallel and independent phases as follows:

1. **Input Scheduling**: Every input scheduler selects, independently and in a parallel, one cell from the HoL of an eligible[9] VOQ and transmits it to the buffered crossbar.
2. **Output Scheduling**: Every output scheduler selects, independently and in a parallel, one cell from all the internally buffered cells corresponding to its output and delivers it to the output port.
3. **Flow Control**: Following every output scheduling phase, a flow control is carried from the crossbar to every input to notify the input scheduler about the state of its corresponding internal buffers.

Several scheduling algorithms have recently been proposed for the CICQ architecture. These algorithms can be classified into weight-based schemes [60]

---

[9] A VOQ is eligible if it is not empty and its corresponding internal buffer, XP, is available.

[66] and Round Robin (RR) based schemes [67] [68]. In [60], a scheme based on OCF policy at the input as well as the output scheduling was proposed. While this scheme achieves high throughput under uniform Bernoulli arrivals, the same benefits were not achieved for the non-uniform case. A scheme based on the Longest Queue First (LQF) selection in the input and a Round Robin (RR) arbitration at the output was presented in [66]. The LQF_RR (input_output) scheduling algorithm was proven, through a fluid model, to be stable under uniform input traffic. A set of round robin algorithms were proposed [67] [68] [69] and were shown, through simulation, to achieve high performance under uniform arrivals. These schemes are desirable because of their simplicity in hardware and fairness, however they experience the same problem as in [60] and have low performance under non-uniform traffic patterns. Recent work in [70] has shown that RR based algorithms can provide 100% under uniform traffic inputs.

Alongside the work on scheduling, important research work has been devoted to OQ emulation by a CICQ switch. In [71], it has been proven that a CICQ employing a speedup of two can emulate an OQ switch. This result applies to a wide range of OQ switches policies such as FIFO, strict priority and Early Deadline First (EDF). A more recent and extended result [72] showed that a CICQ switch with two to three times speedup can provide 100% throughput, rate and delay guarantees. Other recent proposals have shown the same results, as above, for variable-length packets as well [73] [74].

The advantages of the CICQ, however, do not come for free. The CICQ switching architecture has the following drawbacks:

- A costly and complex crossbar with N^2 internal buffers is required, where N is the switch valency. These buffers are required to be large enough to match the cell transmission round-trip delays. The quadratic growth of the internal buffers can limit the scalability and implementation feasibility of the switch.
- The CICQ switch requires a large number of flow control signals. Up to N^2 control signals are required to carry the flow control information, from the buffered crossbar core to the input line cards, on a per-packet basis. These signals can double if we consider a CICQ switch with output schedulers implemented in the output cards, such as [60] [64] [65]. In this case, $2N^2$ flow control signals are required. This is undesirable as it severely limits the scalability of the CICQ architecture. Solutions to address this problem have been proposed, such using a limited number (N [62] and $N\log N$ [61]) of I/O pins over multiple time slots. How-

ever, this results in longer time for flow control updating and can cause performance degradation.

- At the scheduling level, very little has been done to address multicast traffic scheduling. Moreover, the integration of unicast and multicast flows in CICQs is still a largely untapped area.

The goal of this dissertation is to address all the above CICQ switching architecture shortcomings and provide solutions to each of them.

2.6 Summary

The design of scalable high performance routers requires improving their datapaths. The underlying packet switching architecture is at the heart of the router datapath. The current chapter gives an overview of existing packet switching architectures and highlights their advantages and limitations. The single-stage IQ bufferless crossbar switch is the prominent architecture for today's high performance routers. This is due to its low hardware cost and scalability. The crossbar requires a centralized scheduler to transfer cells from the input ports to their destination output ports. Unfortunately, the high computational complexity of the optimal MWM algorithms and the low performance of the practical RGA scheduling algorithms makes it difficult for the crossbar to deliver switching at such high bandwidth.

With IQ crossbar switches reaching their practical limitations due to higher port numbers and data rates, buffered crossbar (CICQ) switches are gaining increased interest due to their great potential in solving the complexity and scalability issues faced by their bufferless predecessors. CICQ switches, however, use expensive and complex buffered crossbar fabric. Additionally, the design and placement of the proposed algorithms over the input and output line cards has resulted in excessive use of the buffered crossbar chip pins, limiting its scalability. Additionally, the CICQ has thus far been studied only in the context of unicast traffic scheduling.

This dissertation undertakes a comprehensive study of the CICQ switch architecture, proposes solutions to its scalability and studies its performance under all types of traffic flows. The next chapter addresses the scalability of CICQ switches by optimizing the number of control pins of the buffered crossbar fabric chip.

Chapter 3

The Embedded CICQ Scheduling Architecture

This chapter proposes a novel buffered crossbar (CICQ) switching architecture where the input and output distributed schedulers are embedded inside the crossbar fabric chip. As opposed to previous designs, where these schedulers are spread across the input and output line cards, our design: *(i)* allows the schedulers to have cheap and fast access to the internal buffers; *(ii)* optimizes the flow control mechanism; and *(iii)* provides scalability to the CICQ switching architecture. We propose a novel class of scheduling algorithms, where the arbitration process is based only on information about the internal buffers. We refer to this class of algorithms as the Most Critical Buffer First (MCBF). MCBF is shown to outperform all existing algorithms under various traffic settings. In order to validate our proposal, we implemented, in reconfigurable hardware, a CICQ switch core running the MCBF algorithm, with the maximum port count that we could fit on a single chip. The experiments prove that a 24×24 CICQ switch running a 10 Gbps port speed and a clock cycle time of 6.4 ns can be implemented.

3.1 Introduction

With input queued (IQ) crossbar switches reaching their practical limitations due to higher port numbers and data rates, buffered crossbar (CICQ) switches are gaining a lot of interest due to their great potential in solving the complexity and scalability issues faced by their bufferless predecessors [72]. Contrary to traditional IQ switching, where a centralized and complex scheduler is required [40], for an $N \times N$ CICQ switch, there are N input schedulers and N output schedulers. These schedulers are decoupled and can work independently in parallel [75]. Figure 3.1 depicts the CICQ switching architecture. A scheduling cycle consists of three independent phases: input scheduling, output scheduling and flow control mechanism. The flow control informs the input (output) schedulers about the status, or occupancy, of the internal buffers. It is the only communication means throughout which the schedulers communicate in order to perform their arbitrations and prevent internal buffers overflow.

A plethora of scheduling algorithms has been proposed for the CICQ switching architecture [69] [67] [60]. The vast majority of these algorithms have been designed under the assumption that the input schedulers are located in the input line cards -one per each card- and the output schedulers are placed at the output ports of the switch [60] [64] [65]. This implies that, in every time slot, the flow control mechanism has to communicate to every input (output) scheduler the occupancy of its corresponding internal buffers. This can be considered not only costly, in terms of latency and I/O pins, but also a scalability limiting factor.

In this chapter, we propose a novel design for the CICQ switching architecture where the input and output schedulers are all embedded within the crossbar fabric chip. We propose a novel class of scheduling algorithms that we call MCBF. The MCBF arbitration is fully based on the internal buffers information, unlike previous algorithms that base their arbitration process on the input VOQs. The MCBF input scheduling phase gives priority to the VOQ for which the corresponding internal crosspoint buffer belongs to the least occupied column of internal buffers. Whereas the MCBF output scheduling favors the crosspoint buffer belonging to the most occupied row of internal buffers.

Embedding the schedulers inside the buffered crossbar fabric chip stems from the fact that the crossbar fabric switch is bound by pin count and not by the amount of memory inside the chip. VLSI density increases [62] make it possible to include enough memory inside the crossbar fabric chip. The fabric I/O pin count constraint implies that there must be unused area inside the chip that can be used. The benefits of our proposed design are:

- Optimizing the flow control mechanism between the crossbar fabric chip and the schedulers. This has the benefit of speeding up the scheduling time while using a limited number of I/O pins resulting in more scalable CICQ switches. For a 32×32 switching system, our CICQ embedded switching architecture achieves up to 70% saving of chip I/O control pins when compared to existing CICQ switch architectures.

- Improving the performance of the scheduling algorithms, as there are many algorithms that base their decisions on the internal buffers and when embedded within the crossbar chip would have faster decisions and cheaper access to resources.
- More effective use of the crossbar chip area and saving area on the input (output) line cards that could be used for additional tasks.

Our implementation results showed the feasibility of such a design for a 24×24 CICQ switch core with embedded schedulers running a 10 Gbps port speed. The target technology was the Xilinx Virtex-4FX [76]. Our design can be extended to implement broader classes of scheduling algorithms such as the Longest Queue First-Round Robin (LQF-RR) algorithm [66] at no extra cost. To the best of our knowledge, there has been no other study showing the feasibility of such embedded design. When embedding the scheduler within the crossbar fabric, the flow control mechanism is optimized resulting in the feasibility of implementing scalable switches both in terms of port numbers and speed per port. From a performance viewpoint, our proposed MCBF scheduler was shown to outperform all alternative algorithms under a wide range of traffic patterns.

The remainder of this chapter is structured as follows: Section 3.2 reviews conventional CICQ switch design and scheduling and discusses its limitations. Section 3.3 introduces the embedded CICQ scheduling architecture, discusses its components and explains its dynamics. We describe our proposed MCBF algorithm and illustrate its properties. In Section 3.4, we propose a possible reconfigurable hardware based implementation of the MCBF algorithm. We devise two variations of our design and discuss the implementation as well as the performance of each of them. Section 3.5 presents the implementation results and performance evaluation. Finally, Section 3.6 summarizes the chapter.

3.2 Conventional CICQ Architecture

To keep pace with the Internet's exponential growth, building routers with large port numbers and higher interface speed is becoming a must. Generally, the interconnect runs faster than the line speed to amortize the time spent on some additional requirements such as QoS related processing and imperfect output contention resolution. If we consider transferring packets (or ATM cells), of

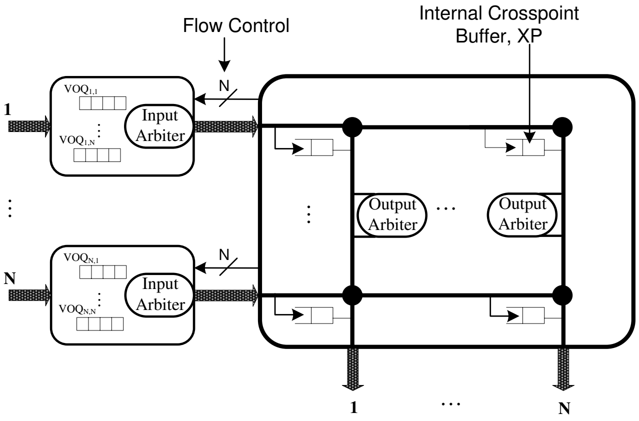


Figure 3.1: The CICQ Switching architecture.

size 53 Bytes each, through a 40 Gbps switch port with a speed up of 2, the scheduler has approximately 5.3 ns to decide which packet to forward. This short time constraint requires the scheduler to make its arbitration as fast as it possibly can. Moreover, current CICQ schedulers rely on a flow control mechanism that requires a non negligible proportion of the crossbar chip pins. As a result, it is difficult to achieve the above goals with current architectures and algorithms. We address each of these issues below.

3.2.1 Scheduling in Conventional CICQ Switches

Recently, there have been many scheduling algorithms proposed for the CICQ switching architecture. Most of these schemes are based on sorting, such as LQF-RR and OCF-OCF, or a combination of sorting and Round-Robin. If we consider the hardware complexity of the input scheduling LQF for example, we can see that it takes a relatively long time to make its arbitration. This is mainly due to the large number of input values (i.e., number of packets in a line card or VOQ) and the basic building blocks of the arbiters, which are mainly two-integer comparators and two-integer MUXes [34]. In a similar implementation (the *i*LQF [45]), it was shown that the arbitration time is more than 7 ns for a 32×32 switch with 10 bits representing the input weight. Even with the fastest implementation, the two-input integer comparator still takes $O(\log B)$ time units to complete the comparison [77], where B is the number of bits equaling to log$L_{\max}$ (the maximum number of packets a line card can hold). The 10 bits representing the weight above correspond to a maximum of 53 KB (1 Kilo ATM cells) as the buffer space at the line card. However, it is normally required that the buffer size at each line card should hold up to 100 ms worth of packets [78]. Meaning that, at 40 Gbps, the buffer size can be as large as 500 MB. Assuming that this buffer space is divided amongst 32 VOQs, this would result in every VOQ having a size of approximately[1] 20 MB and catering for up to 386 Kilo ATM cells. This requires up to 19 bits to represent the input weights. Thus, it is clear that employing LQF (or OCF) arbitration will result in a much longer arbitration time and therefore will most likely be the bottleneck of the whole switch. As a result, designing CICQ schedulers with reduced complexity is required. In fact, CICQ switches are interesting because of their simple and distributed scheduling.

Alongside their complexity, the existing algorithms have been compared to the conventional bufferless IQ crossbar switches. As expected, they were shown to exhibit better performance. These algorithms, however, are just a simple mapping of earlier algorithms proposed for the bufferless crossbar switches into the new CICQ switching architecture. Moreover, none of the algorithms presented in the literature has addressed the issue of flow control optimization and the interaction between the internal buffers and the input line cards.

In fact, as will be illustrated in this chapter (see Section 3.3.3), by carefully considering the interaction between the VOQs and the internal buffers of the CICQ architecture, we can design matched pairs of input/output arbitration algorithms that outperform the straightforward algorithms both in performance and in hardware cost.

3.2.2 Flow Control in Conventional CICQ Switches

The broad class of proposed algorithms can be classified into round robin based algorithms [67] and weighted algorithms [79] [66] or a combination of the two. Most of the proposed algorithms have been designed with the assumption that the input schedulers are taking place at the input line cards and the output schedulers are placed on the output cards. When the input schedulers are implemented on the input line cards, the flow control mechanism can be the bottleneck as the number of ports of the switch or the speed per port increases. For an $N \times N$ CICQ switch, every time slot the flow control mechanism has

---

[1] This is assuming that the VOQs occupancies are balanced and that the incoming traffic is uniformly distributed.

to send N bits (one for each crosspoint buffer) to each input scheduler in order for the latter to know which internal buffer to be served next. A total of up to N^2 flow control signals are required for a lossless switch operation, as shown in Figure 3.1. If we consider some CICQ designs, where the output arbiters are implemented on the output cards [60] [64] [65], then up to $2N^2$ bit signals are required for flow control. Clearly, as N increases, the crossbar implementation becomes infeasible due to I/O pin limitation. The alternative solution to this problem is to sacrifice time instead of pins by using the same limited number of I/O control pins for all input schedulers over many time slots, resulting in longer arbitration times [62].

In order to overcome the buffered crossbar chip I/O limitation, in the next section, we propose the CICQ switching architecture with embedded scheduling that reduces the flow control bit signals requirement and permits the design of scalable CICQ switches.

3.3 Embedded Scheduling Architecture

In this section, we describe our proposed CICQ switching architecture where the input and output schedulers are embedded within the crossbar fabric. We explain its dynamics and the interaction between its components. To show the feasibility of our design, we describe the MCBF class of algorithms in Section 3.3.3 and propose two possible hardware implementations for our algorithm. For the sake of clarity, we introduce some notations that will be used throughout the remainder of this chapter.

3.3.1 Reference Architecture

The proposed CICQ switching architecture with embedded schedulers is depicted in Figure 3.2. Fixed size packets, or cells, are considered. Variable length packets are segmented into cells for internal processing and reassembled before they leave the switch. There are N input cards; each maintaining N logically separated VOQs. When a packet (cell), destined to output j, $1 \leq j \leq N$, arrives to the input card i, $1 \leq i \leq N$, it is held in $\mathit{VOQ}_{i,j}$. In addition to the above, we define the following:

- Eligible VOQ: A $\mathit{VOQ}_{i,j}$ is said to be eligible for being scheduled in the input scheduling process if it is not empty and the internal buffer $\mathit{XP}_{i,j}$ is available to accept at least one cell.

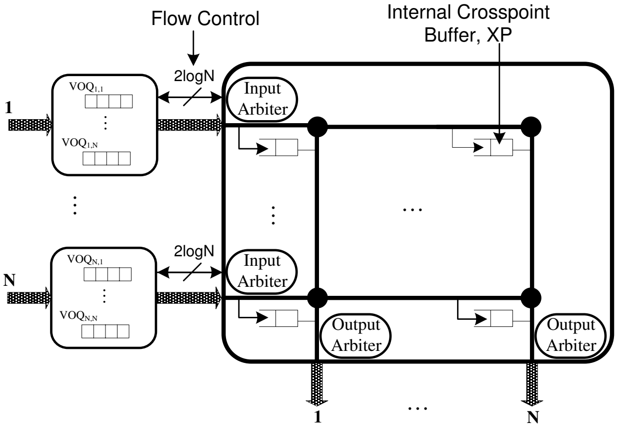


Figure 3.2: The CICQ Switching architecture with embedded schedulers.

- The internal fabric consists of N^2 buffered crosspoints (*XP*), N input schedulers (IS) and N output schedulers (OS). A crosspoint *XP*$_{i,j}$ holds cells coming from input i and destined to output j.
- The line of crosspoint buffers *LXPB*$_i$ is the set of all the internal crosspoint buffers (*XP*$_{i,j}$) that correspond to the same input, i, and holding cells for all outputs. *NLB*$_i$ is the number of cells held in *LXPB*$_i$. *IS*$_i$ schedules the arrival of cells from input card, i, to *LXPB*$_i$.
- The column of the crosspoint buffers *CXPB*$_j$ is the set of the internal buffers (*XP*$_{i,j}$) that correspond to the same output, j, and receiving cells from all inputs. *NCB*$_j$ represents the number of cells queued in *CXPB*$_j$. *OS*$_j$ arbitrates the departure of cells from *CXPB*$_j$.

3.3.2 The Dynamics of The Switch

The proposed CICQ switching architecture has the schedulers embedded within the buffered crossbar core. It works similarly to a conventional CICQ switch, in that the input scheduler *IS*$_i$ controls the transfer of cells from input i to the row (line) of internal buffers *LXPB*$_i$. The output scheduler *OS*$_j$ schedules the departures of cells from the column of internal buffers *CXPB*$_j$. The novelty of our CICQ switching architecture resides mainly in its input scheduling and the flow control between the input line cards and the buffered crossbar

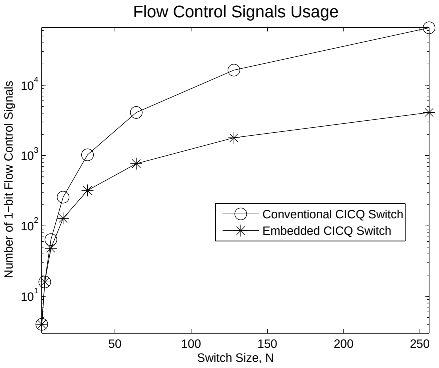


Figure 3.3: Flow control signals usage for different switch sizes.

core. The architecture is shown in Figure 3.2 for an $N \times N$ switch and operates as follows. When a new cell, destined to output j, arrives at input i, the index of *VOQ*$_{i,j}$ is forwarded to *IS*$_i$ using log$N$ flow control bits. *IS*$_i$ keeps record[2] of the arrivals and departures of cells to and from the VOQs, in input i, and updates its record accordingly. Simultaneously with the forwarding of a VOQ index, *IS*$_i$ requests a cell for input scheduling (transfer from the input line card to the internal buffer). It does this by sending log$N$ bits to the line card indicating the index of the selected VOQ. The cell is then forwarded to the corresponding internal buffer. The output scheduler works as in conventional CICQ switches, it checks the existence of cells in its *CXPB* and selects one cell to be transmitted to the output port.

Embedding the schedulers inside the buffered crossbar fabric core has two major advantages. First, the number of flow control 1-bit signals are greatly reduced. Figure 3.3 shows the number of 1-bit signals required for the flow control mechanism of our embedded CICQ switch (as depicted in Figure 3.2) and compares it to the conventional CICQ (as depicted in Figure 3.1). We can see that for any switch size greater than 4 ports, our architecture requires far fewer flow control signals ($O(N\log N)$) as compared to traditional CICQ switches ($O(N^2)$). For a 32×32 switch size, our architecture provides a saving of up to 70% of flow control signaling (I/O pins) as compared to the conventional CICQ architecture. These pins could be used for useful bandwidth for additional ports, resulting in more scalable CICQ switches.

The second advantage of the embedded CICQ switch is its scheduling. When the schedulers are implemented inside the buffered crossbar chip, they can

---

[2] Depending on the implementation, it could be a table with N entries, one per VOQ.

have faster access to the internal buffers, hence faster decisions and access to more information resulting in better and more efficient scheduling. Using embedded schedulers, each input scheduler has access to the state of *all* crosspoints, whereas an external input scheduler can only has access to the state of its own line of internal buffers. In the next section, we propose the MCBF embedded algorithm and discuss the scheduling in more details.

3.3.3 The Most Critical Buffer First Algorithm (MCBF)

The MCBF algorithm is proposed to be a good compromise between performance and hardware cost. It is based on the internal buffers information only. It favors the least occupied internal buffer at the input side. Whereas the output gives priority to the most occupied internal buffer. Meaning that the scheduler retains the information about the internal buffers only, instead of the input queues length in the case of LQF for example. Doing so, B will equal log(PS) instead of log$L_{\max}$, where P is the switch port count and S equals to the internal buffer size in number of cells. The MCBF has the following specification:

Input Scheduling (IS):

- For each input i:
    - Starting from the highest priority pointer location, select the first eligible VOQ corresponding to $\min_j\{NCB_j\}$ and send its HoL cell to the internal crosspoint buffer $XP_{i,j}$.
    - Move the highest priority pointer to the location $(j+1)(\text{mod } N)$.

Output Scheduling (OS):

- For each output j:
    - Starting from the highest priority pointer location, select the first $XP_{i,j}$ corresponding to $\max_i\{NLB_i\}$ and send its HoL cell to the output.
    - Move the highest priority pointer to the location $(i+1)(\text{mod } N)$.

3.3.4 MCBF Properties

The MCBF scheme has three major properties when compared to other schemes. These properties are related to its efficient matching process, its hardware cost and its flow control requirement.

- **Efficient Matching:** MCBF is designed to be a matched pair of input and output scheduling. The internal buffer element is of key importance in finding matched scheduling because of its shared nature. No output is idle as long as $\mathit{NCB}_j \geq 1, \forall 0 \leq j \leq N-1$. To keep the outputs as busy as possible, MCBF maintains a load balancing among the internal buffers. The input and output schedulers each perform a merit function ($\min_j\{\mathit{NCB}_j\}$ and $\max_i\{\mathit{NLB}_i\}$) designed to maintain load balancing inside the buffered crossbar matrix. For the input scheduler, this is achieved by giving priority of service to the crosspoint buffer, XP, belonging to the column of crosspoint buffers (CXPB) with the minimum workload (packets). In this way, and as long as all the CXPB contain cells, the load can be distributed (balanced) over all outputs. Meanwhile, each output arbiter gives priority to the XP belonging to the line of crosspoint buffers (LXPB) with the maximum workload. This enables more work to be brought forward by ensuring free space for the input arbiter (with the least choices of scheduling) to have wider scheduling choices.
- **Hardware Cost:** MCBF is simpler in hardware complexity when compared to LQF-RR or OCF-OCF for example. Recall that the MCBF scheduling decision is based on the number of cells in the internal buffers (NLB_i, NCB_j). That is, for an $N \times N$ one-cell internally buffered crossbar switch, an arbiter's encoder consists only of log$N$ bits ($P = N$ and $S = 1$). This is much faster than comparing log$B$ bits, where B is equal to log$L_{\max}$ in the case of comparing the queues occupancies [45]. Moreover, the product PS remains small irrespective of the internal buffer size. It grows linearly with the switch size and/or the internal buffer size. This makes the hardware sub-blocks of MCBF easier to design and faster to run at high rates.
- **Flow Control:** MCBF is designed to be a stateless scheme with respect to the input line cards. It performs its arbitration with the least interaction with the input VOQs. The only feedback information that MCBF needs to know during its arbitration process is the state of an input VOQ (empty or not). This reduced feedback information between the input line cards and the internal buffers optimizes the flow control mechanism.

In fact, a recent related work has shown the optimal performance of our proposed MCBF algorithm, particularly its output phase, and its ability to achieve 100% throughput [80]. To achieve fairness, each input (output) arbiter maintains a highest priority round robin pointer to break ties among different inputs (outputs) in the presence of conflicts. Similar to LQF [40], the MCBF algorithm has a drawback of input queues starvation. Under certain traffic patterns, MCBF may starve flows. One way to overcome this disadvantage is to use a time stamping mechanism for the MCBF output scheduling. The next section describes the hardware implementation of the MCBF algorithm.

3.4 The MCBF Implementation

Because the scheduling decision is fully based on the internal buffers, XPs, choosing the internal buffer size is critical to MCBF. In our implementation, we target an FPGA device (Xilinx Virtex4-FX) and make use of the Block RAMs (BRAMS) as internal buffers (XPs). If we consider 64-Byte packets and 18 Kbits of buffering space per XP (BRAM size), each XP can accommodate up to 36 packets (cells). The implementation of an MCBF input (output) scheduler consists mainly of two sub-blocks, namely the merit function of computing $\min_j\{NCB_j\}$ ($\max_i\{NLB_i\}$) and the highest priority pointer to break ties in presence of conflicts. However, we can optimize these two sub-blocks in hardware cost while maintaining the same performance of the MCBF algorithm. In what follows, we will propose two alternative implementations that both achieve the same performance as MCBF with lower hardware requirements. In the first implementation, we will approximate the merit function sub-block with a simpler, yet similar, merit function while keeping the highest priority pointer sub-block unchanged. We refer to this implementation as α-MCBF. In the second implementation, we use the same merit function as MCBF, however we omit the highest priority pointer sub-block and we show that it is not indeed required for our specific design. We refer to this implementation as β-MCBF.

3.4.1 First Approximation: α-MCBF

In this implementation, we approximate the merit function of each input (output) scheduler with a simpler merit function that would result in a faster implementation while maintaining similar performance. Recall that each MCBF input scheduler attempts to keep as many output line cards busy as possible.

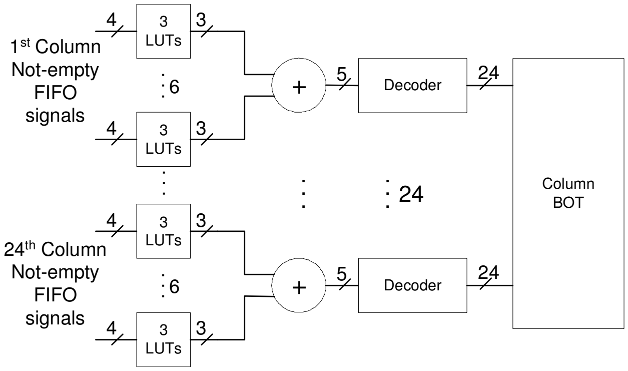


Figure 3.4: The buffers occupancy table controller.

It performs this task by servicing the XP belonging to the the CXPB with the least number of packets. If, instead, the input scheduler gives priority to the XP belonging to the CXPB with the least number of full XPs, this means: (i) Servicing the XP belonging to the CXPB with the least number of full XPs still achieves the objective of the input arbiter by keeping as many output line cards busy as possible. This merit function approximates the original MCBF function well, especially under heavy input loads (which is critical for the scheduler). This is because the likelihood of having many full XPs is proportional to the input load. (ii) Instead of computing $\min_j\{NCB_j\}$ which would require sorting, we can simply avoid it by encoding the number of full XPs per column as one hot. This results in faster implementation with lower cost. The same approximation is applied to each output scheduler. Instead of computing $\max_i\{NLB_i\}$, each output scheduler gives priority to the XP belonging to the LXPB with the maximum number of full XPs.

To efficiently map the input scheduler into hardware the following structure has been used. Inside the crossbar fabric, there is one 24×24-bit double array, named Column Buffer Occupancy Table (C_BOT), and each row represents the number of occupied internal buffers, XP, for each column of the crossbar fabric. Each row is initialized with the first bit asserted "1" and all the others with "0". The position of the "1" in the row represents the number of occupied internal buffers in this column. The controller of the C_BOT is depicted in Figure 3.4. Each Xilinx Virtex Look-Up-Table consists of 4 inputs and 1 output. Hence, 4 not-empty signals are used as inputs to the LUTs to encode the number of ones, as it is shown in Table 3.1. The number of the occupied internal buffers (XPs) for a column are added and then decoded and forwarded to the C_BOT. For example, in Figure 3.5, the first, the second and the fourth

Table 3.1: Encoding of the number of ‘1’s.

Not-Empty	LUT Output
0000	000
0001	001
0010	001
0011	010
0100	001
...	...
1111	100

columns have 2 occupied internal buffers, while the third and the fifth columns have 3 occupied internal buffers.

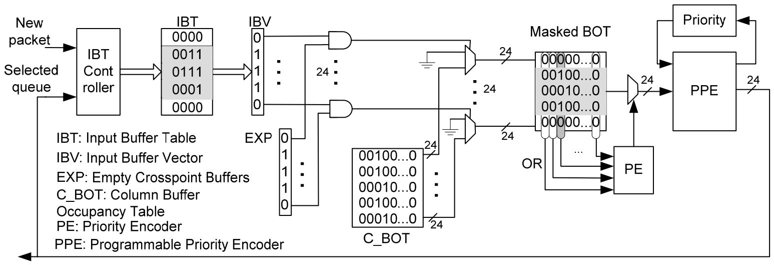


Figure 3.5: α-MCBF input arbiter micro-architecture.

The micro-architecture of the input arbiter is shown in Figure 3.5. When a new packet arrives to the input card, a signal is asserted stating the id of the VOQ. The input arbiter first updates the Input Buffer Table (IBT). The IBT keeps the number of waiting cells in the input line card. The number of waiting cells is represented using 15 bits (up to 32 Kilo cells). The input card asserts a “new packet” for a specific VOQ only when the number of waiting cells in the corresponding entry in the table is less than 4 (The input card keeps a record of the number of new cells and selected cells of a VOQ). Thus, the IBT is used mainly to speed up the time consuming process of communication between the input card and the crossbar switch. The interaction between the input line cards and the buffered crossbar fabric is as depicted in Figure 3.2. In our design, each input card uses 6 signals (5 signals for the 24 input VOQs and 1 signal for valid data) to notify the switch fabric card that a new packet has arrived to the VOQs (indexed by the 5 bits signal). Once the input scheduler has decided which input VOQ is selected, a 6 bits signal is sent back to the input line card containing the selected VOQ index. The cell is then forwarded to the corresponding internal buffer. Cells can be sent using the SERDES transceivers [81]. The output card is simpler than the input card. Cells are, again, sent using the SERDES transceivers as soon as the output scheduler makes a decision. Furthermore, no flow control is needed since the output scheduler is moved inside the buffered crossbar fabric chip.

The Input Buffer Vector (IBV) is a 24-bit vector that represents the state of the IBT. If the row is 0 then the bit is also 0; otherwise it is 1. IBV can be obtained as follows:

$$IBV_j = \begin{cases} 0 & \text{if } \mathit{IBT}_j = 0; \\ 1 & \text{otherwise}. \end{cases}$$

The Empty Crosspoint Buffers (EXP) vector represents the empty (available) internal crosspoint buffer, XP, belonging to a LXPB. The IBV is AND-ed with the EXP, resulting in a vector that represents the eligible queues. This vector is used as a mask for the C_BOT and a new table (Masked BOT) is created that represents the number of occupied buffers of the eligible queues . Each row, j, of the Masked BOT ($\mathit{M\_BOT}_j$) can be computed as follows:

$$\mathit{M\_BOT}_j = \mathit{C\_BOT}_j \wedge \mathit{IBV}_j \wedge \mathit{EXP}_j$$

The elements of each column of this table are OR-ed and are forwarded to a priority encoder to find the eligible queues with the minimum number of occupied buffers. The priority is given to the first column, from the left (in order to find the minimum), with a non-zero value. If we consider the example of the Masked BOT in Figure 3.5, we can see that CXPB_1 and CXPB_{23} each has two full XPs which corresponds to $\min_j\{\mathit{NCB}_j\}$. Using this micro-architecture we can easily locate the VOQ corresponding to the CXPB with the minimum number of full XPs, among all eligible queues. Finally, this vector is forwarded to a Programmable Priority Encoder (PPE) to select the queue based on the highest round-robin priority pointer. The PPE can be implemented in several ways as it is shown in [82]. In this implementation we used the fastest implementation, segmented in 3 clock cycles.

The output arbiter, depicted in Figure 3.6, is similar to, even simpler than, the input arbiter. A 24×24-bit array, named Row Buffer Occupancy Table (R_BOT), is used to store the occupancy of the internal buffers for each row of the buffered crossbar fabric. The position of the '1' in the entry, i, in the array represents the number of queued cells in the line of crosspoint buffers LXPB_i. For example, if the '1' is on the third position it means that LXPB_i has

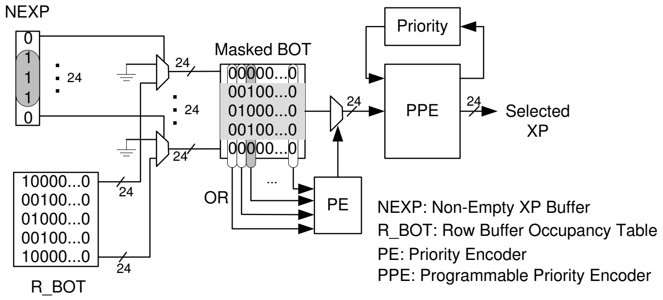


Figure 3.6: α-MCBF output arbiter micro-architecture.

2 queued cells. The Non-Empty Crosspoint buffers (NEXP) is a 24-bit vector that represents the non-empty internal crosspoint buffer, XP, belonging to a LXPB. NEXP is used as a mask for the R_BOT to create a masked BOT. Each row, j, of the Masked BOT (M_BOT_j) can be computed as follows:

$$M\_BOT_j = \begin{cases} R\_BOT_j & \text{if } NEXP_j = 1; \\ 0 & \text{otherwise}. \end{cases}$$

Each 24-bit column of the masked BOT is OR-ed and the first column from the right with a non-zero value is forwarded to the Priority Encoder (PE). The priority is given to the first column from the right in order to find the maximum value. Then, this vector is forwarded to the PPE to decide which will be the selected crosspoint, XP, based on a the highest priority pointer.

3.4.2 Second Approximation: β-MCBF

In this implementation, we omit the highest priority pointer sub-block while keeping the original MCBF merit function for each input and output scheduler. The highest priority pointer is used to break ties in the presence of conflicts. If two or more CXPBs (or LXPBs) have the same number of packets, then the choice of which to favor is based on the highest priority pointer's location. However, by observing that our target design is a 24×24 CICQ with each XP size of 36 cells, the chance of having two or more CXPBs (or LXPBs) with the same number of packets is very low. The likelihood of having two or more CXPBs with equal number of packets, which requires the highest priority pointer to decide which one to favor, is less than 0.1% (or $\frac{1}{24 \times 36}$). Therefore, the possibility of unfairly favoring a XP over others is very unlikely. In fact, we designed the merit function of each input (output) scheduler using a carry propagation circuit (see Figure 3.8). This affords us the option to break ties based on the lowest port number first, which is commonly used by other weight based scheduling algorithms [60]. Additionally, from a performance point of view, simulation results showed that the average cell delay of MCBF is shorter without using the highest priority pointer[3]. Finally, the elimination of the highest priority pointer sub-block, which requires implementing a PPE that takes 3 clock cycles, results in a faster implementation of both the input and output schedulers.

The architecture of the β-MCBF scheme is depicted in Figure 3.7. The Encoder, the Accumulator and the Column Buffer Occupancy Table (C_BOT) are common for all the input arbiters. The accumulator is used to keep the number of cells stored for each column. Each time a new selection has to be made, the accumulator adds the number of new incoming packets for the specific column and subtracts one packet if the output arbiter has sent any. The maximum number of new packets is the same as the number of input arbiters (24). These signals are encoded using a 25-to-5 encoder and used by the accumulator. The maximum number of cells that each column has to store is 864 (24 XP buffers, each with 36 cells), hence the accumulator is 10-bits wide. The C_BOT stores the data of the accumulator used for each input arbiter. If there is a waiting cell at an input queue for which there is space in the corresponding internal buffer, then the queue is deemed eligible and the row of the corresponding C_BOT is forwarded to the masked C_BOT. Otherwise, the row in the masked C_BOT is filled with 1's (indicating it is not the minimum). The rows of the masked C_BOT are used as entries to a *Min Index Function* circuit. The *Min Index Function* outputs the index of the eligible VOQ with the corresponding least occupied CXPB.

The minimum index function is performed using a tree structure with comparators as it is shown in Figure 3.8. In each stage, the elements are compared in pairs and the minimum (or maximum, in the case of the output arbiter) is propagated to the next stage along with its partial index (in the first stage the output index is 1 bit, in the second 2-bits and so on). The detailed structure of finding the minimum among two elements is also depicted in Figure 3.8. The MIN box performs the following: one of the inputs is inverted and a carry-propagation circuit is used, if the carry is '1' then the inverted input is smaller than the normal input. The output of the MIN is used to select the minimum element and its partial input index through a multiplexer and the outputs of the

---

[3] The better performance comes at the cost of unfairness. See Section 3.5.2 for more details.

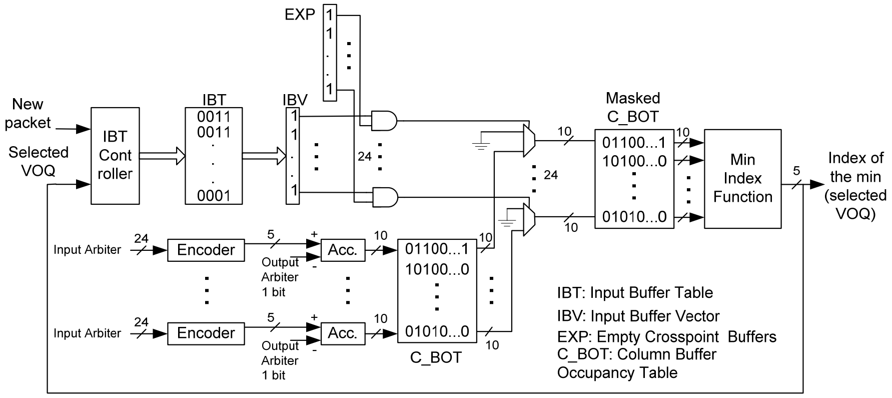


Figure 3.7: β-MCBF input arbiter micro-architecture.

two multiplexors are forwarded to the next stage. Please note that, for the first stage, the index multiplexer (upper multiplexer) is not required, since there are no partial input indexes. Also, for the last stage, the lower multiplexer is discarded since the output of the overall block is only the index of the minimum element. Additionally, if the two inputs are equal, we can design the circuit to choose the input with the lower index to be selected[4]. This means that ties are broken based on the lowest port number first (in the presence of conflicts) and this conforms to the desired result. The same applies for finding the maximum element, for the output arbiter.

The micro-architecture of the β-MCBF output scheduler is depicted in Figure 3.9. While the dynamics of the output arbiter are opposite to the input arbiter, most of the components used by each are similar. Recall that each CXPB, used by the input arbiter, can receive up to 24 new packets and can discharge at most one packet every scheduling cycle. Oppositely the same applies to the output arbiter, in that each LXPB can receive at most one packet and can discharge up to 24 packets each scheduling cycle. For this reason we employed the same encoder and accumulator for the design of the output arbiter. The accumulator keeps track of the number of cells at each LXPB. The Row Buffers Occupancy table R_BOT, similar to the C_BOT, is used to store the number of queued cells per LXPB. Similar to the α-MCBF output scheduler, a 24-bit vector named the Non-Empty Crosspoint buffers (NEXP) is used to represent the non-empty internal crosspoint buffers, XP, belonging

---

[4] We can achieve this by always inverting the input with higher index (Min1 in Figure 3.8).

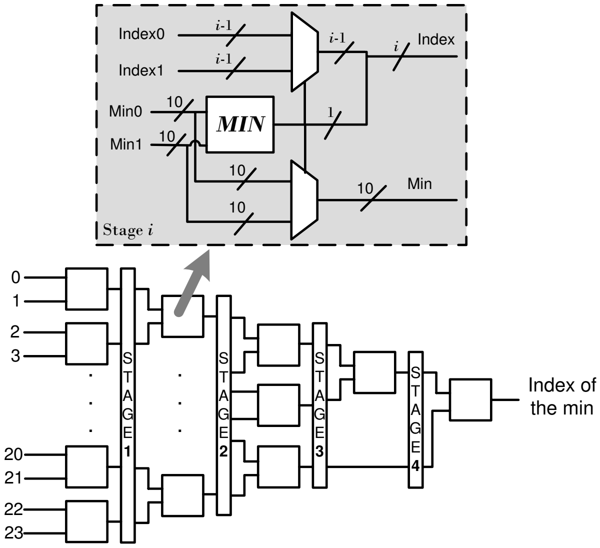


Figure 3.8: The minimum index function.

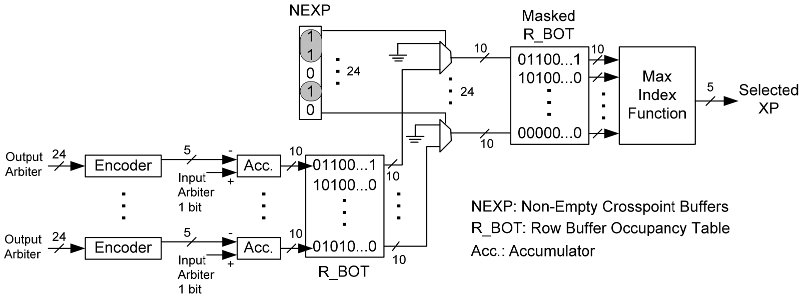


Figure 3.9: β-MCBF output arbiter micro-architecture.

to a LXPB. NEXP is used as a mask for the R_BOT to create a masked BOT (M_BOT). The rows of the M_BOT are used as inputs for the *Max Index Function* block. The max index function is similar to the one shown in Figure 3.8 and the carry-propagation circuitry is used to find the index of the maximum internal buffer, XP, selected by the output arbiter.

3.4.3 Extension to Wider Range of Algorithms

Our design can be extended to implement a wider range of scheduling algorithms. For example, we can use our design and embed the Longest Queue First-Round Robin (LQF-RR) algorithm [66] or the Oldest Cell First (OCF-OCF) algorithm [60] in a similar manner as MCBF. We expect all these algorithms to be easily mapped inside the buffered crossbar chip because of their similar hardware requirements, as weighted schemes. In the case of the LQF-RR algorithm, the input scheduler (LQF) can be embedded within the buffered crossbar chip by using the IBT table with a slight modification. The LQF algorithm gives priority to the input VOQ with the highest number of packets (cells). As mentioned in the previous section, the IBT table uses 15 bits to represent the occupancy (length) of each input VOQ in number of cells. However, in practice, an input line card should hold up to 100 ms worth of packets [78]. At 10 Gbps, the buffer requirement per line card would be 125 MB. For a 24×24 CICQ switch and 64 B cells, every VOQ is approximately 5.2 MB (or 81.25 Kilo cells). This translates to an IBT entry of 17 bits, which can be easily modified. Besides the IBT modification, LQF would not require much of a difference compared to the MCBF scheme. The same modification can be applied to the OCF algorithm in order to represent the arrival time of cells to the VOQs instead of queue length.

3.5 Implementation and Performance Results

This section presents the hardware implementation results in terms of timing and area of our design. In addition to the implementation results, we also conduct a simulation study (Section 3.5.2) to evaluate the performance of our switching architecture in terms of average cell delay, throughput and input buffer requirements. We study the performance of the MCBF algorithm for the specific design presented above (the FPGA implementation of the 24 ports switch) as well as for more generic switching systems.

3.5.1 Implementation Results

In this section, we present the implementation results in terms of timing and area. The design is mapped to a Xilinx Virtex4-FX device and the results are presented after place and route. The arrival rate of 64-Bytes packets at OC-192 line rate is one packet every 51.2 ns. The Rocket IO transceiver can

Table 3.2: α-MCBF area results.

Module	Slices	Instances	Total Slices
Input Arbiter	1197	24	28728
Output Arbiter	632	24	15168
Column BOT	28	24	672
Row BOT	28	24	672
BRAMS	19	529	10051
Total Slices			55291

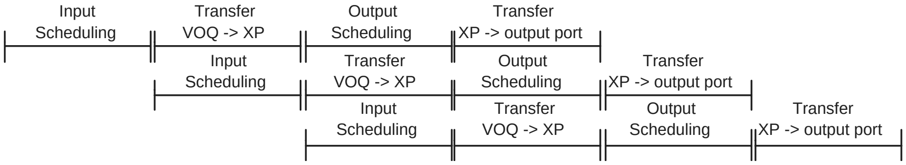


Figure 3.10: Packets flow.

be configured to de-serialize the input into a 64-bit wide bus at 156 MHz (64x156x10^6 $\approx$ 10x10^9 = 10 Gbps). The clock cycle time is 6.4 ns, hence each packet can be transferred in 8 cycles (6.4x8=51.2 ns). The input arbiter for both α-MCBF and β-MCBF have been designed to work at the same clock frequency and each has been divided into 8 cycles. Hence, while a packet is being transferred from the input card to the crossbar switch, a new queue is selected by the input arbiter, as depicted in Figure 3.10. The critical path of the first design is the Priority Encoder used to forward the selected vector to the PPE. This module is made of a 24-to-1 24-bit multiplexer, checking 24 bits to decide which vector is selected. For the β-MCBF design, the function used to find the index of the minimum dominates the timing of the circuit and was segmented in 5 clock cycles.

The area results of the implementation into a Virtex4-FX140 of the α-MCBF and the β-MCBF schemes are depicted in Table 3.2 and Table 3.3, respectively. We can see that the area, in number of slices, required for the β-MCBF is far less than that of the α-MCBF implementation. The allocation of the resources, of both implementations is shown in Table 3.4. While both schemes use the same interfaces in terms of Rocket IOs and pin count, their main difference lies in the number of slices required for each of them. The α-MCBF implementation requires 87.3% of the available slices on the FPGA device, while the β-MCBF implementation requires only approximately half the slices of

Table 3.3: β-MCBF area results.

Module	*Slices*	*Instances*	*Total Slices*
Input Arbiter	676	24	16224
Output Arbiter	300	24	7200
Column BOT	51	24	1224
Row BOT	51	24	1224
BRAMS	19	529	10051
Total Slices			35923

Table 3.4: Percentage of resource allocation.

Module		*Instances Used*	*Available*	*Percentage*
BRAMs		529	552	95.83
RocketIO		24	24	100 %
Slices	α-MCBF	55291	63168	87.3 %
	β-MCBF	35923	63168	56.8 %
Pins		288	896	32.14 %

the device (56.8%). Please note that the number of crossbar buffers (BRAMs) is 23×23 =529 and not 24×24, since the transmission of cells from the same input and output indexes is not required to go through the crossbar fabric.

3.5.2 Performance Results

This section studies the performance results of the MCBF set of algorithms and compares it to alternative algorithms. Because MCBF is a weight-based algorithm, it is compared to LQF-RR and OCF-OCF since they also use weight for their arbitration process. The performance study is structured in two parts. The first part studies the performance of the MCBF and its two implementations for the specific design proposed in Section 3.4. The second part presents the MCBF performance for a general $N \times N$ CICQ switching system. The performance metrics studied here are the average cell latency, throughput and input queues occupancies. Simulations run for 1 million time slots and statistics are gathered when fourth of the total simulation length has elapsed. The analysis is carried under Bernoulli uniform traffic, Bursty uniform traffic, Unbalanced traffic and Diagonal traffic. More details about the simulation environment and

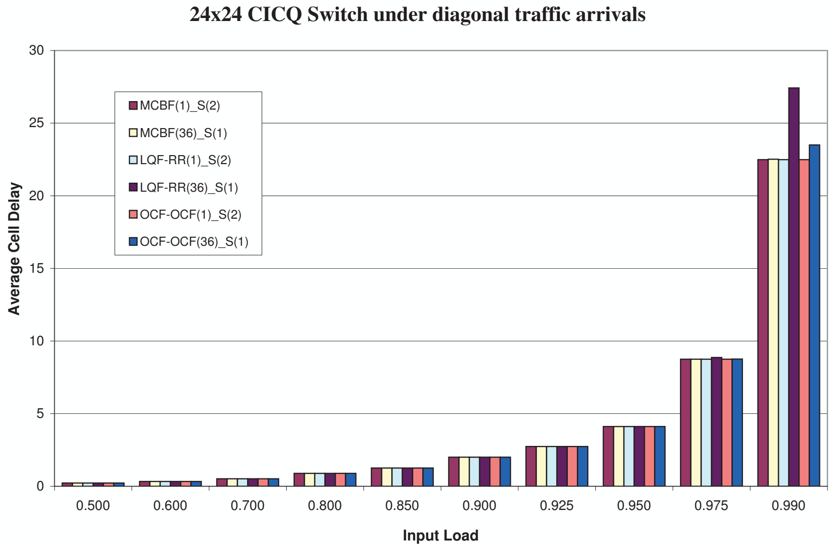


Figure 3.11: Average delay comparison between using a speedup of 2 and internal buffer size per cross point of 36 cells, under diagonal traffic.

the traffic scenarios used are defined in Appendix B.2.

MCBF Performance: The designed 24×24 CICQ Switch

In this section we present the performance of MCBF of our proposed architecture and specific design of a 24×24 CICQ Switch. The 18 Kbit Block RAMs (BRAMs) of the FPGA device have been used as internal crosspoint buffers meaning that every crosspoint buffer can hold up to 36 cells (64 Bytes each). We also compared MCBF to the two approximations we implemented (α-MCBF and β-MCBF).

As depicted in Figure 3.11 and 3.12, the average delay of the algorithms used in our design is closely comparable to the average delay of these algorithms when running on the same switch but with just one cell as internal buffer size and a speed up of two[5]. Note that MCBF(1)_S(2) refers to the CICQ switch running the MCBF scheduler, using an internal crosspoint buffer size of just 1 cell and running at a speedup of 2, while MCBF(36)_S(1) refers to the same system but with an internal crosspoint buffer size of 36 cells and a speedup of just 1. This

---

[5] A speedup of two means that the crossbar fabric runs twice as fast as the input/output ports.

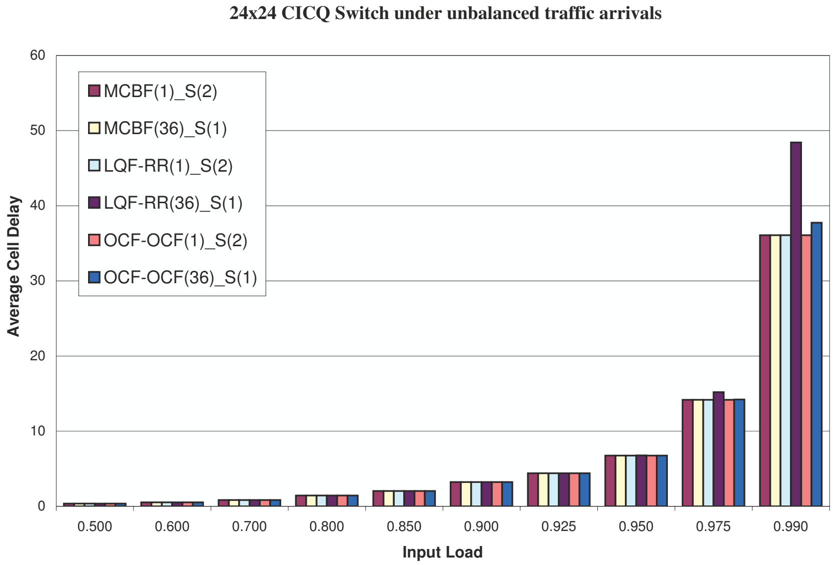


Figure 3.12: Average delay comparison between using a speedup of 2 and internal buffer size per cross point of 36 cells, under non-uniform unbalanced traffic, ω = 0.5.

result suggests that we can trade fabric speed for internal crosspoint memory size. However, as mentioned before, our FPGA device contains the BRAMS that can be directly used as internal buffers. Therefore, we achieved delay performance equal to that of a speedup of two. Although it is quite expected to have such good performance, since the internal buffer size is big enough making it similar to OQ switch, the idea here is to assess the efficiency of MCBF as compared to other algorithms. We can see that MCBF has a shorter delay than the other two due to its matched pair of input/output scheduling as well as its balanced use of the internal buffers.

In the remaining figures, we studied the performance of the MCBF and its variations under different internal buffer sizes. Note that MCBF(i) (respectively α-MCBF(i) and β-MCBF(i)) refers to MCBF running on an CICQ switch with an internal crosspoint buffer size of i cells where $\{i = 1, 4, 8, 36\}$. We compared the α-MCBF and β-MCBF variations to the original MCBF scheme under both the diagonal and unbalanced traffic models. Figure 3.13 depicts the average cell delay of the MCBF and α-MCBF schemes with different internal buffer size under the diagonal traffic. When the internal buffer size is equal to one cell per crosspoint, both algorithms have the same average delay because

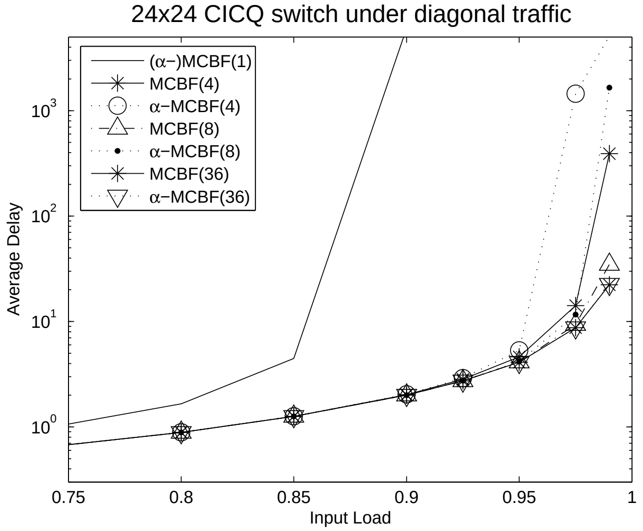


Figure 3.13: Delay performance comparison between the MCBF and α-MCBF schedulers with different internal buffer sizes under Diagonal traffic.

their merit functions are the same under these settings. The same performance is achieved when the XP size is 36 cells, and these settings correspond to our target system. However, when the internal buffer size is between 1 and 8 cells, the MCBF delay is better than that of α-MCBF. The reason for this is because the merit function used by α-MCBF just approximates the occupancy of the internal buffers, whereas the original MCBF uses the exact occupancy of the internal buffers.

As for the β-MCBF scheme, Figure 3.14, its average delay under the diagonal traffic model is consistently better than the original MCBF scheme. This is because of the different priorities policies used by each of the schedulers. β-MCBf breaks ties based on the lowest port first, although this causes some sort of unfairness. However, due to the very unlikely event of having two or more cells with the *same* highest priority in our system, the highest priority pointer can indeed be omitted.

The same results are found with the unbalanced traffic pattern. Figure 3.15 and Figure 3.16 depict the average delay performance of α-MCBF and β-MCBF compared to the original MCBF scheme. Based on these results, and the implementation results, it is clear that the β-MCBF is the best choice due to its lower hardware cost as well as its shorter average delay performance. One

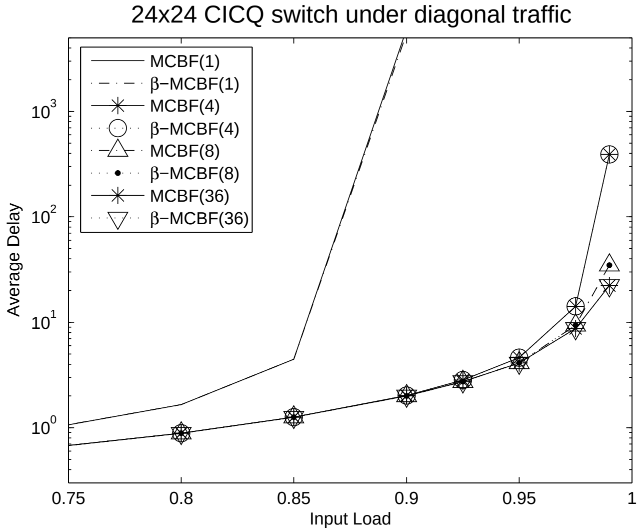


Figure 3.14: Delay performance comparison between the MCBF and β-MCBF schedulers with different internal buffer sizes under Diagonal traffic.

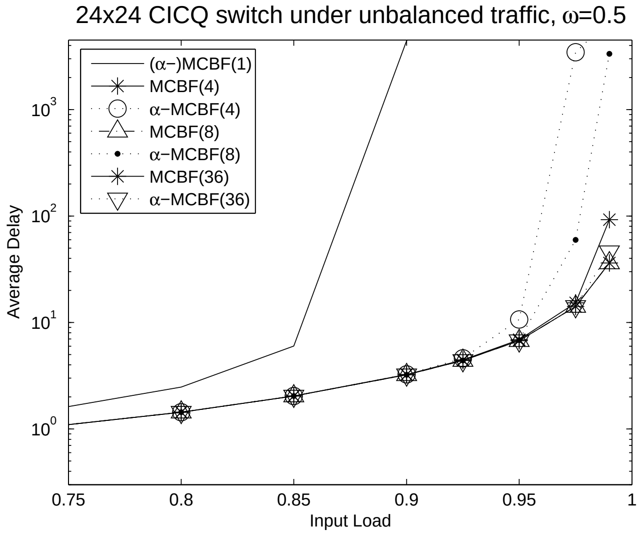


Figure 3.15: Delay performance comparison between the MCBF and α-MCBF schedulers with different internal buffer sizes under Unbalanced traffic.

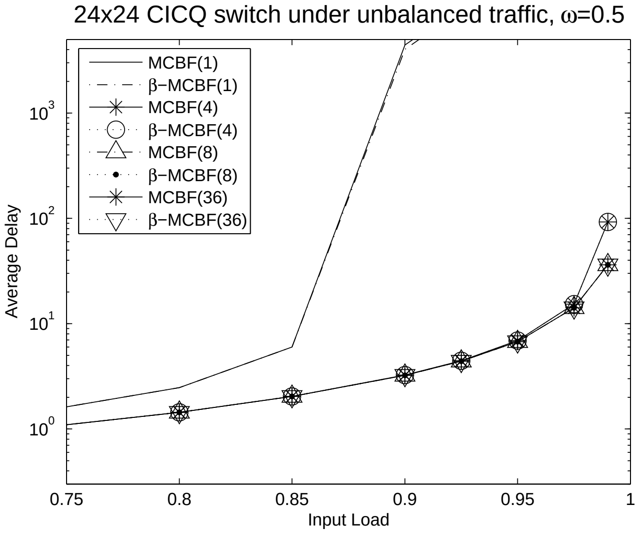


Figure 3.16: Delay performance comparison between the MCBF and β-MCBF schedulers with different internal buffer sizes under Unbalanced traffic.

more interesting result is, we found that by allowing enough buffering for the internal cross points, the input line card size is no longer required to be as large. The experimental results (not shown) suggested that our CICQ switch running any of the three algorithms mentioned above and using 36 cells per cross point do not require a line card buffer of more than 16 KB.

In traditional scheduler design, where the input and output schedulers are implemented outside the crossbar fabric chip, it is hard to take full advantage of the internal buffer information. This is because, as the internal buffers size increases, extra control pins are required for flow control to capture the exact sate of the internal buffers. Our design, however, overcomes this constraint by avoiding the requirement for extra pins irrespective of the internal buffers size. Because the schedulers are embedded within the buffered crossbar fabric, there are no restrictions on the internal crosspoint buffer size as the flow control is performed *locally* (on the same chip). This would not have been possible if the schedulers were taking place outside the crossbar fabric chip. To show the benefit of our design as compared to traditional implementations, Figure 3.17 shows how the MCBF delay improves dramatically as the internal buffer size increases. We can see that a small increase in the XP size results is far shorter average cell delay, as in MCBF(1) and MCBF(8) (for XP size of 1 cell and 8 cells respectively).

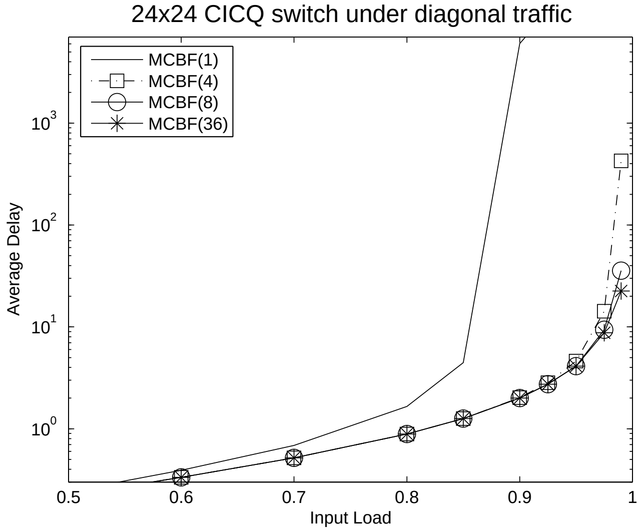


Figure 3.17: Delay performance of the MCBF scheduler with different internal buffer sizes under Diagonal traffic.

MCBF Performance: $N \times N$ CICQ Switch

We analyzed the performance of MCBF with LQF-RR and OCF-OCF for two switch sizes of 16×16 and 32×32 and different internal buffers sizes, respectively. The performance analysis is carried under various traffic models as defined in Appendix B.2.

Uniform Traffic

Figure 3.18 depicts the average cell delay performance under bursty uniform traffic with burst lengths (b) equal to 1, 10, 50 and 100 respectively. Under heavy loads, MCBF exhibits the shortest delay of all the three schemes presented. Note that when the burst length equals 1, the traffic is Bernoulli uniform. At 99% load and burst length of 10, MCBF has an average queueing delay less than 80% of that of LQF-RR.

As for the L^2 norm vector (see Appendix B.3) representing the occupancies of the input VOQs, illustrated in Figure 3.19, MCBF has surprisingly the best performance amongst all despite the fact that it maintains no state information about the input VOQs, neither does it use their occupancies for scheduling decisions.

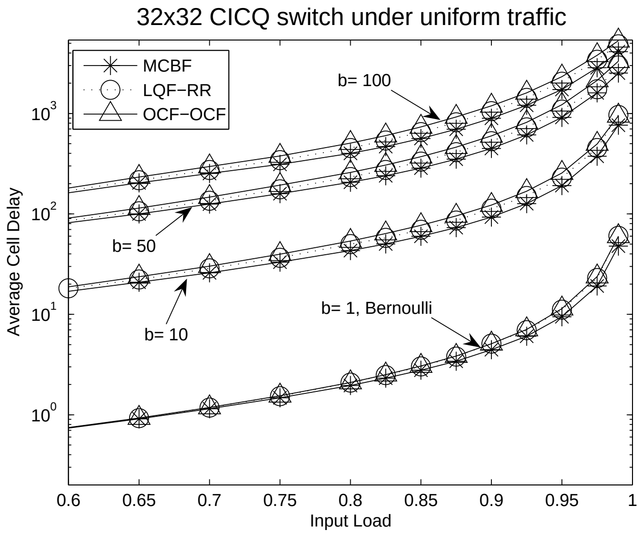


Figure 3.18: Average cell delay performance under uniform traffic.

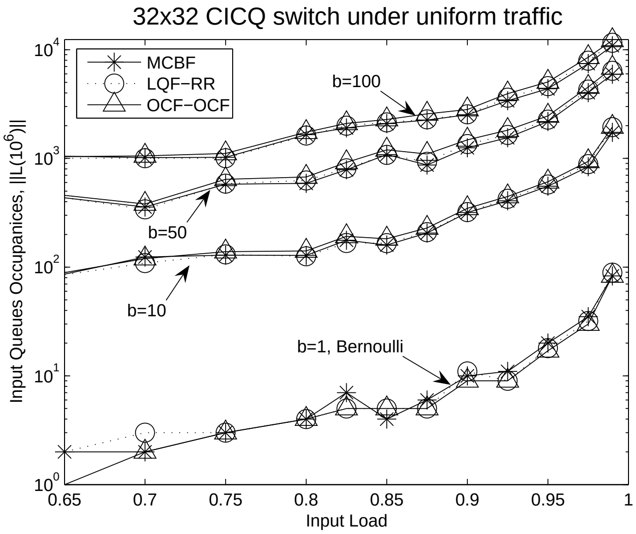


Figure 3.19: The Input queues occupancies under uniform traffic.

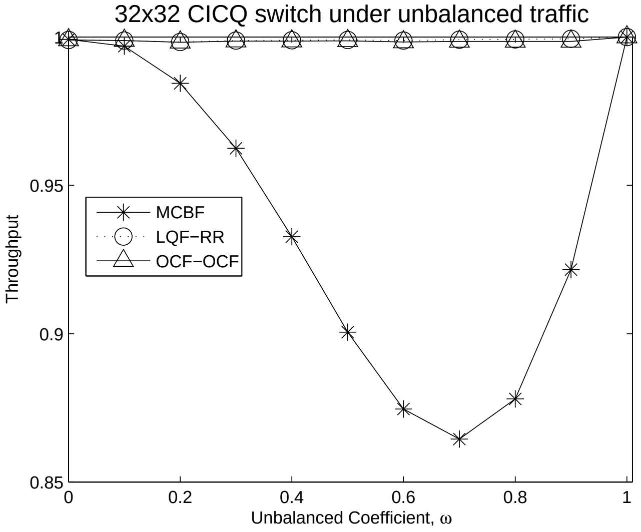


Figure 3.20: Stability under unbalanced traffic with internal buffer size of 1 cell.

Non Uniform Traffic

The remainder of the simulation is carried under non uniform traffic models, where we wanted to test the stability of MCBF when using internal buffers of small sizes. Figure 3.20 depicts the stability performance of MCBF as compared to LQF-RR and OCF respectively. We can see that MCBF no longer exhibits the best performance. This is due to the small size of the internal buffers. The MCBF scheduling is based fully on the internal buffers occupancies, and setting the XP size to be 1 cell appears to be not enough for an efficient MCBF scheduling decision.

It is natural to ask the question as to what is the minimum internal buffer size for which MCBF would make efficient scheduling choices. For this, we set the XP size to be 4 and 8 cells respectively and observed the stability of MCBF, as depicted in Figure 3.21 and Figure 3.22. When XP=4 cells, the MCBF performance increases from 87%, for XP=1, to more than 98%. The MCBF has a comparable performance to that of LQF-RR. Setting XP=8 results in MCBF outperforming the other algorithms by having the highest throughput. This suggests that the optimal internal size for MCBF can be somewhere between 4 and 8 cells per crosspoint. This result is also endorsed from the average cell delay standpoint. Figure 3.23 depicts the average cell delay of a 16×16

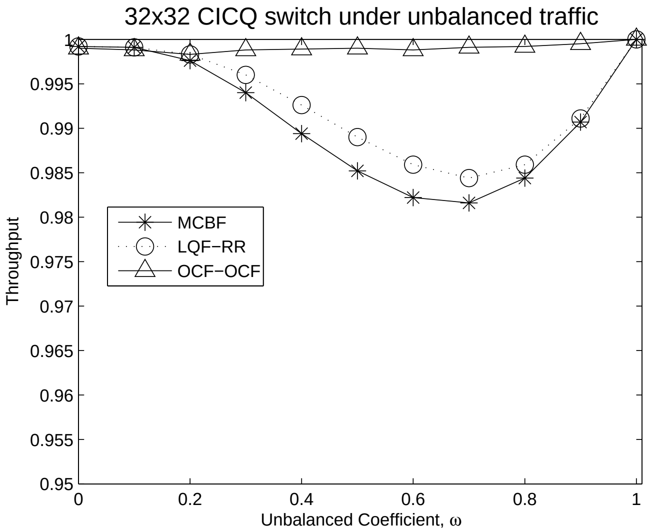


Figure 3.21: Stability under unbalanced traffic with internal buffer size of 4 cells.

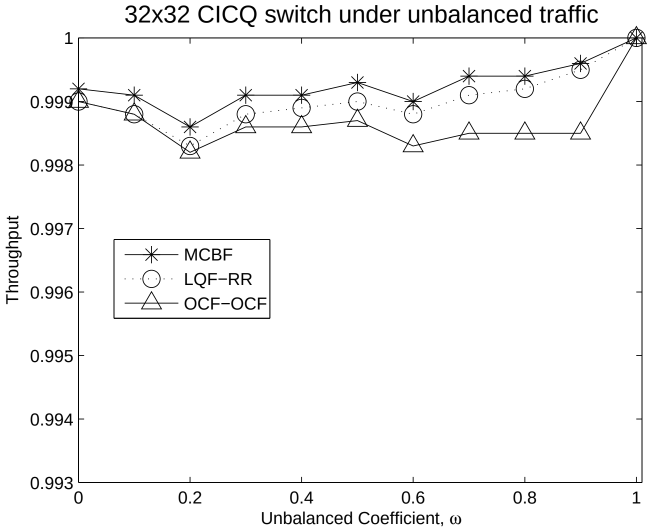


Figure 3.22: Stability under unbalanced traffic with internal buffer size of 8 cells.

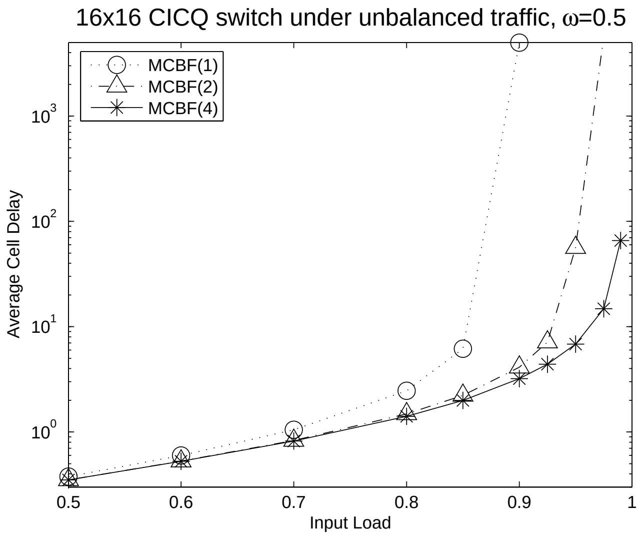


Figure 3.23: Average cell latency for different internal buffer settings under unbalanced traffic, $\omega = 0.5$.

CICQ running MCBF with unbalanced traffic arrivals and an unbalanced coefficient ω = 50%. As we can see from the figure, the cell latency improves substantially when using 4 cells per crosspoint instead of just 1 cell.

3.6 Summary

This chapter proposes a new trend in designing scheduling algorithms. Instead of being distributed over the input and output line cards, the new design embeds all the schedulers within the buffered crossbar fabric chip. Placing the schedulers inside the crossbar chip has the benefit of optimizing the flow control mechanism. For a 32×32 switching system, our embedded CICQ switching architecture achieves up to 70% saving of chip I/O flow control pins when compared to existing CICQ switch architectures. When the schedulers are embedded within the buffered crossbar fabric chip they have faster and cheaper access to resources, with the further benefit of saving area on the input and output line cards. We proposed a new class of scheduling algorithms named the Most Critical Buffer First (MCBF). Unlike alternative algorithms, the MCBF scheduling decision is fully based on the internal buffers.

To show the feasibility of our design, we proposed two implementations of the

MCBF scheduling scheme and showed that each can fit within the crossbar fabric chip. We studied the trade offs between both implementations in terms of hardware cost and performance. Our design shows that a 24×24 CICQ switch with embedded schedulers running a 10 Gbps port speed and a clock cycle time of 6.4 ns can be readily implemented within a single FPGA chip.

The MCBF algorithm has been shown to exhibit high performance and outperform state of the art algorithms. Performance results indicate that an internal buffer size of less than 8 cells is sufficient for MCBF to achieve high performance. MCBF is, however, not always stable. If the internal buffer size is just 1 cell, MCBF cannot achieve high throughput under non uniform traffic. In order to provide throughput guarantees, we either need more sophisticated algorithms or we need to use speedup. We derived a theoretical study and showed that our proposed CICQ switch employing appropriate embedded scheduling and running a speedup of two can emulate an ideal FIFO output queued switch. We divert our study to Appendix A.

While it is possible to achieve high performance and even performance guarantees with distributed and simple algorithms, the traffic we envisioned is limited to unicast. It is desirable to achieve the same, or similar, goals for broader class of traffic such as multicast. In the following chapter, we will address the problem of scheduling multicast traffic flows.

Chapter 4

Scheduling Multicast Traffic

The tremendous growth of the Internet coupled with newly emerging applications has created a vital need for multicast traffic support by backbone routers and switches. In this chapter, we study the multicast scheduling problem in buffered crossbar (CICQ) switches. We propose a novel CICQ switching architecture with one multicast FIFO queue per input port. We describe a simple scheduling algorithm for this architecture. Our algorithm, named the Multicast Round-Robin (MXRR) scheduling is shown to outperform alternative algorithms. We extend our study and address the CICQ switching architecture with multiple input multicast FIFO queues per input. We devise a cell placement algorithm, named Modulo, that maps incoming traffic to the input multicast queues faster and more efficiently than existing algorithms. We also extend the MXRR scheduling algorithm to schedule cells in the presence of multiple multicast queues per input port of the switch. Simulation results show that we can trade the size of the internal buffers for the number of input multicast queues. Hence, affording a switch designer the choice between the cost and complexity of the switch core and the scheduler.

4.1 Introduction

Traditionally, network nodes (IP routers, ATM switches, Ethernet switches) were designed for point-to-point communication (unicast). However, the variety of services on the Internet today has resulted in the emergence of new applications such as teleconferencing, distance learning, IPTV etc. These new applications have led to a high demand for high-speed switches/routers capable of handling point-to-multipoint communication (multicast). Several architectures for efficient multicast support have been investigated and implemented. These architectures can be classified based on various factors such as queueing schemes, scheduling algorithms and switch fabric type. The crossbar-based architecture [83] is widely considered the most suitable switching architecture due to its low cost, scalability and more importantly its *intrinsic multicast capabilities* [84].

Unlike unicast traffic, where a packet (cell) at an input port is destined to only one output port, a multicast cell queued at an input port can have 1 or more destination output ports known as its fanout set. Although different architectures have been proposed for multicast traffic handling [85] which are based on copy networks, in this chapter we consider the crossbar-based switching architecture due to its architectural intrinsic multicast capabilities. There has been significant research work on multicast scheduling in the literature, most of which is based on a multicast FIFO queue architecture [83]. However, because of a similar HoL blocking problem as for the unicast traffic, the performance is low. Avoiding the HoL problem in this case requires a FIFO queue for every fanout set per input. This implies maintaining up to $2^N - 1$ separate FIFO queues per input, where N is the number of ports of the switch. This architecture is known as the multicast VOQ (MC-VOQ) switching architecture [86]. This architecture is clearly impractical for even small sized switches due to the large number of queues required. As a compromise, researchers have proposed to use a small number of queues, k ($1 \leq k \ll 2^N - 1$) per input [87].

This chapter focusses on the multicast scheduling problem in Combined Input and Crossbar Queued (CICQ) switches. We describe a set of multicast scheduling algorithms along with appropriate architectures. In particular, we propose the following:

- A novel CICQ switching architecture, where there is one multicast FIFO queue per input port of the CICQ switch. We describe a simple scheduling algorithm for this architecture. Our algorithm, named the Multicast Round Robin (MXRR) is based on FIFO scheduling as its input scheduling and a Round Robin scheduling for its output scheduling.
- We studied the multicast problem in CICQ switches with multiple multicast FIFO queues per input port. We devise a cell placement algorithm, named *Modulo*, that efficiently maps incoming traffic to the input multicast queues. We also extend the MXRR scheduling algorithm to schedule cells in the presence of multiple multicast queues per input port of the switch. We refer to this algorithm as MXRR_k.

The experimental results showed the superiority of the CICQ architecture compared with its bufferless predecessor and its high capability to support multicast traffic. Although simple in hardware, the MXRR was shown to outperform state of the art alternative algorithms [88]. The same results apply for the MXRR_k scheduling algorithm. We compared the Modulo cell assignment scheme to the Majority scheme [87] and showed that our scheme assigns arriving traffic to the input multicast queues more efficiently and quickly than Majority, while requiring low hardware cost. Additionally, the experimental results showed an interesting trade off between the number of input multicast queues and the size of the internal buffers. The reduction in the number of input queues, at the expense of adding small extra internal buffering, is very important not only because it results in better performance, but also because it greatly reduces the complexity of the scheduler in terms of information exchange, resulting in faster and more scalable switching. This affords a switch designer the choice between the cost and complexity of the buffered crossbar core on one hand, and the complexity and speed of the scheduling algorithm on the other.

The remainder of this chapter is structured as follows: Section 4.2 introduces the multicast scheduling problem and presents background knowledge and related work. We review existing multicast switching architectures and discuss their scheduling. In Section 4.3, we describe our proposed multicast FIFO queue CICQ switching architecture. We describe the MXRR algorithm and present its properties. Section 4.4 describes the CICQ switching architecture with multiple input multicast queues. We describe a novel, simple and efficient cell placement algorithm, named Modulo. We also propose an extended version of the MXRR algorithm, that we have named MXRR_k. In Section 4.5, we present and analyze the performance of our devised architecture and scheduling algorithms and compare them to existing solutions. Finally, Section 4.6 concludes the chapter.

4.2 The Multicasting Problem

Multicast traffic handling, in its simplest form, is the capability of a router to transfer the same data (a cell) to multiple output ports at minimum cost in terms of data processing and time. This is important due to the growing volume of multicast traffic on the Internet (audio, video, IPTV, etc.). Consider the example in Figure 4.1, and assume that the 3 hosts connected to router R2 are receiving the same multimedia content from the server. If the server sends

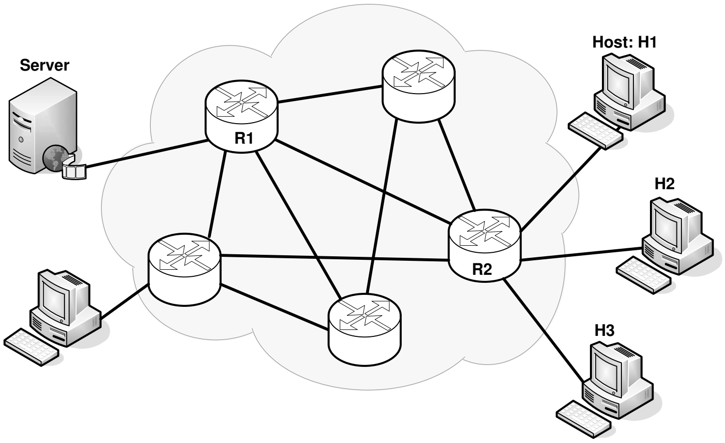


Figure 4.1: Multicast traffic support in core routers.

the same message to hosts H1, H2, and H3, it either sends the same message 3 times (one per destination) or it can send the message only once over routers R1 and R2. Upon reaching R2, the message gets split into 3 copies, one copy per destination host. Clearly, the latter case is a better choice since it optimizes the network resources and the time taken for the hosts to receive the data. To achieve this, routers R1 and R2 must be designed to support multicast traffic.

The set of destination output ports of a multicast cell is known as its fanout set. If we consider an $N \times M$ router with multicast capabilities, a multicast cell arriving at any of the N input ports can have any set of destinations between 1 and M. In order to avoid the HoL problem, the router must maintain up to $2^M - 1$ separate FIFO queues per input in order to cover all possible fanout set configurations (see Section 4.4.1). This architecture is known as the multicast VOQ (MC-VOQ) [86]. Because of the huge number of queues maintained at each input and the need for extensive information exchange in order to schedule the traffic, this architecture is considered impractical. Instead, researchers have implemented just one FIFO queue per input. While using just one queue per input is practical, it has poor performance due to the HoL problem. Another solution was to maintain a small number, k, of queues per input for multicast traffic. This proved a good compromise to achieve good performance while maintaining affordable hardware requirements. Because k is much smaller than $2^M - 1$, cells with different fanout sets may have to be queued in the same input queue. This mapping is known as the multicast cell placement policy and will be discussed in more detail in Section 4.4.1.

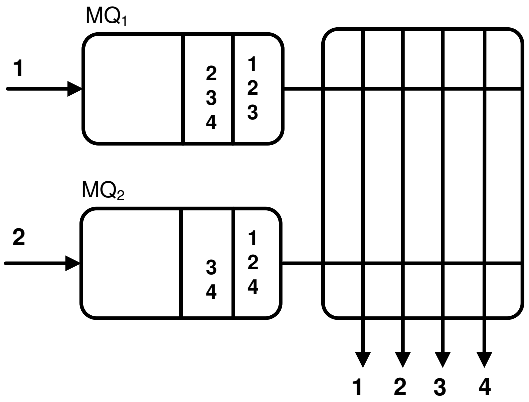


Figure 4.2: A 2×4 FIFO multicast crossbar switch.

4.2.1 The Multicast FIFO Architecture

If we consider that router R2 (in Figure 4.1) uses just one FIFO queue per input for multicast traffic, its architecture can be described as depicted in Figure 4.2. By considering that the crossbar fabric operates at the same speed as the external lines, at each time slot every input can send at most one cell and every output can receive at most one cell. Because of the intrinsic multicast capabilities of the crossbar fabric, a cell can be sent to all its destinations at the cost of one by simply closing those crosspoints corresponding to the cell destination output ports provided that these outputs are ready (available) to receive cells.

Subject to output availability and the scheduling algorithm used, a cell may not reach all its destinations, indicated by its fanout set, during one time slot. There are two known service disciplines used to deal with this situation [83]. The first discipline is known as *no fanout splitting* and the latter is known as *fanout splitting*. When no fanout splitting discipline is used, a cell must traverse the crossbar fabric only once. Meaning that a cell is switched to its output destination ports if and only if all its destination outputs are available at the same time. If one, or more, of the output destinations are busy, the cell loses contention and all of its copies remain in the input port. If we consider the no fanout splitting discipline in Figure 4.2, then only one of the two HoL cells of queues MQ_1 and MQ_2 will be switched out but *not both*. The reason for this is that both cells have output ports 1 and 2 in their fanout sets, and knowing that an output port can receive at most one cell and the no fanout splitting discipline does not allow partial cell switching, this results in only one cell of the two being eligible for transfer. The no fanout splitting discipline is known to be bandwidth efficient and easy to implement, however it achieves low throughput because it is not work conserving[1]. This can be seen from the example above as either output 3 or output 4 will receive a cell, but not both depending on which MQ has been selected.

When, however, fanout splitting discipline is used, a cell can be *partially* sent to its destination output ports over many time slots. Copies of the cell that are not switched, due to output contention, during one time slot continue competing for transfer during the following time slot(s). The flexibility of allowing partial cell transfer is achieved through a little increase in implementation complexity, however it provides higher throughput because it is work conserving [89]. In this chapter, we consider fanout splitting. If we consider the example of Figure 4.2 again and assuming a fanout splitting discipline is used, then both the HoL cells of MQ_1 and MQ_2 can send copies to a subset of their output ports. Output 3 and 4 are receiving one cell each and therefore both copies destined to them, in the input queues, are transferred with no contention. Additionally, both HoL cells of MQ_1 and MQ_2 have cells destined to outputs 1 and 2. However, we know that each output can receive at most one cell at a time. Therefore, at the end of the time slot, we will have remaining cells for output ports 1 and 2. These remaining cells are referred to as the *residue*.

Depending on the policy used, the residue can either be *concentrated* on the input ports or it can be *distributed* over the input ports. As defined in [83], the residue is the set of cells remaining at the HoL of the input queues after losing contention for the output ports at the end of each time slot. In the example of Figure 4.2, the residue is $\{1, 2\}$. A concentrating policy is one that leaves the residue on the minimum number of input ports. If we consider a concentrating policy in Figure 4.2, the residue will be left (concentrated) on either MQ_1 or on MQ_2 but not on both. On the other hand, a distributing policy is one that leaves the residue on the maximum number of input ports. Using a distributing policy in Figure 4.2 would result in the residue being distributed over MQ_1 and MQ_2 but not on one queue only.

4.2.2 Algorithms For The Multicast FIFO Architecture

Several algorithms have been proposed for this architecture, all of which were designed for the bufferless crossbar fabric switches.

- *The Concentrate Algorithm:* As the name indicates, the concentrate al-

---

[1] A work conserving policy ensures that an output port is never idle so long as there are cells destined to it in the input ports

gorithm [83] always concentrates the residue onto as few inputs as possible. The purpose of this algorithm is to provide a basis for evaluating the performances of other algorithms, since it achieves high throughput for the FIFO queue structure. However, this algorithm does not meet the fairness requirement due to the starvation problem it creates. The Concentrate algorithm is not considered a practical algorithm. It requires up to M iterations per cell time to complete, which makes it difficult to implement at high speed.

- *The mRRM Algorithm:* The Multicast Round-Robin Matching (mRRM) was proposed by [88]. A single round-robin pointer is collectively maintained by all of the outputs. Each output selects the next input that requests it at, or after, the pointer. At the end of the packet time, the pointer is moved to one position beyond the first input that is served. Designed to be simple to implement in hardware, mRRM tends to concentrate the selection onto a small number of inputs, yet maintains fairness.
- *The TATRA Algorithm:* The general multicast scheduling problem can be mapped onto a variation of the popular block-packing game Tetris. TATRA is based on the Tetris model and was first introduced in [88]. TATRA has the properties of guaranteeing at least one input packet is discharged within each packet time, and also concentrates the residue. Designed to approximate the concentrate algorithm with less complexity, unfortunately TATRA is a complex algorithm since it cannot be parallelized. Moreover, TATRA treats all inputs uniformly which is of no value when the inputs are non-uniformly loaded or when some inputs request a higher priority.

4.2.3 The Multicast k FIFO Queues Architecture

Due to the impracticality of the MC-VOQ switching architecture [86] and to the low performance of the multicast FIFO architecture, a good compromise is to use the multicast k FIFO queues architecture. It is a queueing architecture with a small number of input multicast FIFO queues per input ($1 \leq k \ll 2^M - 1$). This queueing architecture has been studied in the context of bufferless crossbar switches [90] [91] as well as CICQ switches [92]. Figure 4.5 depicts the multicast k FIFO architecture for an $N \times M$ buffered crossbar switch. Recent work has showed that, similarly to IQ architectures, CICQ switches with arbitrarily large number of ports may also suffer significant throughput degradation for multicast traffic [92].

Because the number of multicast queues maintained at each input is significantly smaller than the cardinality of the set of all fanout sets, incoming cells must be inserted in appropriate queues, MQs, by following certain criteria [87]. This is known as the cell placement strategy and will be discussed in Section 4.4.1.

4.2.4 Algorithms For The Multicast k FIFO Queues Architecture

The low performance and high complexity of the multicast FIFO architecture have stressed the need for the multicast k FIFO queues architecture and many scheduling algorithms have been proposed. These algorithms have been designed for the bufferless as well as for the buffered crossbar architectures.

- *Bufferless Crossbar Based Algorithms*: Algorithms for this architecture include the random scheduler (RS), the Greedy Scheduler (GS) and the Greedy Min-Split Scheduler (GMSS) [90]. The first algorithm (RS) makes decisions randomly among the input and output ports. In addition to its costly hardware requirement, this scheme has poor performance as it leaves idle outputs due to the contention effect. The second algorithm, GS, tries to overcome the shortcoming of RS. It assigns weights to the queues such as queue length and then makes its selection based on weight ordering. The third algorithm is also a weighted algorithm and tries to combine the advantages of the previous. Because of the required sorting process, the implementation of this algorithm is not trivial and prevents it from running at high rates.

- *Buffered Crossbar Based Algorithms:* A group of scheduling algorithms has recently been proposed for the multicast k FIFO queues architecture designed for the CICQ switching architecture [93]. These algorithms were proposed along with a class of cell placement schemes. The input arbitration was based on some policies such as giving preference to HoL cells that would result in the minimum left residue. Another input scheduling algorithm was based on selecting the cell with the maximum number of reachable destinations first. A third policy is to give preference to cells with the maximum service ratio, defined as the number of reachable destination outputs divided by the fanout number of a cell. The output arbitration was based on Round Robin (RR) and Longest Queue First (LQF).

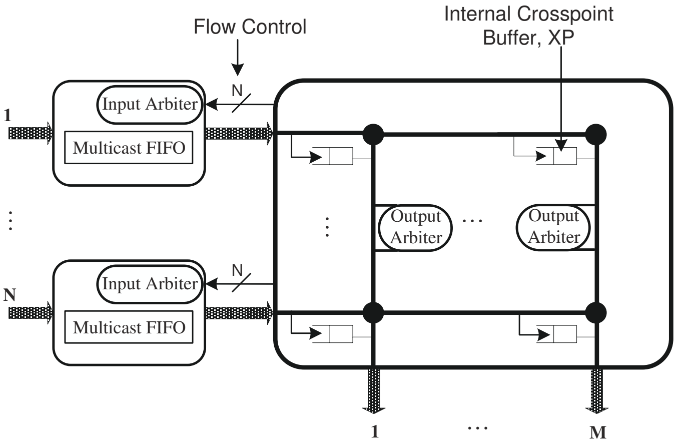


Figure 4.3: $N \times M$ multicast CICQ Switch.

As a summary of the related work, we argue that each of the above presented schemes tries to address some issues but fails to meet other vital requirements. So far, none of these algorithms have proven simultaneously efficient in terms of high throughput, practical in terms of implementation complexity or fair with respect to the input FIFO queues. In the following section, we propose our new architecture along with a scheduling scheme that meets all these requirements.

4.3 The Multicast CICQ Switching Architecture

Our choice of the multicast CICQ crossbar switch architecture is motivated by the fact that this architecture has key advantages in simplifying the scheduling process. The presence of internal buffers drastically improves the overall performance of the switch due to the advantages it offers. The adoption of internal buffers makes the scheduling totally distributed, hence reducing the arbitration complexity to linear. It is this autonomy and the absence of coordination between the input (output) scheduling algorithms that makes CICQ switches appealing and desirable for multicast traffic support. Additionally, these internal buffers reduce (or avoid) the output contention. Meaning, they allow the inputs to send cells to an output irrespective of simultaneous cell transfer to the same output by other inputs. If an output is not ready to receive a cell from an input, the input can still send it to the internal buffer, provided that this internal buffer can accommodate that cell.

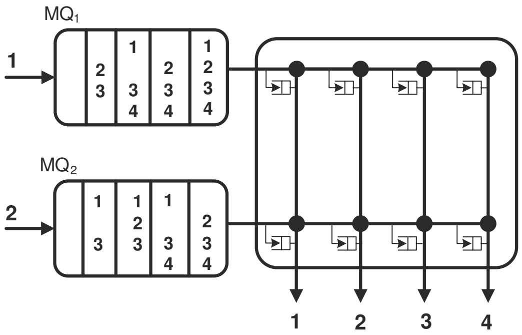


Figure 4.4: A 2×4 multicast CICQ Switch.

4.3.1 Switch Model

We consider our proposed switching architecture as depicted in Figure 4.3. The switch is assumed to operate on fixed-size packets (cells). There are N input cards, each one contains a FIFO multicast queue. The internal fabric consists of NM buffered crosspoints (XP). When an arriving cell, to an input port $i, \forall\, 1 \le i \le N$, has a fanout vector containing the output $j, \forall\, 1 \le j \le M$, it passes through the crosspoint $XP_{i,j}$, before continuing to the output buffer.

As with unicast scheduling, a multicast scheduling cycle consists of the following three steps: input scheduling, output scheduling and delivery notification. During the input scheduling phase, each input i, in an independent and parallel way, sends the HoL cell of its multicast FIFO queue to the internal buffer corresponding to its fanout set. Likewise, each output j selects, independently and in parallel, a non empty crosspoint buffer, $XP_{i,j}$, and sends its cell to the output queue. Then, the flow control is performed between the internal buffers and the input queues, to inform the inputs about the internal buffers status.

As depicted in Figure 4.3, the input scheduler is not embedded as proposed in Chapter 3. Embedding an input multicast scheduler within the buffered crossbar chip would require N flow control signals, from the input line card to the crossbar chip, to carry the fanout set of every new cell. Additionally, $N + \log k$ flow control signals are required, from the crossbar chip to the input line card, to carry the scheduler decision, where N denotes the set of reachable internal buffers that the cell can be sent to and log$k$ represents the index of the selected input multicast queue[2]. As a result, it is better to implement the input

---

[2] In the case of Figure 4.3, k equals one. However, k can be greater than one for other architectures, such as the one presented in Figure 4.5

scheduler at the input line card since only N flow control signals are required.

4.3.2 The Multicast Crosspoint Round Robin Algorithm: MXRR

The description of each scheduling phase of the Multicast cross-point Round Robin algorithm, MXRR, is as follows:

Input Scheduling:

- For each input, i, do
  - Send the FIFO HoL multicast cell to the set of internal buffers corresponding to its fanout vector.
  - If one or more internal buffers are not free, the cell remains at the HoL of that input and waits for the next input scheduling phase to send to its remaining internal buffers.

Output Scheduling:

- Initialization: All the output pointers are, arbitrarily, set to the same initial position and incremented, in each time slot, by one $mod\ (N)$.
- For each output, j, do
  - Starting from its pointer index, select the first non empty crosspoint buffer, XP, and send its queued cell to the output buffer.

The MXRR algorithm exhibits good properties such as fairness, non starvation, speed and simplicity in design. The output pointers setting is of key importance due to their synchronous update mechanism. To better see this, consider the 2×4 multicast CICQ switch depicted in Figure 4.4. Let us assume that the output pointers are all pointing to input 1 and all the internal buffers are empty. During the input scheduling phase, both HoL cells of MQ_1 and MQ_2 will be completely transferred to the internal buffers. During the output scheduling phase, since the output pointers have index 1 each, every output j will select the internal buffer $XP_{1,j}$, $\forall\ 1 \leq j \leq 4$. During the current time slot, the HoL cell of MQ_1 is completely served and its copies are all transferred to their destination output ports. At the beginning of the second time slot, $XP_{2,2}$, $XP_{2,3}$, and $XP_{2,4}$ are occupied. Therefore, the second cell of MQ_2 (which becomes the HoL cell) cannot send its copies to all their destination outputs

$\{1, 3, 4\}$. It can only send to $XP_{2,1}$ which leaves a residue of $\{3, 4\}$. MQ_1, however, can send its HoL cell completely to the internal buffers. During the output scheduling phase, since the output pointers indexes are incremented to 2, each output j will select the internal buffer $XP_{2,j}$, $\forall\ 1 \leq j \leq 4$. This means that, during this time slot, the HoL cell of MQ_2 is completely served and the second cell is also partially served. From this example we draw the following properties and advantages of the MXRR scheduling scheme:

- The MXRR scheme guarantees the total service of at least one packet each time due to the output pointers setting (which point to the same internal buffer and advance synchronously). Moreover, the time a packet waits at the HoL is bounded by number of input ports, N.
- Fair and starvation free. Since the output pointers move artificially and in a synchronous fashion irrespective of the chosen packet, the starvation problem will never occur. The chance of service for any two cells from two different input ports is exactly the same due to the round robin pointer movement.
- Simple in hardware implementation. Each input independently carries out FIFO arbitration. The outputs, on the other hand, work in a totally distributed and parallel manner. No computation and comparison of weights is needed to make an arbitration decision. Each output arbiter just performs simple static round-robin arbitration.
- The MXRR achieves higher throughput and a lower packet latency than all existing bufferless multicast FIFO algorithms. We will examine this property through performance evaluation in Section 4.5.

4.4 The Multicast K FIFOs CICQ Switch Architecture

Although the Multicast FIFO architecture is simple and practical, it suffers poor performance due to the HoL problem. In order to completely eliminate the HoL blocking problem, multicast cells having the same fanout sets must be placed in the same *separate* multicast queue (MQ), which requires as many MQs as the multicast VOQ (MC-VOQ) architecture would and this is clearly infeasible even for a small switching system. An attractive alternative is to use a small number, k, of MQs per input to accommodate the incoming multicast cells. This is a good compromise to achieve good performance while

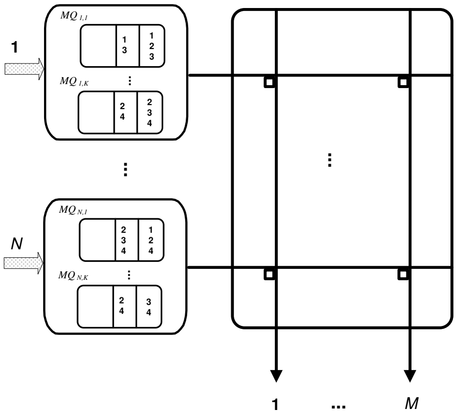


Figure 4.5: An $N \times M$ multicast k FIFO queues CICQ switch.

maintaining affordable hardware requirements. This is the model we adopt, as depicted in Figure 4.5. Each input maintains a small number, k, of multicast FIFO queues per input, where $\{k \mid 1 \leq k \ll 2^M - 1\}$. At each input, multicast queues are denoted by $MQ_{i,j}$ where $\{(i,j) \mid 1 \leq i \leq N; 1 \leq j \leq k\}$. An input multicast queue, MQ, is considered eligible (denoted EMQ) if it is not empty and at least one of its destination output ports corresponds to a free XP. Because k is much smaller than $2^M - 1$, cells with different fanout sets may have to be queued in the same input queue. This mapping is known as the multicast cell assignment policy, which we describe next.

4.4.1 Multicast Cell Assignment

Before going into detail, we first explain why multicast cells require a cell assignment scheme (policy) in order to place incoming multicast cells into specific multicast queues (MQ) while waiting their turn to be scheduled. In general, the cardinality of the fanout set, Φ, of a multicast cell can vary between $|\Phi_{\min}|$ and $|\Phi_{\max}|$. Since a multicast cell can have a minimum of 1 destination output port and a maximum of N destination output ports, we can set $|\Phi_{\min}| = 1$ and $|\Phi_{\max}| = N$. Therefore, the cardinality of the set of all fanout sets, C_Φ, can be calculated as follows:

$$C_\Phi = |\wp(\Phi)| = \sum_{\Phi_i=1}^{N} \binom{N}{\Phi_i} = 2^N - 1, \ \Phi_{\min} \leq \Phi_i \leq \Phi_{\max} \tag{4.1}$$

In order to completely avoid the HoL blocking problem, multicast cells having the same fanout sets must be placed in the same *separate* multicast queue (MQ), which requires as many MQs as C_Φ. Maintaining such a big number of MQs is however impractical. The alternative is to use a small number, k, of MQs per input to accommodate the incoming multicast cells. Because k is smaller than C_Φ, cells with different fanout sets have to be queued in the same MQ and every MQ receives cells with at most $\lceil \frac{C_\Phi}{k} \rceil$ different fanout sets. Thus, a *cell placement scheme* is required to map incoming multicast cells into the k multicast queues.

Since, in our model (Figure 4.5), the number of MQs, k, maintained at each input is much smaller than the number of all fanout configurations, C_Φ, a cell assignment policy is required in order to map incoming cells to the MQs. This has a significant effect on the scheduling performance. Previous work [87] has outlined some criteria in designing such a policy: *(i)* The heads of the MQs should be *diverse* in order to span a large number of the outputs. This would ensure more scheduling opportunities and work conservation. *(ii)* Cells with the same, or similar, fanout sets should be stored in the same queue, to reduce HoL blocking and prevent the out of sequence delivery problem. Many cell assignment schemes existed, such as Majority [87], Minimum Distance Queue (MDQ) and Load Balanced Queueing (LBQ) [90]. The MDQ scheme assigns a representative fanout set per MQ. Each incoming packet is inserted in the MQ with the representative having the minimum hamming distance from the packet's fanout set. The LBQ scheme assigns packets to MQ based on either sorting the packets fanout sets or fanout values. The previous two cell assignment algorithms are inspired by the Majority algorithm.

In the majority scheme, each MQ is associated a bit mask that corresponds to a subset of outputs. The mask is created by forming a balanced partition of the set of outputs whose cardinality is equal to the number of MQs. For example, if an 8×8 switch has 2 MQs per input, the partition will result in every MQ's mask equalling to $\frac{8}{2}$ = 4. The drawback of this partition is that the number of bits set for each mask decreases with increasing numbers of MQs. As a result, this assignment does not adequately capture the multicast destination sets of packets. The solution to the bit mask partition was to allow the bit masks to overlap. In addition to the original balanced partition, the remaining bits of every mask are set at random. Because masks overlap, Majority has to deal with cases when a packet has the maximum match with more than one mask. Majority resolves this by associating multiple sets of masks for each MQ and resolves ties with multiple levels of comparisons, with the final comparison breaking ties statically.

Unfortunately, in addition to its complex and time consuming procedure, Majority suffers a major problem by causing MQs to be unbalanced in the number of packets they cater for, causing part of these MQs to become under utilized. This problem arises from the statical tie breaking mechanism of Majority. Because ties are broken statically, packets with small numbers of fanout sets will always be inserted in the same MQ. This results in the same MQs receiving more packets than others and severely limits the performance of Majority, and defeats the purpose and advantage of increasing the number of MQs per input port (Figure 4.11 and Figure 4.12 illustrates this effect).

4.4.2 The Modulo Cell Assignment Algorithm

Our queueing structure implements a simple and efficient cell assignment scheme that does not require any sorting. Our cell assignment scheme, named *Modulo*, works as follows:

Modulo:

- For each input, i, do
- If an incoming multicast cell, c, has a fanout set Φ_c
    - Insert c in $MQ_{i,j}$, where $j = |\Phi_c|\ mod(k)$.

In addition to its simplicity, especially if the number of MQs, k, is a power of two, *Modulo* meets all previously mentioned criteria for efficient cell assignment. To better understand this, let us consider an example. Assume we have an 8×8 switch and 2 multicast queues per input ($k = 2$). At each input, i, *Modulo* places cells with even fanout sets in $MQ_{i,0}$ and those cells with odd fanout sets in $MQ_{i,1}$. This way, the heads of the MQs can span large numbers of destinations[3] (i.e.,the fanout set cardinality of the HoL cell of $MQ_{i,0} = 6$ and that of the HoL cell of $MQ_{i,1} = 3$). Moreover, cells with the same fanout sets are ensured to be queued in the same MQ, avoiding the out of sequence problem. Additionally, our scheme exhibits one further important property as a fair scheme, in the sense that it gives equal opportunities to the cells to advance to the head of the queues irrespective of their number of destinations. This is important as there are scheduling algorithms that use the fanout set as

---

[3] This is assuming uniform traffic. When the traffic is non-uniform or bursty, the HoL fanout sets may overlap to a large degree.

the weight for priority scheduling and, unless the cells fanout sets per MQ are diverse, MQ starvation (unfairness) can occur.

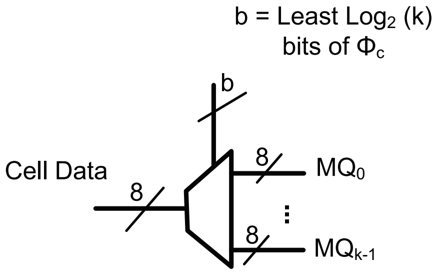


Figure 4.6: The Modulo cell placement scheme.

Table 4.1: The Modulo scheme implementation results for different k.

Number (k) of MQs	Area		Delay (ns)
	Slices	Equivalent Gate Count	
k = 2	4	248	1.5
k = 3	13	432	1.9
k = 4	22	576	1.9
k = 8	39	1032	2.4

Finally, the *Modulo* scheme has simple hardware requirement allowing it to place cells in the MQ at very high speed. Assuming there are k MQs per input and that each incoming cell has a fanout set value $|\Phi_c|$, our scheme can be implemented as DeMux with its input consisting of the cell data, its select bit(s) consist(s) of the least $\log(k)$ bits of $|\Phi_c|$ and its outputs consist of MQ_j where $\{j \mid 0 \leq j < k\}$ as depicted in Figure 4.6. We implemented the *Modulo* scheme in reconfigurable logic, using the Xilinx Virtex IV [76] as the target device and the Xilinx ISE platform 7.1 design flow platform. The design was simulated for different numbers of MQs (k = 2, 3, 4, and 8 respectively) and the post place and route results in terms of area and timing are depicted in Table 4.1. We can see from the table that the delay of the scheme is very small irrespective of the number of MQs used per input. When k = 2, the *Modulo* scheme checks the least significant bit of the cell's fanout value; if it is 0 the cell will be placed in MQ_0 else it will be placed in MQ_1. When k = 8, however, we can see that the delay is higher. It is of note that when setting k = 3 or 4 results in the same delay. This is because the select bits b are equal to

2 resulting in a 2-to-4 DeMux whether k is equal to 3 or 4. This results in the same delay and different area due to the partial use of the DeMux. As a result, it is better to choose a number of MQ, k, that is a power of 2 as it results not only in a full DeMux but also in balanced MQs in the number of fanout sets per queue. This is because, generally, the switch sizes are a power of 2 and using a small number k of MQs per input (that is also a power of 2) will result in balanced queues in the number of fanout sets that they can accommodate. Next, we need to devise the appropriate algorithm to schedule the transfer of cells from the input MQs to the output ports. The next section describes our proposed scheduling algorithm.

4.4.3 The Multicast k FIFOs Algorithm: MXRR_k

The MXRR_k algorithm is based on a static round robin selection. To avoid pointers synchronization, MXRR_k uses a fully unsynchronized pointer updating scheme similar to [54]. The description of each scheduling phase of the Multicast cross-point Round Robin algorithm, MXRR_k, is as follows:

Input Scheduling:
All input pointers are initialized to arbitrarily different positions.

- For each input, i, do
    - Starting from the pointer location, j, select the first eligible queue $EMQ_{i,j}$ and send its HoL cell copies[4] to the free internal buffers ($XP_{i,j}$).
    - Move the pointer to position $(j+1)\ (mod\ N)$.

Output Scheduling:
All the output pointers are, artificially, set to the same initial position and incremented, each time slot, by one $mod\ (N)$.

- For each output, j, do
    - Starting from the pointer position, select the first non empty cross point buffer and send its queued cell to the output buffer.

---

[4] Only copies destined to c outputs are sent, where $c \in \{1, ..., N\}$ and $XP_{i,c}$ is not full. Other copies will have to compete in later time slots.

The MXRR_k output scheduling remains the same as that of MXRR because every cell is treated the same irrespective of whether it is coming from an input FIFO or an input multicast queue, MQ. Additionally, maintaining the same output scheduling retains the same property of ensuring the *complete* service of at least one cell per time slot. MXRR_k differs from MXRR in its input scheduling. First, it uses a round robin priority pointer in servicing the input multicast queues, MQ. Second, the delay a cell waits at the HoL under MXRR_k is bound by kN time slots, where N is the number of input ports of the switch and k is the number of input multicast queues per input. The queueing delay inside the crossbar fabric is the same as that of MXRR (N time slots). So, the delay experienced by a HoL cell inside the switch under MXRR_k is bound by $N(k+1)$ time slots. The round robin mechanism of MXRR_k allows it to be fair and starvation free while kept simple in hardware.

4.5 Performance Results

This section analyzes the performance of different CICQ based queueing and switching architectures and their multicast scheduling algorithms. The performance results presented in this section are obtained for two different switch sizes of 8×8 and 16×16. The mean fanout size used throughout the simulation is equal to $\frac{N}{2}$, where N is the number of the output ports of the switch. Unless otherwise specified, the default internal buffer size is equal to 1 cell per crosspoint. We conducted the performance analysis under two input traffic scenarios: Bernoulli uniform and Bursty uniform. Please refer to Appendix B for finer details about the simulation environment and traffic scenarios.

The experimental results are structured in three parts. In the first part, we studied the performance of the MXRR algorithm for the multicast FIFO queues CICQ switching architecture. We compared the TATRA algorithm [88] and a Multicast SLIP-like (that we denote Mcast_SLIP) for the bufferless crossbar switch and the MXRR algorithm for the CICQ architecture. We chose TATRA because it is considered to be one of the most practical algorithms that achieve high performance and multicast SLIP because it resembles MXRR. The second part of the experiments targets the multicast k FIFO queues CICQ switching architecture. We started by assessing the performance of our devised Modulo cell assignment scheme. We compared Modulo with the Majority scheme. Then, we compared the performance of MXRR_k with that of the Mcast_SLIP algorithm. The latter part studies the tradeoff between the number of input multicast queues and the size of the internal crosspoint buffers.

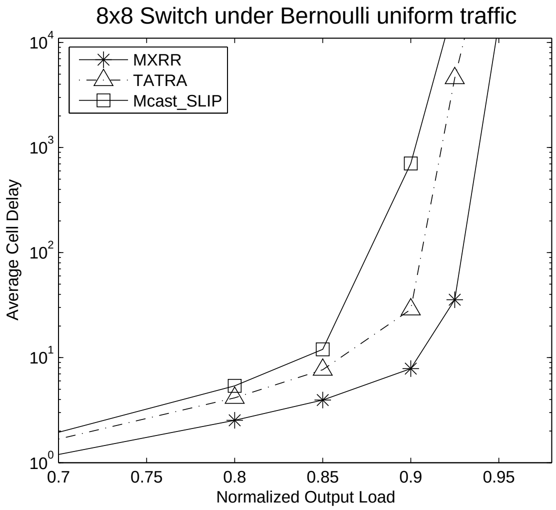


Figure 4.7: Average cell delay of 8×8 multicast FIFO switch under Bernoulli uniform traffic.

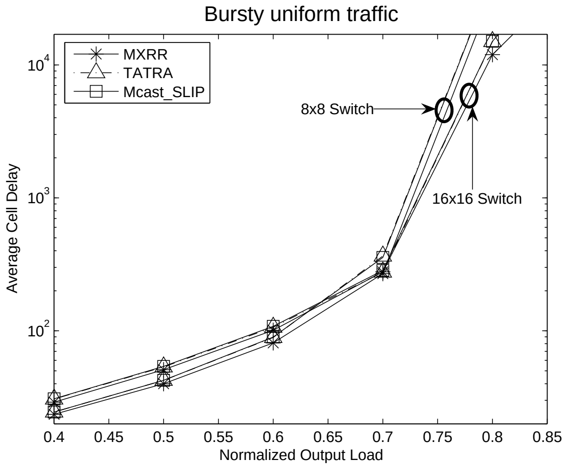


Figure 4.8: Average cell delay of a multicast FIFO switch under Bursty uniform traffic.

4.5.1 Performance of the Multicast FIFO Architecture

Figure 4.7 depicts the average delay performance of the TATRA and Mcast_SLIP for the an 8×8 bufferless crossbar switch compared to the MXRR

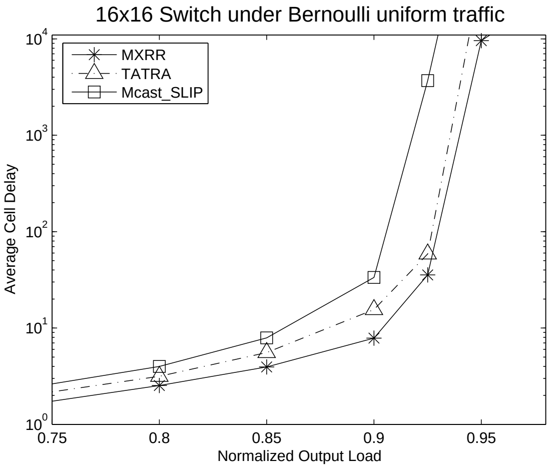


Figure 4.9: Average cell delay of 16×16 multicast FIFO switch under Bernoulli uniform traffic.

algorithm for an 8×8 buffered crossbar switch. As the figure shows, MXRR has better delay performance than the other two. This result remains the same uniform bursty arrival as well. This confirms the superior performance of the CICQ, even with simple scheduling, as compared to the IQ architecture which employs a sophisticated algorithm, such as TATRA. Figure 4.8 illustrates the average cell delay under the same settings as above but with uniform bursty arrivals, with a burst length of 16 cells.

In order to better analyze the behavior of each algorithm, we tested the algorithms under the same settings as above using a larger sized switch. This is important because, as the switch size increases, the fanout sets of the cells also increase making it harder for the algorithm to schedule the traffic due to increased contention. To assess this behavior, Figure 4.9 (and part of Figure 4.8) depicts the average cell delay of each of the three algorithms for a 16×16 switch. Again, the MXRR algorithm keeps the shortest cell delay amongst the three algorithms both under Bernoulli uniform and bursty uniform arrivals. Average cell delay is expected to improve if we increase the internal buffer size. This is confirmed in Figure 4.10, where internal buffers sizes of 2, 4 and 8 cells are used for MXRR. This suggests that bigger internal buffer sizes can help in absorbing the multicast cell fanout problem and therefore giving scheduling opportunities to more cells in shorter durations.

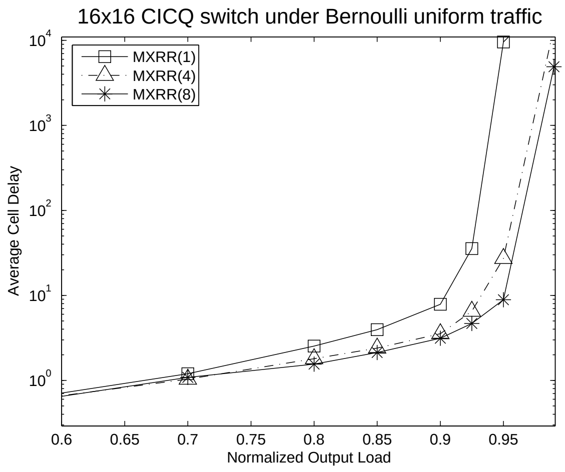


Figure 4.10: Average cell delay of MXRR with different internal buffer settings.

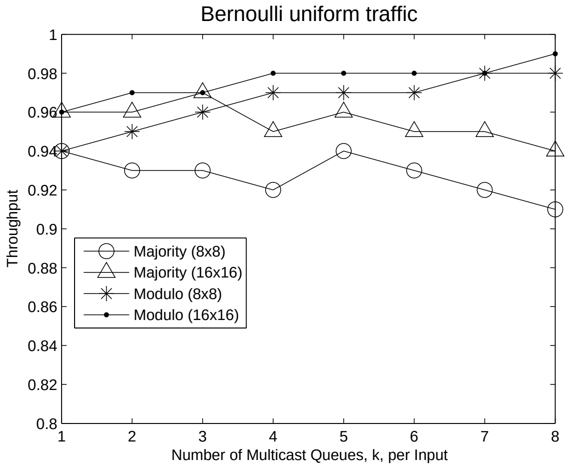


Figure 4.11: Throughput comparison between Modulo and Majority cell placement schemes under Bernoulli uniform traffic.

4.5.2 Performance of the Multicast k FIFOs Architecture

This section starts by assessing the performance of the Modulo cell assignment algorithm. We compared the Modulo and Majority cell placement schemes in terms of throughput and input queues occupancies for two switch sizes of

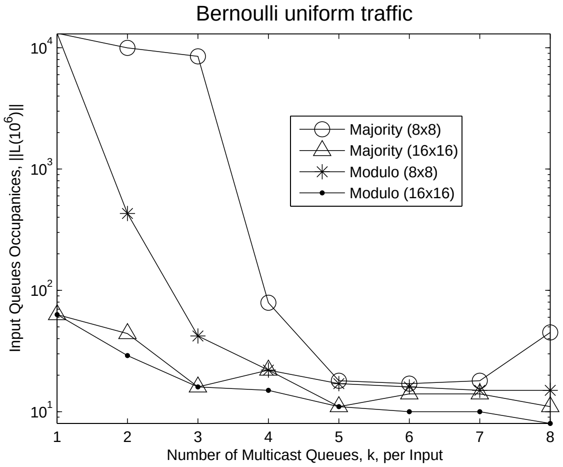


Figure 4.12: Input queues occupancies of Modulo and Majority under Bernoulli uniform traffic.

8×8 and 16×16 both employing the MXRR_k algorithm under uniform traffic. We varied the number of MQs per input and observed the maximum throughput achieved by each of the cell placement algorithms. We can see from Figure 4.11 that the throughput of Modulo increases proportionally with the number of MQs. However, Majority tends to have even lower throughput with increasing number of MQs. This is attributed to the unbalanced MQs effect by its assignment policy and its statical tie breaking. Figure 4.12 depicts the MQs occupancies for each algorithm just before saturation (at 95% input load). Again, Modulo outperforms Majority irrespective of the switch size or number of MQs. Since the input traffic is uniform, by applying Little's Law [94], we can directly deduct the average cell delay under each scheme from Figure 4.12. Since the input load is 95%, therefore the values of Figure 4.12 are similar to the average cell delay. This delay does not include the internal buffers delay, which is bound by the number of input ports N, as discussed in Section 4.4.3

In the previous experiments, we tested all three algorithms for the multicast FIFO queueing architecture. In the following simulations, we compare the delay performance of the Mcast_SLIP bufferless algorithm with the MXRR algorithm because of their similarities, as non weighted algorithms. Figure 4.13 depicts the average delay performance for each of Mcast_SLIP and MXRR for the multicast k FIFO queues architecture. We used 2 and 4 MQs per input for a 16×16 switch under Bernoulli unform traffic arrivals. We can see from Figure 4.13 that MXRR outperforms the bufferless Mcast_SLIP algorithm irre-

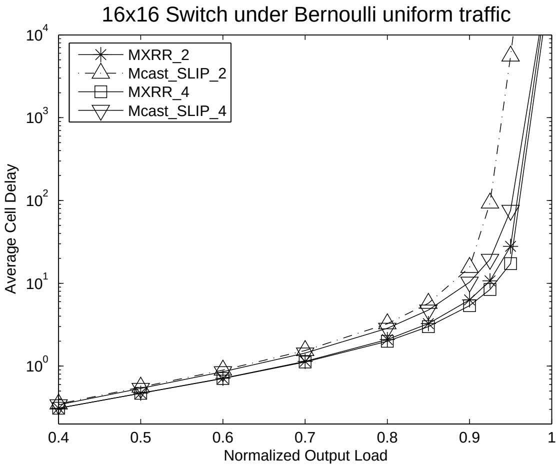


Figure 4.13: Average cell delay of 16×16 multicast k FIFO switch with different numbers of input queues, k = 2, 4.

spective of the number of MQs used per input. MXRR_2 still achieves higher performance while using half the number of MQs that Mcast_SLIP_4 does.

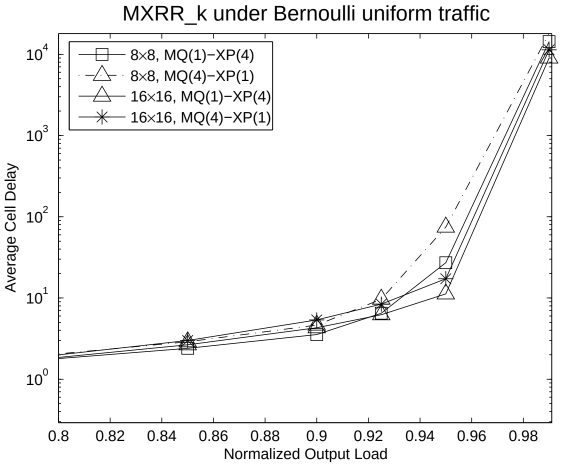


Figure 4.14: Average cell delay of MXRR_k with different MQ numbers and XP sizes.

In the remainder of the simulations, we study the effect of varying the size of the internal buffers and the number of multicast queues, k, per input port of the

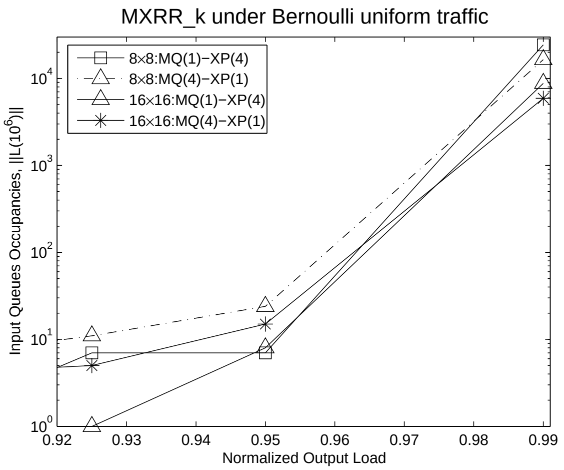


Figure 4.15: Input queues occupancies with different MQ numbers and XP sizes.

switch. Figure 4.14 depicts the average cell delay of MXRR_k using different numbers of MQs per input, different internal buffers sizes and different switch sizes. For example, 16×16-MQ(1)-XP(4) refers to the average cell delay of MXRR_k when using a 16×16 CICQ switch, 1 MQ per input port and internal buffer size, XP, of 4 cells per crosspoint. We can see from the figure that MXRR_k achieves a slightly shorter cell delay with just 1 MQ and an XP size of 4 cells than when using 4 MQs per input and just 1 cell per XP. In addition to confirming the importance of the internal buffers in improving the switching delay and reducing the HoL blocking, this result has a major implication on the design of the MXRR_k input scheduler. Instead of using MXRR_k that requires a priority encoder for the round robin selection among the k MQs per input port, a switch designer has the option to just use the MXRR algorithm (consequently the FIFO CICQ architecture) with k cells per XP.

Figure 4.15 depicts the occupancy of the MQs using the L^2 norm vector as defined in Appendix B.3. In the case of multicast CICQ architecture, the L^2 norm vector, at time slot n, is calculated as follows:

$$\|L(n)\| = \sqrt{\sum_{i=1}^{n} \sum_{j=1}^{k} \big(MQ_{i,j}(n)\big)^2}$$

We can see from the figure that, at 99% input load and with both switch sizes, the L^2 norm of MQ(1)-XP(4) is 1.5 bigger than that of MQ(4)-XP(1). However, the total number of cells per input port is smaller with MQ(1)-XP(4) than with MQ(4)-XP(1). To see this, let a be the L^2 norm of MQ(1)-XP(4) and b equals to the L^2 norm of MQ(4)-XP(1). Then, we have:

$$\sqrt{a^2} = 1.5\sqrt{4b^2} \text{ hence, } a = 3b$$

Meaning, at 99% input load, the total number of cells –using just 1 FIFO queue per input port and an XP size of 4 cells– is equal to 75% of the total number of cells when we use 4 FIFO queues per input port and XP size of 1 cell. This result is conform to the lower average cell delay (Figure 4.14) of MXRR with MQ(1)-XP(4) instead of MQ(4)-XP(1).

4.6 Summary

This chapter studies the multicast traffic scheduling problem in CICQ switches. We began by surveying existing multicast switching architectures with their scheduling algorithms and discussing their shortcomings. We proposed a CICQ switch architecture based on one multicast FIFO per input port. We devised a simple round robin algorithm, named MXRR, for this architecture and showed its better performance by comparison to existing algorithms.

We further studied the CICQ switching architecture, where there are multiple multicast queues per input port. We proposed a cell assignment algorithm capable of assigning incoming traffic to the input queues more efficiently than existing state of the art algorithms. We also extended the MXRR algorithm to schedule cells in the presence of multiple queues per input. Our devised algorithm, MXRR_k, outperforms alternative algorithms under various traffic scenarios. The experimental results showed an equivalence between using only one multicast FIFO queue per input or multiple queues per input, subject to a trade off in the size of the internal buffers. Hence, giving the option of choosing between the complexity and speed of the scheduling algorithm on one hand, and the cost of the buffered crossbar fabric on the other.

This chapter and the previous chapter each addresses either purely unicast traffic scheduling or purely multicast scheduling. However, real Internet traffic is a mixture of both traffic types. In the next chapter, we study the scheduling of integrated unicast and multicast traffic flows in one unified CICQ switch.

Chapter 5

Integrated Unicast and Multicast Scheduling

Internet traffic is a mixture of unicast and multicast flows. In the previous two chapters, we studied the scheduling problem in either a pure unicast context or a pure multicast context. In this chapter, we propose a novel switching architecture that combines both architectures studied in Chapter 3 and Chapter 4. Our proposed buffered crossbar (CICQ) switching architecture is capable of supporting both unicast and multicast traffic flows concurrently. We propose an integrated Round Robin based scheduler that efficiently services both unicast and multicast traffic simultaneously. Our algorithm, named Multicast and Unicast Round robin Scheduling (MURS), has been shown to outperform all existing schemes under various traffic patterns. We further propose a hardware implementation of our algorithm for a 16×16 CICQ switch. The implementation suggests that MURS can sustain a 20 Gbps line rate and a clock cycle time of 2.8 ns, reaching an aggregate switching bandwidth of 320 Gbps.

5.1 Introduction

The growing number of newly emerging applications such as teleconferencing, distance learning, IPTV etc. over the Internet has resulted in a growing proportion of multicast traffic. In addition to point-to-point (unicast) communications, a network node (high speed IP routers and ATM switches) is also required to deal with point-to-multipoint (multicast) communications and the combination of the two. In contrast to traditional switch design where unicast and multicast traffic flows are treated separately, designing a switching architecture and scheduling algorithms capable of supporting heterogenous, yet simultaneous, different traffic types is becoming increasingly important.

To date, little research has been done on the design of integrated scheduling algorithms to support both unicast and multicast traffic types. The previously proposed scheduling algorithms are in fact a combination of earlier unicast and multicast algorithms unified in one integrated scheduler. The input queueing structure has also combined a unicast queueing structure and multicast queueing structure. The widely used unicast queueing structure is the virtual output queueing (VOQ) [40], since it avoids the head-of-line (HoL) blocking problem [36]. As for multicast traffic, a multicast packet (cell) can have more than one destination, known as its *fanout set*. Consequently, a multicast queueing structure can vary from just one multicast FIFO queue per input to $2^N - 1$ queues per input, where N is the number of output ports of the switch. Depending on the input queueing structure, integrated scheduling algorithms have been proposed. These algorithms were mainly proposed for the input queued (IQ) bufferless crossbar fabric switching architecture because of its scalability, low hardware requirements and *intrinsic multicast capabilities*. Most of the proposed algorithms were based on input VOQs for unicast traffic and one FIFO queue for multicast traffic [18] [95]. Other algorithms [96] used VOQs for unicast and k queues for multicast traffic, where $1 < k \ll 2^N - 1$. The major drawback of these algorithms lies in their inability to either achieve high performance or run at high speed. This is mainly due to their centralized design and to the nature of the crossbar fabric switching architecture.

The previous chapters have shown the optimal performance of the CICQ switching architecture. Instead of one centralized and complex scheduler, a CICQ switch maintains one scheduler per input as well as one scheduler per output. These schedulers are therefore decoupled and can work independently in parallel, improving the switching performance. Substantial work has focussed on designing unicast algorithms for the CICQ switching architecture (see Chapter 2). However, fewer results have appeared for multicast scheduling in CICQ switches [97] [92] [93]. These algorithms, unicast and multicast, have been shown to have superior performance than all algorithms proposed for the IQ bufferless switching architecture.

Despite the CICQ switches potential in solving the scheduling complexity issues faced by their bufferless predecessors, the problem of scheduling integrated (unicast and multicast) traffic in CICQ switches has not been addressed so far. In this chapter, we fill this gap and propose the following:

- An integrated CICQ switching architecture that supports concurrent unicast and multicast flows. The proposed architecture, shown in Figure 5.1, is based on input VOQs for unicast traffic and k $(1 \leq k \ll 2^N - 1)$ FIFO queues per input for multicast traffic.
- A simple round robin scheduling algorithm, termed Multicast and Unicast Round robin Scheduling (MURS), capable of arbitrating both traffic types simultaneously. Different variations of the MURS algorithm are also proposed, depending on the scheduling priority of each traffic type.

The proposed MURS algorithm was shown, through simulation, to achieve high performance and outperform alternative algorithms under various traffic scenarios and combinations of unicast and multicast fractions. Similar to the previous chapter, simulation results showed that, even in the presence of mixed input traffic, we can still trade the size of the internal buffers for the number of input multicast queues resulting in better delay performance as well as simpler scheduler design. We further implemented the MURS algorithm for a 16×16 buffered crossbar switch. It has been segmented into 7 clock cycles with a clock cycle time of 2.8 ns. The hardware implementation results suggest that our proposed algorithm can sustain up to 20 Gbps line rate, allowing every switch line card to forward more than 47 million ATM cells per second.

The remainder of this chapter is organized as follows: Section 5.2 presents background knowledge and related work. In Section 5.3, we introduce the integrated CICQ switching architecture. We describe the proposed MURS algorithm, along with two variations: one for unicast priority scheduling and the other for multicast priority scheduling. Section 5.4 presents the performance study of our algorithms with a comparison to existing schemes. In Section 5.5, we propose a possible hardware implementation of our proposed scheme, for a 16×16 CICQ integrated switch. Finally, Section 5.6 provides a summary of the chapter.

5.2 Background

The problem of packet scheduling has been extensively studied over the past two decades for IQ bufferless crossbar based switches. Most of the research work has focused either in a purely unicast or a purely multicast context. Several unicast scheduling algorithms have been proposed (see Section 2.4.1).

The scheduling of multicast traffic flows in IQ switches has also been extensively studied, as discussed in Chapter 4. The multicast queueing structure is paramount to the performance and the implementation feasibility of the switching system, and different solutions have been studied. Similar to the work on IQ switches, CICQ switches have attracted great interest recently due to the advantages they offer in reducing the scheduling complexity and scaling the switching performance. Considerable research work has focussed on scheduling unicast traffic in CICQ switches, as described in Chapter 2 and Chapter 3. Little attention, however, has been dedicated to scheduling multicast traffic in CICQ switches. The first work on multicast traffic scheduling in CICQ switches date back to just 3 years ago [97]. Since then, only a few additional results have been proposed [92] [93].

Despite the substantial work advocated to either unicast or multicast scheduling, comparatively little has been done on integrating unicast and multicast traffic. Except [98], where the architecture is a shared memory, the few other algorithms have been proposed for the IQ buffer-less switching architecture. In [95], the problem of integration of unicast and multicast has been addressed and its hardness has been derived. At each input, the queueing architecture was based on VOQs for unicast and one multicast queue for multicast. A practical algorithm was proposed that consists of scheduling multicast traffic first and leaving the unicast traffic for idle inputs (or outputs). While this solution achieves good performance, it leads to permanent starvation of unicast flows. A similar architecture to [95] was proposed in [99]. In more recent work [96], the input queueing structure used was based on VOQs for unicast and a small number, k, of multicast queues for multicast per input. The authors proposed integrated algorithms based on previous unicast and multicast scheduling algorithms. The integration was based on some priority metrics, such as time and/or multicast service ratio. These algorithms perform many iterations in order to achieve good performance, limiting their scalability in port counts and/or speed per port.

Although the CICQ switching architecture exhibits higher performance than the IQ bufferless architecture, the integration of unicast and multicast traffic in CICQ switches has thus far not been addressed. Motivated by the above, in this chapter we describe a CICQ switch capable of supporting both unicast and multicast traffic flow simultaneously. The next section introduces our proposed architecture.

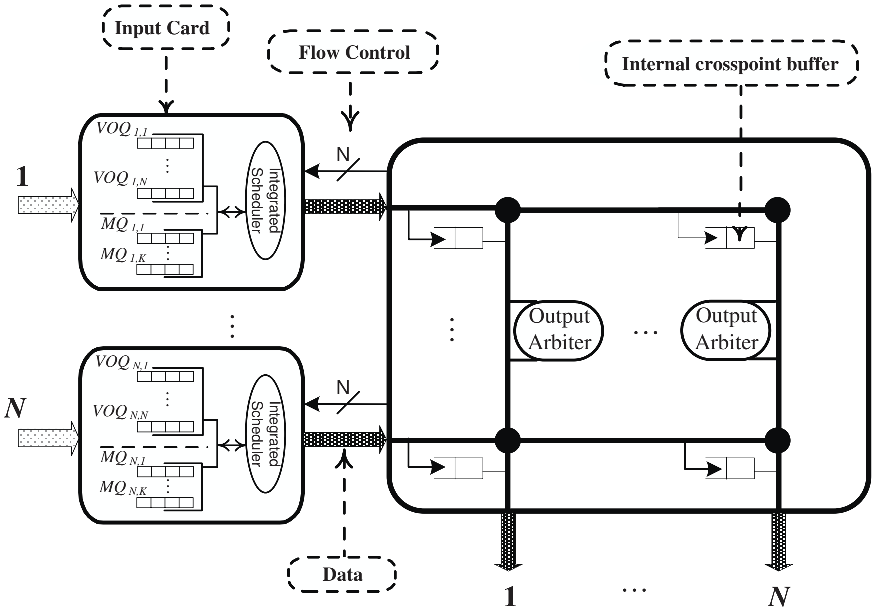


Figure 5.1: The integrated CICQ Switching architecture.

5.3 The Integrated CICQ Switching Architecture

The proposed CICQ switch model is depicted in Figure 5.1, for an $N \times N$ switch. This architecture differs from conventional CICQ switches [60] in its input queueing structure as well as its input scheduling. There are N input ports, each maintaining two types of queues: unicast traffic queues and multicast traffic queues. The VOQ structure is adopted for unicast queues and there are N VOQs per input, one per output. When a unicast cell, destined to output j, arrives at input i, it is placed in $VOQ_{i,j}$. The multicast k FIFO queueing architecture is used for multicast flows, as described in Section 4.4. At each input, multicast queues are denoted by $MQ_{i,j}$ where $\{(i,j) \mid 1 \leq i \leq N; 1 \leq j \leq k\}$. The Modulo cell assignment scheme (see Section 4.4.2) is used to place incoming multicast cells into their appropriate input queues.

In addition to the input queueing structure, each input card contains an *integrated input scheduler*. The scheduler, at each input, examines the HoL of the *eligible* queues belonging to that input and selects one cell to be transmitted to the buffered crossbar fabric chip. An input VOQ is deemed eligible if it is not empty and its corresponding XP is available. An input multicast queue (MQ) is considered eligible if it is not empty and at least one of its destination output ports corresponds to an available XP. We denote an eligible queue by EQ irrespective of whether it is a VOQ or a MQ. The buffered crossbar fabric chip, along with the output schedulers and the flow control mechanism, remains the same as that described in Section 4.3.1.

5.3.1 Integrated Scheduling

This section introduces the proposed integrated scheduling algorithms, Multicast and Unicast Round robin Scheduling (MURS). Because the input queueing structure consists of two types of queues and two types of traffic (unicast and multicast), extra focus has to be placed on the input scheduler at each input. The input scheduler, is not only required to *select* cells to be transmitted to the fabric chip, but also needs to decide *when* to choose a cell and *from which* set of queues (VOQs or MQ). This is called the *integration* phase of the scheduler. The integration phase of the scheduler determines the priority of scheduling of each traffic type. The selection policy of our scheme is based on round robin because of its fairness and simple hardware implementation. The selection policy is fixed and independent of the integration phase. Before discussing the integration policy, we first describe the selection policy since it is fixed. The specification of the selection policy is as follows:

Selection Phase:
Select_Queue (Queue_type , Pointer_type):
N = number of queues in Queue_type;
i = current input;

- Starting from the *Pointer_type* index, select the first eligible queue *EQ*$_{i,j}$ and send its HoL cell[1] to the internal buffer *XP*$_{i,j}$.
- Move *Pointer_type* to the location $(j+1)\ (mod\ N)$.

The integration phase of the scheduler decides the priority of the scheduling of cells at the queue type level. Each input scheduler maintains 2 priority pointers: a unicast pointer (UP) and a multicast pointer (MP). If the VOQs

---

[1] If the cell is multicast, then only copies destined to c outputs are sent, where {c|c $\in \{1, ..., N\}$ and *XP*$_{i,c}$ is available}. Other copies will have to compete in later time slots.

(MQ) set of queues is chosen to select a cell from, the round robin pointer will be based on UP (MP). Note that we consider *fanout splitting* when serving multicast flows [100]. The integration phase is responsible for deciding which set of queues to choose from. As the traffic can be unicast, multicast or a mix of both, we derived 3 integration policies. The first is called MURS_uf (unicast first) and always gives priority to unicast traffic. The second, MURS_mix, is designed to be a fair policy and treats both traffic types equally. MURS_mix gives priority to unicast traffic during even time slots, while multicast traffic is favored during odd time slots. The third integration policy is named MURS_mf (multicast first) and always gives priority to multicast traffic. Each integration policy corresponds to an input scheduling algorithm.

MURS_uf always gives priority to unicast flows over multicast flows. Therefore, in the presence of mixed unicast and multicast traffic, MUR_uf will always favor the VOQ set of queues to receive service and leave remaining idle connections to multicast flows. As a result, this scheme will produce more one-to-one connections than one-to-many connections. This causes performance degradation under heavy loads since when a unicast cell is chosen to be sent from an input port containing multicast cells, only one cell (*copy*) will be transmitted to the buffered crossbar fabric. The specification of the MURS_uf algorithm is as follows:

```
MURS_uf:
/*Always Unicast traffic first (prioritized)*/:
    • Select_Queue(VOQs , UP);
    • If no queue was selected
        – Select_Queue(MQs , MP);
```

If instead of MURS_uf we allow preference to multicast flows in the presence of unicast flows at the same input, this would result in more cells (*copies*) being transmitted to the buffered crossbar. As a result, the performance can be greatly scaled up. This is exactly what MURS_mf algorithm achieves, by favoring multicast flows over unicast flows. Despite the performance difference, there are similarities between MURS_uf and MURS_mf. They both have the same performance when the traffic is either purely unicast or purely multicast. Additionally, both schemes are *unfair* . Each tries to monopolize the switch bandwidth to its preferred traffic flows and this is undesirable. The specification of the MURS_mf algorithm is as follows:

```
MURS_mf:
/*Always Multicast traffic first (prioritized)*/:
  • Select_Queue(MQs , MP);
  • If no queue was selected
      – Select_Queue(VOQs , UP);
```

As a compromise between MURS_uf and MURS_mf, we propose MURS_mix. The scheduling priority of each traffic type is time slot dependent. The specification of the MURS_mix algorithm is as follows:

```
MURS_mix:
/*Equal Priority*/:
  • If current time slot is even      /*Unicast is served first*/
      – Select_Queue(VOQs , UP);
      – If no queue was selected
          ∗ Select_Queue(MQs , MP);
  • Else                              /*Multicast is served first*/
      – Select_Queue(MQs , MP);
      – If no queue was selected
          ∗ Select_Queue(VOQs , UP);
```

In addition to its fairness in the presence of different traffic types, the MURS_mix algorithm exhibits the same performance as the other two algorithms when the traffic is all unicast or all multicast. The properties of MURS_mix makes it a good candidate for being an optimal integrated scheme because: (i) It is fair and starvation free both on the traffic level as well as the flow level. In the presence of different traffic types, MURS_mix provides equal chances (even and odd time slots) to heterogeneous traffic types to be served. At the flow level, the round robin scheduling mechanism ensures fairness to flows belonging to different queues (whether unicast or multicast) and schedules them with the same likelihood. (ii) MURS_mix requires simple hardware allowing it to run at high speed. (iii) Finally, MURS_mix shows enhanced performance in terms of high throughput and low cell latency by comparison to existing algorithms. This is illustrated in the following section.

For all the 3 input schedulers, we used the same output scheduling algorithm that we described in Section 4.3.2. It is based on a static round robin selection, wherein all the output arbiters share the same pointer. The pointer is initialized to a random position and is incremented by 1 every scheduling cycle. The pointer setting is very important and has a two-fold advantage. First, the synchronous move of the output pointer ensures that at least one complete multicast cell is discharged every scheduling cycle as described in Section 4.3.2. Second, while our scheme adopts a fanout splitting discipline resulting in higher throughput, it also closely resembles a non-fanout splitting discipline which results in optimized use of internal bandwidth on the serial links between the input line cards and the buffered crossbar fabric core.

5.4 Performance Results

This section presents the simulation study of an 8×8 and a 16×16 CICQ switching systems employing the MURS set of algorithms. The experimental results are structured in 3 parts. In the first part of the experiments, we compare the performance of MURS_mix to the Eslip algorithm which uses a bufferless crossbar switch [18]. The second set of experiments studies the performance of our set of algorithms under different settings of MQs. The last section of the experiments analyzes the effect of varying both the number of MQs and the size of the internal buffers. Additionally, we observe the stability of the input queues under different traffic, input queueing and internal buffer size settings.

We studied the performance of our set of algorithms under the Bernoulli uniform and bursty uniform traffic scenarios described in Appendix B.2. Arriving cells can be either unicast or multicast. Cells arrive with a rate denoted by λ. Since the traffic is uniform, λ is the input load of the switch. The departure rate is denoted by μ. Similarly, μ is the output load of the switch. We consider admissible traffic, no input or output is oversubscribed. Because the traffic is a combination of unicast and multicast flows, the input load consists of a multicast fraction (f_m) and a unicast fraction (f_u), where $\{(f_m, f_u)|f_m = 1 - f_u\}$. The fanout set, Φ, of multicast cells has cardinality (fanout number) $|\Phi|$ which is uniformly distributed between 1 and 16 and all outputs have equal chances to be the destination of a multicast cell. Based on the above, the relationship between the switch input and output loads is expressed by Equation (5.1).

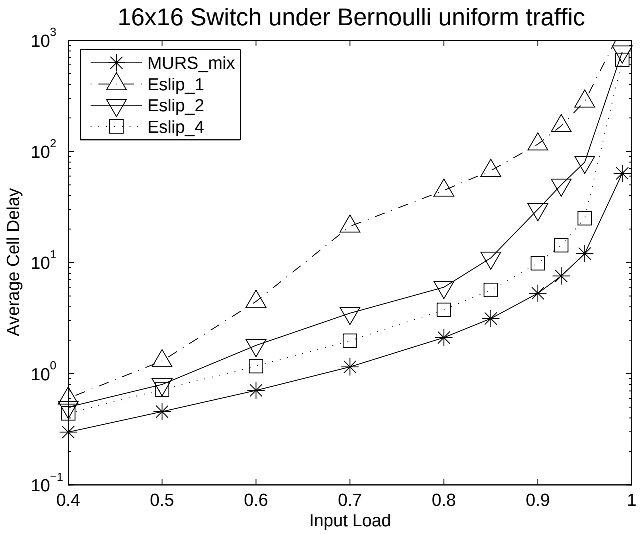


Figure 5.2: Average cell delay of MURS_mix and Eslip under Bernoulli uniform unicast traffic ($f_m = 0$).

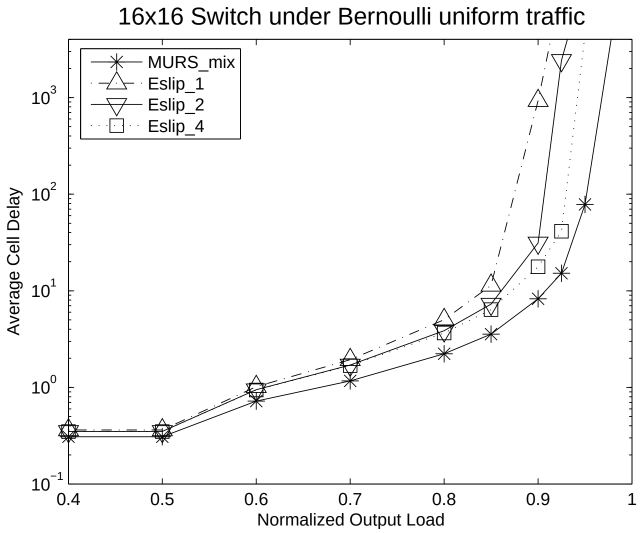


Figure 5.3: Average cell delay of MURS_mix and Eslip under Bernoulli uniform mixed traffic, ($f_m = 0.5$).

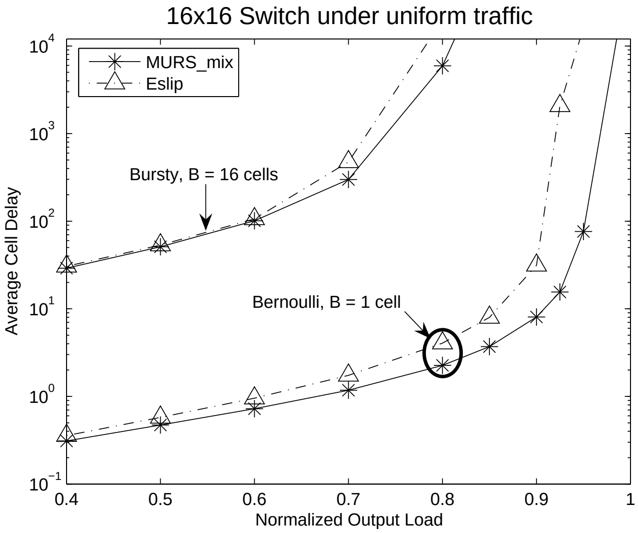


Figure 5.4: Average cell delay of MURS_mix and Eslip under Bernoulli uniform multicast traffic ($f_m = 1$).

$$\mu = \lambda(f_u + |\Phi| f_m). \tag{5.1}$$

In our simulation, we averaged the cells fanout set to be $|\Phi| = 8$. Following our settings and substituting f_u with f_m, we have:

$$\mu = \lambda(1 + 7f_m). \tag{5.2}$$

For example, if we set f_m to be 0 in Equation (5.2), the traffic is all unicast. When we set it to 1, the traffic becomes pure multicast. Whereas, if we fix μ to 1 for example (switch fully loaded), we can vary f_m and see its effect on the throughput. When $f_m = 0.5$, the incoming traffic is evenly distributed between unicast and multicast flows.

5.4.1 MURS_mix vs. Eslip

Because Eslip is based only on a single multicast queue per input, we used the same settings with MURS_mix by using just one MQ ($k = 1$) for fair comparison. Note that Eslip_i refers to Eslip with i iteration(s). In Figure 5.2, we compare the average delay performance of MURS_mix and Eslip under Bernoulli uniform traffic with all cells being unicast. As depicted in Figure 5.2, the performance of MURS_mix is always higher than Eslip irrespective of the

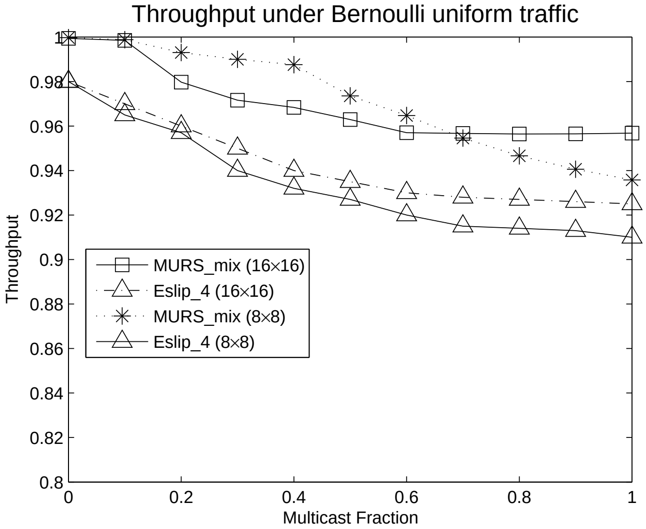


Figure 5.5: Throughput performance of MURS_mix and Eslip_4 under different switch sizes and different multicast fractions.

number of iterations performed by the latter. Figure 5.3 depicts the average cell delay of the two algorithms when the input traffic is evenly distributed between unicast and multicast flows ($f_m = 0.5$). Again, MURS_mix has a shorter average delay than Eslip. Figure 5.4 depicts the delay performance of MURS_mix to Eslip when the incoming traffic is all multicast. As we can see from the previous 3 figures MURS_mix always achieves a far shorter delay than Eslip irrespective of the incoming traffic type.

We wanted to compare the performance of MURS and Eslip for different mixed traffic settings. However, checking all possibilities of mixed traffic requires tuning f_m from 0 to 1 and observing the throughput. To this end, we fixed the output load, μ, to be 100% (fully loaded system) and recorded the throughput of each algorithm as f_m varies from 0 to 1. Figure 5.5 compares the maximum achievable throughput of MURS_mix and Eslip_4 under different switch sizes[1] (8×8 and 16×16). MURS_mix achieves higher throughout than Eslip irrespective of the switch size and/the multicast fraction of the incoming traffic. Note that, while each of our algorithms have a smaller delay than Eslip, we chose MURS_mix because it is more analogous to Eslip in the sense that is does not prioritize one traffic type over another.

---

[1] Note that when the size of the switch is 8×8, the average fanout size becomes 4 and Equation (5.2) becomes: $\mu = \lambda(1 + 3f_m)$.

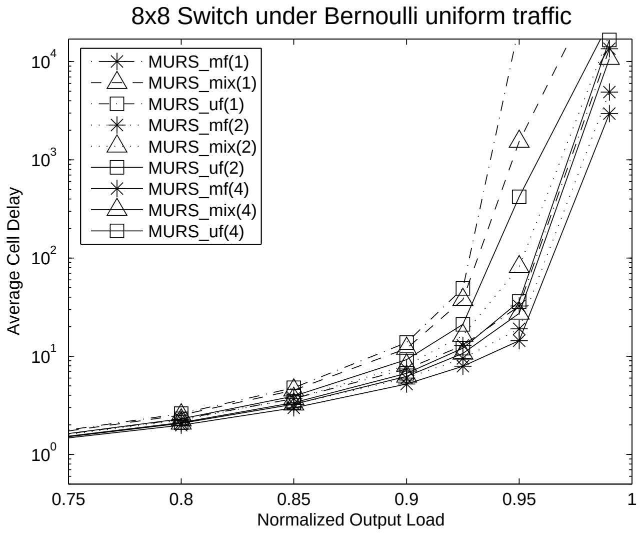


Figure 5.6: Average cell delay of an 8×8 CICQ switch running MURS with different numbers of MQs per input and mixed input traffic ($f_m = 0.5$).

5.4.2 The Effect of MQs Number, k

The remainder of the simulation is conducted for multicast queues set MQ per input equal to or bigger than one ($k \geq 1$). We study the average cell delay performance of each of our algorithms for k = 1, 2 and 4 respectively and evenly distributed traffic over unicast and multicast ($f_m = 0.5$). Figure 5.6 depicts the average delay for an 8×8 switch and Figure 5.7 shows the average cell delay for a 16×16 switch. As expected, the MURS_mf scheme has the best delay irrespective of the arrival traffic and the switch size. This is because it gives priority to multicast flows over unicast flows resulting in more connections per scheduling cycle. This result holds independently of the number of MQ used per input. MURS_uf, however, has the worst delay because it prioritizes unicast over multicast, resulting in fewer cells transferred to each output per scheduling cycle. MURS_mix has a moderate average delay because it treats both traffic types with the same priority. Overall, MURS_mix is the best choice due to its fairness. Figure 5.8 depicts the average cell delay of MURS_mix with varying switch sizes and different numbers of MQs. We can see that the average cell delay decreases with increasing numbers of MQs and this is due to the role of the MQs in reducing the effect of the HoL blocking problem.

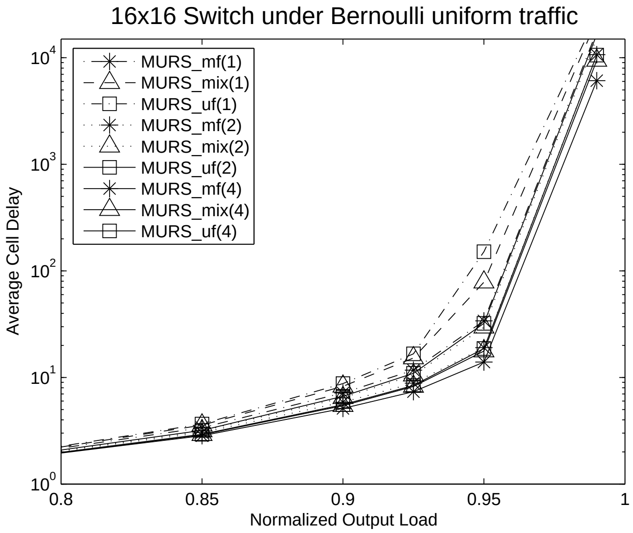


Figure 5.7: Average cell delay of a 16×16 CICQ switch running MURS with different numbers of MQs, k, per input and mixed input traffic ($f_m = 0.5$).

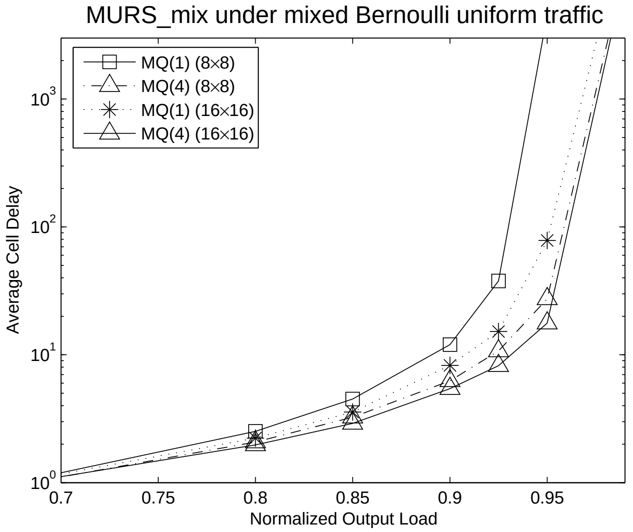


Figure 5.8: Average delay of MURS_mix with different switch sizes and different MQ numbers.

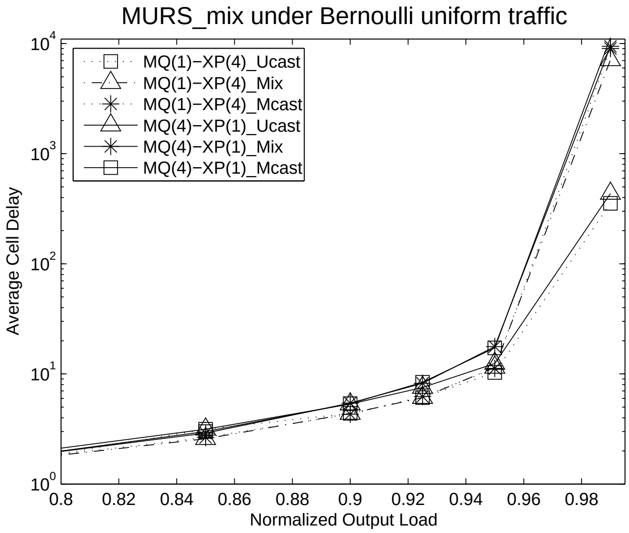


Figure 5.9: Average cell delay of MURS_mix as a function of the numbers of MQs, the XP sizes and input traffic combinations.

5.4.3 The Number of MQs vs. The XP Size

Due to the importance of the internal buffers in simplifying the scheduling, we tested our algorithms under different internal buffer sizes. Figure 5.9 depicts the average delay performance of the MURS_mix algorithm under 3 different scenarios. Incoming traffic is either all unicast ($f_m = 0$), or a mix ($f_m = 0.5$) or all multicast ($f_m = 1$). We varied the number of input multicast queues per input as well as the size of the internal buffers (XP) and studied their effect under each traffic scenario. For example, "MQ(1)-XP(4)_Ucast" corresponds to the MURS_mix algorithm with 1 multicast queue per input (MQ=1), 4 cells per internal buffer (XP=4) and incoming traffic consisting of unicast cells only. "MQ(4)-XP(1)_Mix" corresponds to MURS_mix with 4 MQs ($k = 4$) per input, 1 cell per XP and a mixed incoming traffic over unicast and multicast flows ($f_m = 0.5$).

The plots in Figure 5.9 show that the average delay of MURS_mix is shorter when only one multicast is used per input port (instead of 4) with an internal buffer size of 4 cells per XP (instead of 1). It is apparent that the delay of "MQ(1)-XP(4)_Ucast" should be shorter than that of "MQ(4)-XP(1)_Ucast". The reason for this is because incoming traffic is all unicast and therefore varying the number of MQs does not affect the delay. However, increasing the size

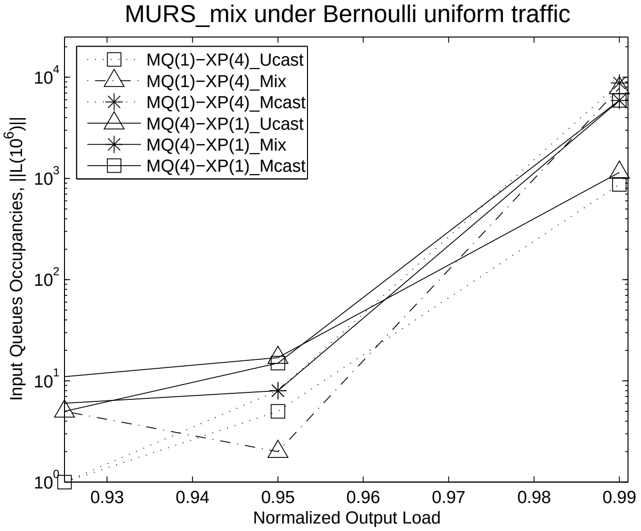


Figure 5.10: Input queues occupancies of MURS_mix as a function of the numbers of MQs, the XP sizes and input traffic combinations.

of XP does. When the incoming traffic is mixed ("MQ(1)-XP(4)_mix") and at 99% input load, the average cell delay of MURS_mix is 25% shorter than the cell delay when using "MQ(4)-XP(1)_mix".

We also studied the stability of the input queues under the same settings as above. We used the L^2 norm vector representing the occupancy of all input queues as defined in Appendix B.3. Because each input port contains both VOQs and MQs, the L^2 norm vector in this case is defined as follows:

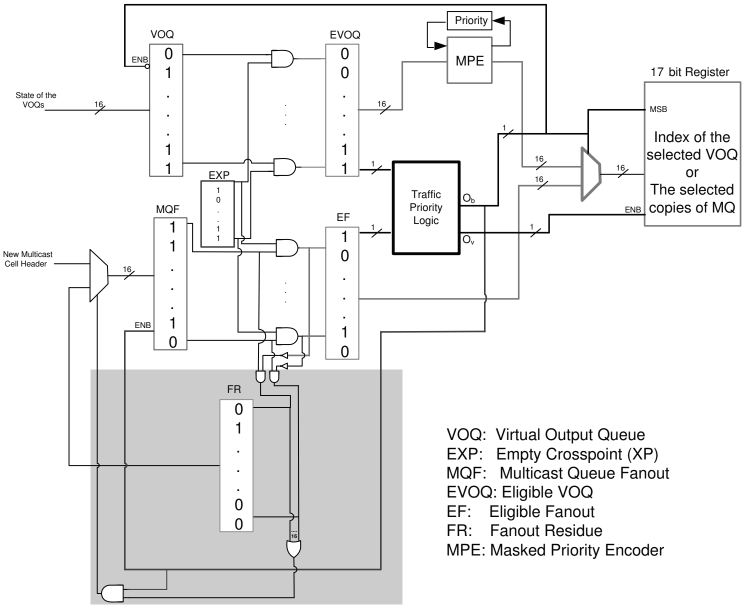
$$\|L(n)\| = \sqrt{\sum_{i=1}^{n}\Big(\sum_{j=1}^{n} VOQ_{i,j}(n)^2 + \sum_{l=1}^{k} MQ_{i,l}(n)^2\Big)}$$

As depicted in Figure 5.10, the input queues occupancy is smaller when we use only 1 multicast queue per input and an internal buffer size of 4 cells compared with employing 4 multicast queues per input and internal buffer size of 1 cell.

These results endorse our argument in Chapter 4 regarding the simplification in the design of the input scheduler. Using just 1 MQ per input port instead of 4 MQs at the expense of a little increase in the size of the internal buffers results in significant reduction in the hardware complexity of the input integrated scheduler. This is because the scheduler needs to maintain the state of the fanout sets of the HoL cells of every MQ and using just 1 MQ would translate in reduced information exchange and consequently in a shorter scheduling cycle time.

5.5 Hardware Implementation

This section presents the hardware implementation of the MURS scheduling algorithm for a 16×16 CICQ switch. Figure 5.11 depicts the schematic diagram of the algorithm and the scheduling process. The design can be divided into four main blocks, as follows:



VOQ: Virtual Output Queue
EXP: Empty Crosspoint (XP)
MQF: Multicast Queue Fanout
EVOQ: Eligible VOQ
EF: Eligible Fanout
FR: Fanout Residue
MPE: Masked Priority Encoder

Figure 5.11: The MURS_mix input scheduler algorithm.

- **Unicast Block:** This block is responsible for handling unicast traffic and consists of a 16 bit vector called VOQ (Virtual Output Queue) that contains the state of each VOQ in a line card. A 16 bit vector named the EVOQ (Eligible VOQ) is used for the index of the VOQs eligible for scheduling. The EVOQ is obtained by ANDing the VOQ vector with the EXP (Empty internal Crosspoint, XP). A component named MPE (Masked Priority Encoder) that is responsible for selecting the next index from the EVOQ vector in a round robin fashion. In our design, we used the MPE proposed by [101].

- **Multicast Block:** This block is responsible for the multicast traffic and contains the following components. A 16 bit vector, MQF (Multicast Queue Fanout) that contains the fanout set of the HoL cell of the MQ. A 16 bit vector, EF (Eligible Fanout) that contains the subset of output that the multicast cell can be sent to. The EF is the result of a logic AND of the MQF and the EXP vectors. A vector called FR (Fanout Residue) that contains the subset of unreachable output ports of the HoL cell of the MQ. This vector is obtained by ANDing the MQF and the logic inverse of the EF vector.

- **Traffic Priority Block:** This block manages the scheduling priority of unicast and multicast cells over time. It is designed as a state machine and works as follows: it takes as input two bits, the first bit (U_b) is the logic OR of the EVOQ vector bits and the second bit (M_b) is the logic OR of the EF vector bits. There is an internal bit (P_b) that determines which traffic is prioritized during the current scheduling cycle[2]. The value of P_b is inverted every scheduling cycle[3]. There are two output bits of the traffic priority block denoted O_b and O_v (see Figure 5.11). The value of each of them is obtained as follows:
  $O_b = (P_b \wedge M_b) \vee \overline{U_b}$
  $O_v = M_b \vee U_b$.

  The upper output bit of the traffic priority block is used as the select bit of the 2-to-1 Mux that decides which traffic type cell is chosen. The other output bit, O_v, indicates whether or not the output O_b is valid. The O_b bit along with the output of the the MPE (first block) and the content of the EF (second block) are used as the select bit and the 2 inputs of the Mux. Finally, the output of the Mux is forwarded, along with the select bit to a 17-bit register that contains the scheduling decision. This decision register is 17 bits wide with the select bit being its most significant bit (MSB). If the MSB bit is 1, then we know that the content of the register (16 bits) represents the reachable destination ports of the HoL multicast cell. Otherwise, the content of the register represents the index of the VOQ containing the selected unicast cell.

---

[2] MURS has been segmented into 7 clock cycles, which equals a scheduling cycle.

[3] Inverting P_b every scheduling cycle results in MURS_mix. Setting P_b to always 1 results in MURS_mf being implemented and when it is set to always 0, it results in MURS_uf.

Table 5.1: Hardware implementation results.

Module	Area (slices)	Delay (ns)
Input Scheduler (MURS_mix)	232	19.6
Output Scheduler (OS)	107	10.2

It is important to note that as soon as the output of the traffic priority block is computed, the following update process takes place. If O_b = 1, then we know that a multicast cell will be scheduled and the content of the VOQs will not need to change (VOQ ENB set to 0). Therefore, the content of the MQF must be updated (ENB = 1) with a new value and this depends on the content of the FR vector (the lower shaded area of Figure 5.11). The bits of the FR vector are ORed and if the result is 1 then the input Mux (see lower left side of Figure 5.11) will forward the content of the FR vector to the MQF vector as its new content. Otherwise, the result is 0 meaning the whole multicast cell was completely scheduled and therefore a new multicast cell fanout will be forwarded to the MQF vector. If, however, O_b = 0, meaning a unicast cell is chosen, the VOQ vector will be updated while the MQF remains unchanged.

As for the output arbiter, as mentioned previously, it consists of a round robin scheduling mechanism based on a priority encoder. In our design we employed the MPE design proposed by [101]. It has been segmented into 3 cycles. We employed the Xilinx Virtex IV platform and implemented our algorithm. The target device of our design was the Xilinx Virtex IV FX family and the results are obtained after place and route. Table 5.1 depicts the area, in number of slices, and delay, in nanoseconds, results of our design. The input arbiter has a clock cycle time of 2.8 ns and was segmented into 7 cycles resulting in a delay of 19.6 ns. The critical path of the design is the MPE block. The output arbiter has been segmented into 3 cycles of 3.4 ns each. The area results is 232 slices for the input arbiter and 107 for the output arbiter respectively.

5.6 Summary

Combined Input and Crossbar Queued (CICQ) switches have been known to outperform IQ switches due to the simplicity of their scheduling. The problem of integrating unicast and multicast traffic scheduling has, so far, mainly been studied for IQ switches. In this chapter, we proposed a novel CICQ switching architecture able to efficiently support both traffic types. We presented a simple set of integrated scheduling algorithms, named MURS, that can schedule concurrent unicast and multicast traffic flows. We studied the performance of our algorithms under a wide range of input traffic settings, different input queueing structures and different CICQ internal buffer sizes. In particular, the MURS_mix algorithm has been shown to exhibit very good performance and outperform previous algorithms. Simulation results suggested that a profitable trade off between the number of input multicast queues and the size of the internal buffers is possible, allowing for simplified design of the input scheduler. We presented a hardware implementation of the MURS algorithm for a 16×16 buffered crossbar switch using the Xilinx reconfigurable logic platform. The implementation results showed that MURS_mix can sustain a 20 Gbps line rate, reaching an aggregate switching bandwidth of 320 Gbps for our target switching system.

Chapter 6

Partially Buffered Crossbar Switches

The crossbar fabric is widely used as the interconnect for high performance packet switches due to its low cost and scalability. There are two main variants of the crossbar fabric: unbuffered and internally buffered. On one hand, unbuffered crossbar fabric switches exhibit the advantage of using no internal buffers. However, they require a complex scheduler to solve input and output ports contention. Internally buffered crossbar fabric switches, on the other hand, overcome the scheduling complexity using distributed schedulers. However, they require expensive internal buffers — one per crosspoint. In this chapter we propose a novel architecture, namely the Partially Buffered Crossbar (PBC) switching architecture, where a small number of *separate* internal buffers are maintained per output. Our goal is to design a PBC switch having the performance of buffered crossbars with a cost comparable to unbuffered crossbars. We propose a class of round robin scheduling algorithms for the PBC architecture. Simulation results show that using as few as 8 internal buffers per fabric output and irrespective of the number, N, of input ports of the switch, we can achieve even better performance than buffered crossbars that use N internal buffers per output.

6.1 Introduction

Various proposals for identifying suitable architecture for high-performance packet switches have been investigated and implemented in both academia and industry [18] [102] [103] [63]. These architectures can be classified based on numerous factors such as queueing schemes, scheduling algorithms, and/or switch fabric topology. The crossbar-based architecture is the dominant architecture for today's high-performance packet switches because of its low cost and scalability. As a result, the vast majority of commercially used core switches/routers are based on crossbar fabric with virtual output queueing (VOQ) [40]. The crossbar fabric architecture can mainly be classified into two categories: unbuffered or internally buffered crossbar fabric.

Extensive research work has been dedicated to unbuffered crossbar switches for over two decades (see Section 2.3 and Section 2.4). Figure 6.1 (a) depicts an Input Queued (IQ) crossbar fabric switch with VOQs at the inputs. The crossbar of an IQ switch runs at the same speed as external input/output ports. In order to maintain this low bandwidth requirement, an unbuffered IQ switch requires a centralized scheduler to resolve two main blocking problems, namely input and output contention. Input contention results from the constraint that an input can send at most one packet every time slot. Similarly, output contention arises from the constraint that an output can receive at most one packet every time slot. These blockings make the task of the scheduler complex and packets delay unpredictable. As a result, the switch performance essentially depends on its scheduling algorithm. Different classes of scheduling algorithms have been proposed [53] [51] [54] [104] [45]. Unfortunately, for high-bandwidth IQ switches, almost all scheduling algorithms are either too complex (see Section 2.4.2) to run at high speed or fail to exhibit satisfactory performance (see Section 2.4.3). This is mainly attributed to the centralized design of these schedulers and to the nature of the unbuffered crossbar switching architecture.

In order to overcome the scheduling complexity faced by IQ unbuffered crossbar switches, buffered crossbar switches have been proposed (see Section 2.5). Figure 6.1 (b) depicts a buffered crossbar switch, a crossbar where a limited amount of memory is added per crosspoint. The existence of internal buffers relaxes the output contention constraint, making the scheduling task much simpler. Buffered crossbars use distributed and independent schedulers (one per input/output port) to switch packets from the input to the output ports of the switch. A scheduling cycle consists of input scheduling, output scheduling and flow control to prevent internal buffer overflow. Efficient scheduling algorithms have been proposed for this architecture [67] [79] [105]. The scheduling simplification comes at the expense of a costly crossbar. The crossbar has to maintain N^2 internal buffers, where N is the number of input/output ports of the switch. The number of internal buffers grows quadratically with respect to

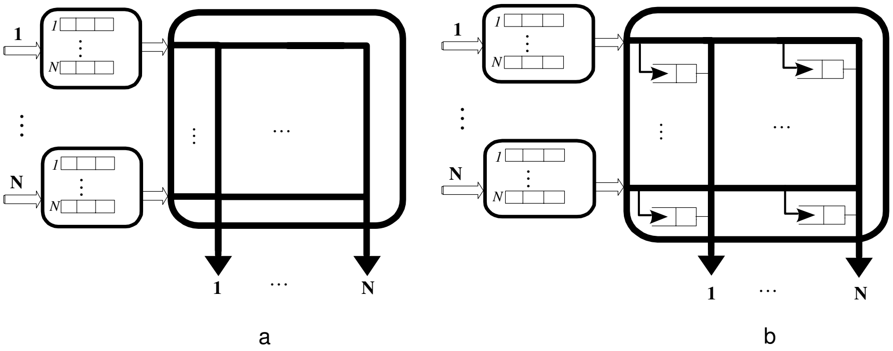


Figure 6.1: Crossbar Fabric variants: (a) Unbuffered Crossbar Fabric. (b) Buffered Crossbar Fabric, with N^2 Internal Buffers.

the switch size and linearly with round trip delays [63]. This makes buffered crossbar switches highly expensive and hence less appealing.

In this chapter, we propose a novel architecture, referred to as Partially Buffered Crossbar (PBC) switching. The PBC is designed to be a switching architecture that exhibits the performance of buffered crossbars but at a cost comparable to unbuffered crossbars. The PBC switch, depicted in Figure 6.2, contains a small number of separate internal buffers, $B \ll N$, per output port. We propose a class of pipelined scheduling algorithms for the PBC switch and study their performance under various traffic patterns. The experimental study shows that setting the number of internal buffers per output to $B = 8$ is sufficient for the PBC to achieve optimal performance irrespective of the switch size, N. Previous work proposed similar switching architecture to the PBC switch [106] [27]. Our work differs from [106] [27] both at the architectural and the scheduling level. While the architecture in [106] relies on internal shared memory per output port, our PBC architecture uses separate internal buffers per output, hence avoiding the requirement of expensive shared buffers. Secondly, the architecture proposed by [27] was targeting multistage switches whereas our proposed architecture targets single stage switches. On the scheduling level, our proposed algorithms outperform those proposed by [106] [27].

The remainder of the chapter is organized as follows: Section 6.2 presents the PBC architecture and its scheduling. In Section 6.3, we introduce our set of scheduling algorithms. The first algorithm is called Distributed Round Robin (DRR). It is based on round robin input and credit schedulers per fabric input and output respectively. Because of the credit release delay experienced by

DRR (see Section 6.3.2), we propose an alternative algorithm named DROP that drops unaccepted grants every time slot and consequently minimizes the credit release delay. We also propose an enhanced version of the DROP algorithm, named DROP-PR, that selects grants based on output priority. Section 6.4 presents the performance study of our algorithms under various settings. Finally, Section 6.5 summarizes the chapter.

6.2 The Partially Buffered Crossbar Architecture (PBC)

This section introduces the Partially Buffered Crossbar switching architectural organization along with its scheduling.

6.2.1 Switch Model

We consider the Partially Buffered Crossbar switching architecture (PBC) depicted in Figure 6.2. The switch operates on fixed sized packets (cells). Variable size packets are segmented into fixed sized cells while inside the switch and reassembled back into packets upon their exit. The PBC has N input and N output ports. When a cell, destined to output j arrives at input i, it gets queued in $VOQ_{i,j}$ while waiting its turn to be selected by the input scheduler (IS_i). There are N input schedulers, one per input port that control the transfer of cells from the input line cards to the internal fabric buffers. The input scheduler decision is coordinated with a grant scheduler that manages the internal buffers availability (and access) for each output. The input and grant schedulers communicate throughout a grant queue (GQ) maintained per input. There are N GQs, one per input, and each contains N entries, one per output. When a grant scheduler (GS), j, sends a grant, g, to input, i, $GQ_{i,j}$ is set to one. Once input, i, accepts g, $GQ_{i,j}$ is reset to zero.

The crossbar fabric contains a small number of internal buffers. These internal buffers are maintained per fabric port and there are $B \ll N$ separate internal buffers per fabric output. The fabric has B internal buses per output, one per internal buffer. These buses run at the same bandwidth as the external line rates. These buses are required in order to maintain low bandwidth. If we use one bus per output, instead, its bandwidth is required to be B times the external bandwidth in addition to intermediate buffering of cells which is costly. Each fabric output contains an output scheduler (OS) that arbitrates cells departures from the internal buffers to the output queue. There are N credit queues (CQ),

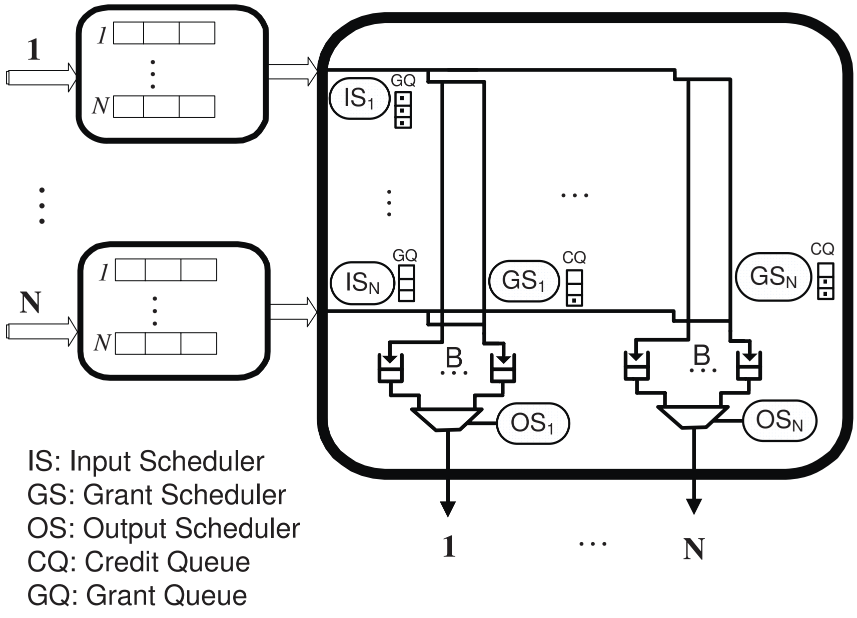


Figure 6.2: The Partially Buffered Crossbar (PBC) Switching architecture.

one per output. Each CQ contains B entries and CQ_j records the availability of the internal buffers belonging to output j. A CQ is decremented whenever a grant is sent to the input, and incremented during output scheduling.

6.2.2 Scheduling Process

The scheduling process in the PBC switch is a combination of unbuffered as well as buffered crossbar scheduling. A scheduling cycle consists of input scheduling and output scheduling phases as in buffered crossbars. The input scheduling phase resembles a scheduling cycle in unbuffered crossbars, as it is based on request-grant-accept handshaking protocol. The input scheduling phase works as follows: During time slot, t, each non empty $VOQ_{i,j}$ sends a request to the grant scheduler (GS) corresponding to output port j. Subject to internal buffers availability (CQ_j) and the grant scheduler policy, a grant may be sent back to the input scheduler i and stored in its GQ ($GQ_{i,j}$ set to 1). At the same time, input scheduler (IS_i) picks a VOQ Head of Line (HoL) cell to be transferred to the internal buffers based on its GQ, excluding the current grants of time slot, t. Meaning that the outcome of GS_i at time t is only valid during time slot $t + 1$ or later. This allows a two-stage pipelined scheduling, avoiding the need for synchronized coordination between the grant schedulers and the input (accept) schedulers on a time slot basis as does iSLIP [53] and PIM [39].

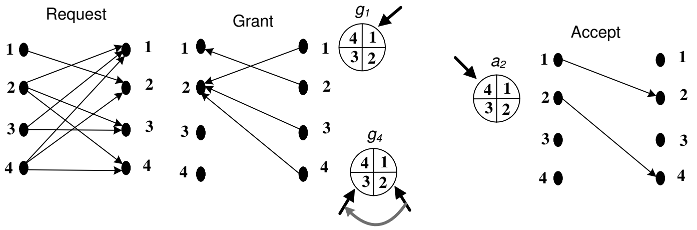


Figure 6.3: The iSLIP scheduling algorithm.

Because the number of internal buffers, B, maintained per output is much smaller than the number of competing inputs, N, crucial care to consider how to service cells during output scheduling is important. The internal buffers are separate and cells from the same VOQ may arrive to different internal buffers during consecutive time slots. In this case, we have to maintain in-sequence cell delivery. To this end, we employed a First-Come-First-Serve (FCFS) output scheduling to ensure in order cell delivery [107]. A cell departure from the internal buffers at output j, causes CQ_j to increments by one. A cell arrival, from an input, to an internal buffer at output j causes CQ_j to decrement by one. Likewise, the grant queue, at an input i, is incremented whenever a grant scheduler sends a grant to input i, and decremented whenever a cell departs the input port i.

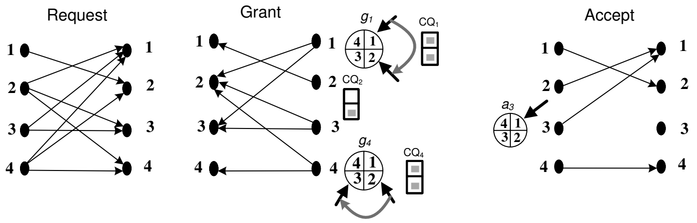


Figure 6.4: A PBC Scheduling cycle, 4×4 PBC switch with $B = 2$.

The input scheduling in PBC is similar to the iterative matching performed by unbuffered crossbar scheduling. However, maintaining a small number of in-

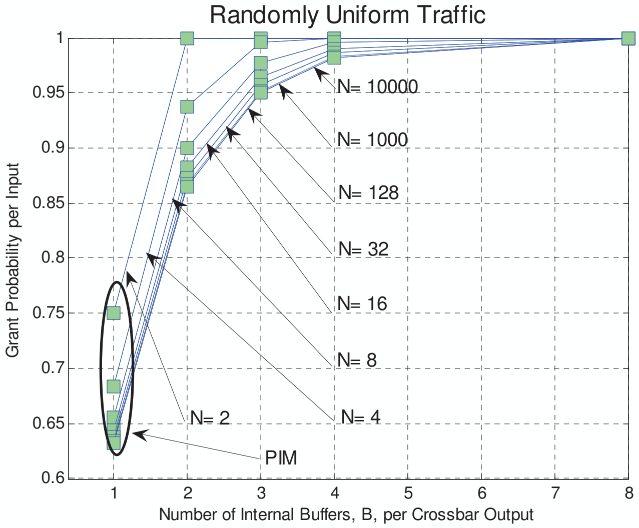


Figure 6.5: Grant probability as function of switch size, N, and different internal buffers settings.

ternal buffers makes it significantly different. The absence of internal buffers in an unbuffered crossbar switch meant that a grant arbiter can grant at most one input, to avoid output contention. Similarly, an input accept arbiter has to accept at most one grant, to avoid input contention. Figure 6.3 depicts the matching process of iSLIP with one iteration. The PBC scheduling, while keeping the input contention constraint enforced, relaxes the output contention constraint by allowing conflicting cells (up to B) to be admitted to the internal buffers for the same output. This is equivalent to unbuffered crossbars schedulers accepting one grant and storing other $B-1$ grants instead of discarding all the rest. Unbuffered crossbar schedulers resort to multiple iterations to improve the match size.

Figure 6.4 describes a PBC input scheduling cycle. As we can see, the grant scheduler at output 1, g_1, sends two grants (to input 1 and 3) because its credit queue, CQ_1, has two available credits. The same process happens with g_3 and g_4. However, g_2 sends only one grant because its output buffers have only one location free ($CQ_2 = 1$). From the example, we can see the benefit in using internal buffers, thereby improving the grant opportunities per output. Consequently, the accept phase produces a bigger match size. Using one iteration for a random scheduling policy such as PIM [39], the probability that an input will remain ungranted is $(\frac{N-1}{N})^N$, where N is the port count of the switch [53]. As N increases, this probability tends to $\frac{1}{e}$. If we use the same random scheduling policy in the PBC with B internal buffers per output and assuming that packets are flushed in every time slot (memoryless Markov process), the probability that an input remains ungranted is $(\frac{N-B}{N})^N$. With increasing N, this probability tends to $\frac{1}{e^B}$ (almost 0 for $B \geq 4$). Figure 6.5 illustrates this behavior. When $B = 1$ the PBC behaves identically to the bufferless PIM algorithm and the grant probability approaches 63% with increasing switch size. A small increase in B, just 2, scales up the grant probability to more than 86% for all switch sizes. When B is set to 4 per fabric output, the grant probability is 100%.

6.3 Scheduling in PBC Switches

This section introduces our set of scheduling algorithms for the PBC switching architecture. We propose a class of round robin based input scheduling algorithms. The output scheduling we use here and throughout this chapter is based on FCFS policy, as discussed in the previous section. Each time the output scheduler, at output j, performs its FCFS selection and sends a cell to the output queue, it increments CQ_j by one. As for the input scheduling, we propose a set of round robin based schedulers. The first algorithm we propose is named Distributed Round Robin (DRR) and is described in the following section.

6.3.1 The Distributed Round Robin (DRR) Algorithm

The DRR algorithm is similar to the scheme proposed by [106] and its grant scheduler's pointer update is the same as iSLIP [53]. This is because, a grant sent by a GS to an input scheduler, if not immediately accepted, is stored and will eventually get accepted in the short run (less than N time slots later). The DRR differs in the way it assigns cells to internal buffers when they leave the input VOQs. However this is specific to the PBC architecture. Cell assignment to internal buffers can be realized by maintaining a separate field in each entry of the GQ. Whenever the GS_j grants to input i, it sets the entry $GQ_{i,j} = 1$ and the field corresponding to the internal buffer to the index of the next free internal buffer $B_{k,j}$.

The DRR algorithms performs its arbitration as described in Section 6.2.2. Figure 6.6 illustrates a DRR input scheduling phase. The first pipeline stage of

```
DRR:
Grant Phase:
For each output, $j$, do
    . While there are credits in $CQ_j$ do
        - Starting from the grant pointer $g_j$ index, send a grant to the first
          input, $i$, that requested this output (set $GQ_{i,j} = 1$).
        - Decrement $CQ_j$ by one.
        - Move the pointer $g_j$ to location $(i + 1)\ (mod\ N)$.

Input Scheduling Phase:
For each input, $i$, do
    . Starting from the input pointer $a_i$ index, select the first non empty
      $VOQ_{i,j}$ for which $GQ_{i,j} = 1$ and send its HoL cell to the internal buffer.
    . Set $GQ_{i,j} = 0$.
    . Move the pointer $a_i$ to location $(a_i + 1)\ (mod\ N)$.
```

the algorithm starts as follows: Based on the VOQs requests (request phase) and the credit queues (CQ), each grant scheduler performs its arbitration. As shown in Figure 6.6, GS_1 and GS_4 can each issue two grants because their credit queues have available credits (CQ_1 and CQ_4). GS_1 receives requests from inputs: 2, 3 and 4. It grants to input 2 and 3 because its pointer is at position 1. After granting input 2 and 3, GS_1 points to output 4, as in iSLIP. GS_4 does the same, by granting to input 2 and 4 respectively. However, GS_2 and GS_3 (not shown in the figure) perform differently in this example because of their credit queues. GS_2 grants only to input one because its credit queues, CQ_2 contains only one credit. The other credit of output 2, is held by input 4 (second entry of GQ_4 = 1). Output 3 cannot issue any grant because it has no credits (CQ_3 = 0) and its credits are held by input 1 and 4. The outcome of the grant scheduler will not be taken into account by the input scheduler until the next time slot. This is shown in Figure 6.6 by dashed entries in GQ_1 and GQ_4 (second entry of GQ_1 and last entry of GQ_4). Simultaneously with the first pipeline phase, the second pipeline phase is executed as follows: Based on the grant queues so far, each input scheduler, i, selects the next non zero entry,

j, of its GQ_i and sends the corresponding HoL cell of $VOQ_{i,j}$ to the internal buffer. It also updates its GQ, by resetting the entry of the sent cell (both third entries of GQ_1 and GQ_1 in accept phase are reset). Then, it increments its pointer by one Mod (N). We can see in this example that output 3 receives 2 cells simultaneously, this is the advantage of the PBC architecture –allowing conflicting cells to enter the fabric.

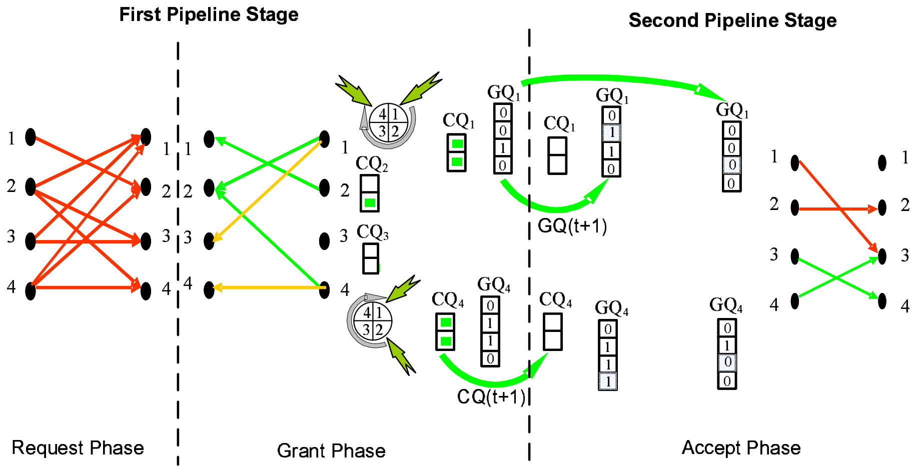


Figure 6.6: A DRR scheduling phase for a 4×4 PBC switch with $B = 2$.

6.3.2 The Credit Release Delay

The DRR scheme experiences the same credit release delay as with the scheme in [106]. Credit release delay arises when multiple grant schedulers grant to the same input concurrently. Because DRR returns credits one at a time (input contention), credits may not return fast enough. This, consequently, affects the rate at which grants are sent back to other inputs, hence delaying the transfer of cells from the input line cards. In order to reduce this delay, a grant throttling mechanism was proposed in [106]. It consists of setting a threshold (*TH*) for the grant queue and requests from an input are eligible so long as the grant queue relative to their input is less than *TH*. While this mechanism speeds up the credits release, it does not completely eliminate it or minimize it. Additionally, it requires some extra signaling to control the grant queues thresholds.

Our solution to credits release delay is different. We do not want to just lower the credit release delay. Instead, our goal is to completely eliminate it or set it to its absolute minimum. First, we need to quantify this delay. A grant has to wait up to N time slots in each grant queue before its associated credit is released back. Therefore, an input request waits at most $\frac{N^2}{B}$ time slots before it gets granted. To better explain this, consider the scheduling example depicted in Figure 6.6. Input 3 has a request for output 3. However, output 3 has no credits because they are already held by input 1 and 4 respectively. So, in the worst case output 3 will grant to input 3 after each credit is released by all other inputs (grant queue updated). This can take up to 4 time slots in each of the 4 grant queues. However, since we have 2 credits per credit queue, they are always divided among requesting inputs. Therefore, output 3 will grant to input 3 no later than $\frac{4^2}{2} = 8$ time slots since input 3 first issues its request.

When $B = 1$, the performance of DRR is similar[1] to iSLIP (see Figure 6.11 and Figure 6.12). However, as B increases, the credit release delay decreases ($\frac{N^2}{B}$ decreases). Thus, the problem of credit release delay is now reduced to solving the grant queueing delay. Minimizing the credit release delay means altering the grant mechanism to reduce the grant queueing delay. We, therefore, modified the way DRR allocates grants per input, hence a new grant scheduler is proposed. Instead of *storing* the grants, that are not accepted, in a grant queue while they wait their turn to be accepted, and therefore causing credits waiting (delayed) to get released, we simply *drop* them. The new devised scheme never stores grants, instead they are just dropped and the scheme is named DROP. Proceeding this way, a request waits no more than N time slots before it gets granted. This is to be compared to $\frac{N^2}{B}$.

6.3.3 The DROP Algorithm

Dropping the not accepted grants implies that the pointer updating scheme of the grant scheduler has to change. This is because, otherwise, using the iSLIP pointer updating mechanism results in pointer synchronization similar to RRM [53]. This convergence of iSLIP to RRM, under our settings, comes from the two-stage pipeline scheduling relative to the PBC architecture. Recall that DRR stores the unaccepted grants, guaranteeing their immediate or soon acceptance, and therefore the grant pointer can safely be updated. This is similar to iSLIP, where the grant pointer gets updated only on accept (which is in the same time slot or stage). With the DROP scheme, however, dropping the unaccepted grants during the accept phase (second pipeline stage) means that we have to update the grant pointer (first stage) which is already late and out

---

[1] The slight difference observed results from the the DRR input scheduler pointer update mechanism. Unlike iSLIP, DRR input scheduler pointer is fully unsynchronized, as in [54].

of date. Therefore, using the iSLIP pointer update mechanism in DROP means ‘blindly’ updating the grant pointers, which leads to synchronization and poor performance as in RRM [53]. To overcome this problem, we use fully unsynchronized grant pointers settings, similar to [54]. The grant pointers are initially set to different positions and are always incremented by one irrespective of the accept/drop outcome. The specification of the DROP algorithm is as below.

```
DROP:
Grant Phase:
All output pointers, $g_j$, are initialized to different positions.
For each output, $j$, do
  . Set $CQ_j$ equals to the number of non full internal buffers for output $j$.
  . While there are credits in $CQ_j$ do
      - Starting from $g_j$ index, send a grant to the first input, $i$, that requested this output (set $GQ_{i,j} = 1$).
      - Decrement $CQ_j$ by one.
  . Move the pointer $g_j$ to location $(g_j + 1)\ (mod\ N)$.

Input Scheduling Phase:
All input pointers, $a_i$, are initialized to different positions.
For each input, $i$, do
  . Starting from $a_i$ index, select the first non empty $VOQ_{i,j}$ for which $GQ_{i,j} = 1$ and send its HoL cell to the internal buffer.
  . Drop the remaining grants (reset GQ: $GQ_{i,*} = 0$).
  . Move the pointer $a_i$ to location $(a_i + 1)\ (mod\ N)$.
```

6.3.4 The Prioritized DROP Algorithm

We wanted to further improve the performance of the DROP scheme. The need to enhance DROP stems from two reasons. From one hand, DROP works in a two-stage pipeline meaning that it experiences some initial delay. On the other, DROP is designed for the PBC architecture that is assumed to have a limited small number of internal buffers per output, B. With these observations, we propose an enhanced version of DROP that services grants based on output priority. We call this version prioritized DROP and refer to it as DROP-PR. The specification of the DROP-PR scheme is as follows:

```
DROP-PR:
Grant Phase:
All output pointers, $g_j$, are initialized to different positions.
For each output, $j$, do
    . Set $CQ_j$ equals to the number of non full internal buffers for output $j$.
    . Set the priority bit, $P$, to the logic OR of $CQ_j$ entries.
    . While there are credits in $CQ_j$ do
        - Starting from $g_j$ index, send a grant to the first input, $i$, that requested this output (set $GQ_{i,j} = 1$ and add bit P).
        - Decrement $CQ_j$ by one.
    . Move the pointer $g_j$ to location $(g_j + 1)\ (mod\ N)$.

Input Scheduling Phase:
All input pointers, $a_i$, are initialized to different positions.
For each input, $i$, do
    . Starting from $a_i$ index, select the first non empty $VOQ_{i,j}$ for which $GQ_{i,j} = 1$ and bit $P = 1$ and send its HoL cell to the internal buffer.
    . If no HoL cell is selected, Then
        - Starting from $a_i$ index, select the first non empty $VOQ_{i,j}$ for which $GQ_{i,j} = 1$ and send its HoL cell to the internal buffer.
    . Drop the remaining grants (reset GQ: $GQ_{i,*} = 0$).
    . Move the pointer $a_i$ to location $(a_i + 1)\ (mod\ N)$.
```

When selecting a cell for input scheduling, the DROP-PR scheme takes into account the occupancy of the internal buffers belonging to an output. When a grant scheduler grants an input request, it sends back the grant with an additional priority bit. The priority bit informs the granted input whether or not the grant comes from an output with empty internal buffers (prioritized output). During the input scheduling phase (second pipeline stage), the input scheduler first gives priority to grants where the priority bit is set to 1. The priority bit is obtained by logically OR-ing the entries of the Credit Queue (CQ).

6.4 Performance Results

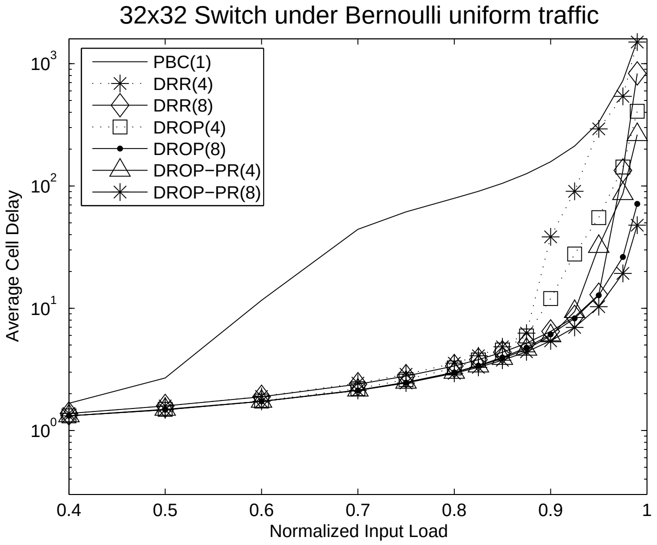


Figure 6.7: Average cell delay of the PBC algorithms under Bernoulli uniform traffic.

This section presents the performance study of the PBC switching architecture. The study is aimed at comparing our proposed architecture to both the unbuffered and the buffered crossbar fabric architectures as well as an ideal Output Queued (OQ) switch. The experiments are carried out under three input traffic patterns: Bernoulli uniform, Bursty uniform and Unbalanced traffic as defined in Appendix B.2. We tested different PBC switch sizes, each with different numbers of internal buffers [2]. In this section, we present the results

---

[2] Extensive simulations have been carried out for switch sizes of 8×8, 16×16, 32×32 and

of switch sizes of 16×16 and 32×32 only.

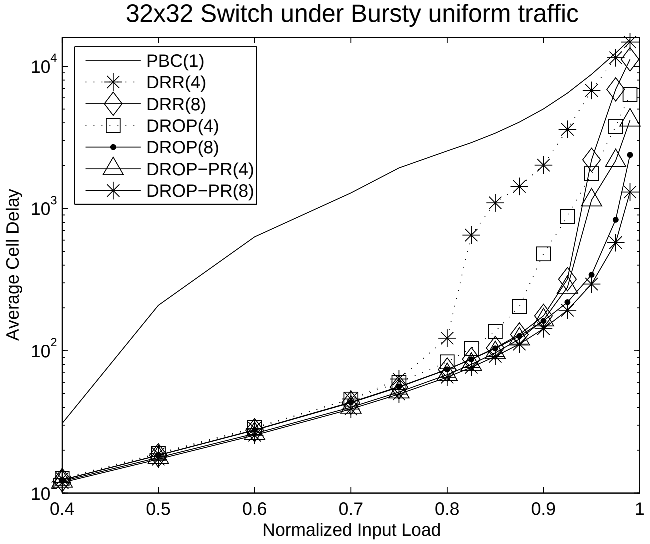


Figure 6.8: Average cell delay of the PBC algorithms under Bursty uniform traffic.

6.4.1 Uniform Traffic

Figure 6.7 illustrates the average cell delay performance of each of our proposed algorithms under Bernoulli uniform traffic. We measured the delay of each of the algorithms with different internal buffer settings. When the number of internal buffers per output, B = 1, the three algorithms all have the same delay because there is no credit release delay. For this reason we denote any of the algorithms by "PBC(1)" in the figure. Increasing B to as few as 4 internal buffers per output boosts up the performance by an order of magnitude. The delay improvement is less sharp for B between 4 and 8. This behavior agrees with our model (refer to Figure 6.5) discussed in Section 6.2.2. When B = 4 or more, the granting likelihood is almost 100% from each grant scheduler to each input scheduler. The same behavior is observed under Bursty uniform arrivals, as depicted in Figures 6.8.

Assessing the performance of each of our three proposed algorithms (DRR,

---

64×64. Depending on each PBC switch size, different internal buffer sizes, B, have been used (1,2,3,4,5,6,8,10,12,16,32) under each scheduling algorithm (DRR, DROP and DROP-PR).

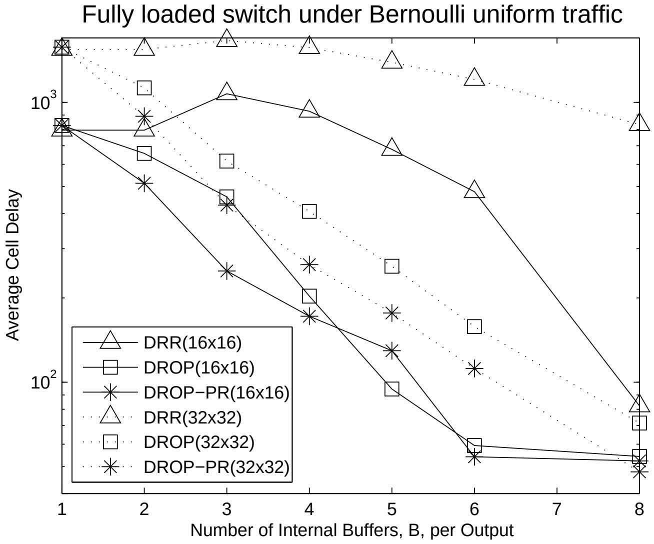


Figure 6.9: PBC Performance under Bernoulli uniform arrivals.

DROP and DROP-PR respectively) requires tuning different parameters such as switch size, internal buffers sizes and input traffic loads, hence many plots. However, because we are most interested in switch cell delays under heavy input loads, in the following two figures (Figure 6.9 and Figure 6.10) we fixed the input load to be 99% and varied the switch size as well as the number of internal buffers per output for each algorithm. Figure 6.9 depicts the performance of each of our algorithms under Bernoulli uniform arrivals for a 16×16 and 32×32 respectively. We observed the average cell delay as a function of the number of internal buffers per output, B. We can see that when $B = 1$, unbuffered crossbar switch, the three algorithms have the same delay which is comparable to iSLIP with one iteration. This is because when $B = 1$, every request waits for the same time (N^2 times slots at most) before it gets served.

However, with increasing B, both DROP and DROP-PR have lower cell delays than DRR because of their fast credits release. Recall that the credit release delay (and consequently grant and service delays) of DRR is $\frac{N^2}{B}$. However, both DROP and DROP-PR have a credit release delay of N. As B increases, (especially as B approaches N, not shown in the Figures) all the algorithms have the same delay. However, we are only interested in PBC switches with $B \ll N$. The same trend is observed also under bursty arrivals, Figure 6.10. The delay of DRR decreases faster under Bernoulli uniform than under Bursty

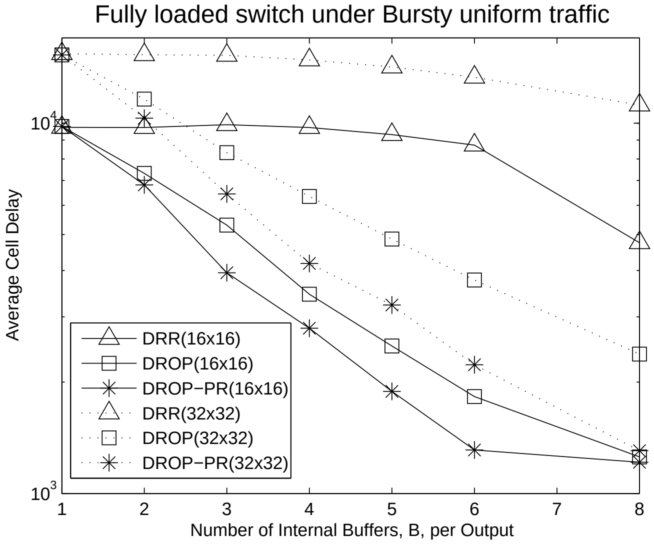


Figure 6.10: PBC Performance under Bursty uniform arrivals.

arrivals because of the burstiness effect. DROP-PR has the overall lowest delay because it prioritizes outputs with empty internal buffers, resulting in more balanced internal buffers occupancies and hence lower cell latencies.

We compared the average cell latency of the DROP-PR algorithm for 32×32 PBC switch to that of an unbuffered crossbar switch, a fully buffered crossbar switch and an ideal OQ switch. The iSLIP algorithm is used for the unbuffered crossbar architecture. The fully buffered crossbar switch uses input round robin (RR) scheduling and Oldest Cell First (OCF) output scheduling. The comparison is performed under uniform Bernoulli and Bursty arrivals. Figure 6.11 depicts the performance of DROP with different internal buffers, iSLIP (with 1 and 4 iterations), RR_OCF and OQ. Irrespective of whether the input traffic is Bernoulli or Bursty, DROP-PR(1) (1 refers to $B = 1$) has a similar behavior to 1SLIP, as described earlier (see Section 6.3). As B increases, the delay of DROP-PR significantly decreases. It approaches that of an ideal OQ with just 8 internal buffers per output ($B = 8$). Similar performance is observed under bursty arrivals (Figure 6.12). These results suggest that a PBC switch can replace a buffered crossbar, or even an ideal OQ switch with as few as 8 internal buffers per output. These results can also afford a switch designer the choice depending on the constraints and needs. For example, if the delay-cost product is the main target, there is the option to replace

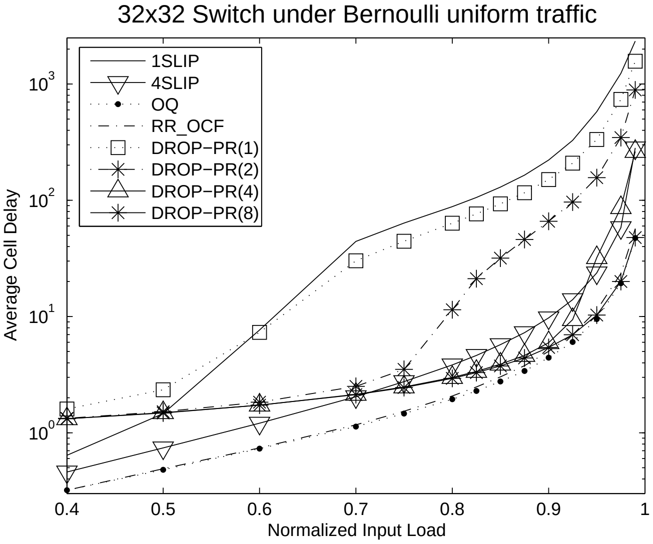


Figure 6.11: Performance under Bernoulli uniform arrivals.

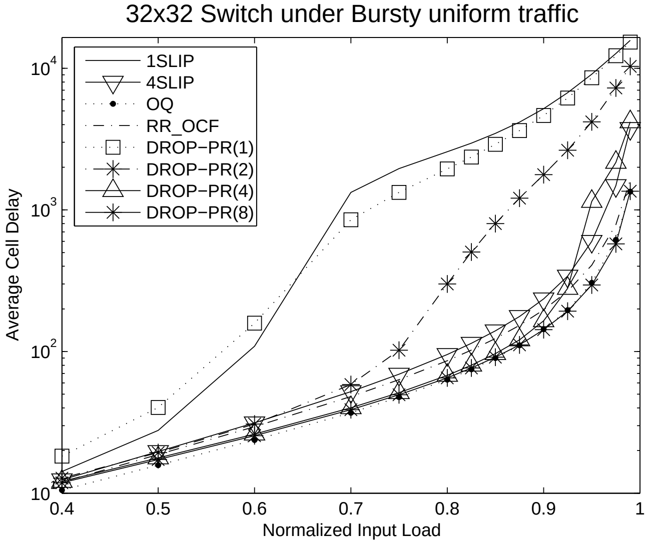


Figure 6.12: Performance under Bursty uniform arrivals.

an unbuffered crossbar switch employing 4SLIP with a PBC switch with only 4 internal buffers per output (see the delay performance of 4SLIP and DROP-

PR(4) in Figure 6.11 and Figure 6.12 respectively). However, if performance is the main criteria with a little flexibility in cost, one can then employ a PBC switch with 8 internal buffers per output since it exhibits ideal performance.

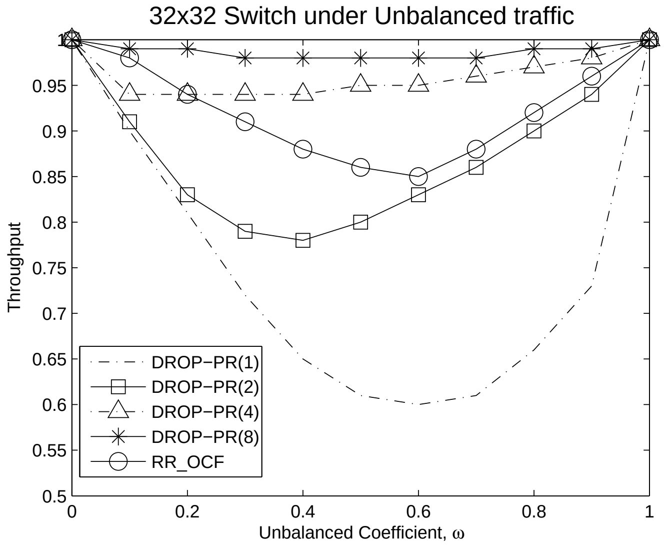


Figure 6.13: Throughput performance under Unbalanced traffic.

6.4.2 Unbalanced Traffic

To further endorse our claims with respect to the PBC performance, we analyzed the stability of a 32×32 PBC switch under unbalanced traffic arrivals (see Appendix B.2). We employed the unbalanced traffic proposed in [67]. We set the switch input load at 100% and varied the unbalanced coefficient, ω and observed the switch throughput performance. Figure 6.13 depicts the performance of the PBC switch with DROP-PR algorithm and different internal buffer settings and compares it to a buffered crossbar (using RR_OCF scheduling). We can see that, with $B = 2$ internal buffers per output, we can achieve comparable throughput to a fully buffered crossbar when the unbalanced coefficient, ω, is higher than 0.6. This translates to a saving worth of up to 960 internal buffers. Setting $B = 4$, we can achieve higher throughput than a fully buffered crossbar. The ideal throughput of the PBC is reached when using 8 internal buffers per output.

6.5 Summary

A novel Partially Buffered Crossbar (PBC) switching architecture is proposed in this chapter. The PBC switch is designed to be the best compromise between unbuffered crossbars and fully buffered crossbars. On one hand, it overcomes the high cost of fully buffered crossbars that use N^2 internal buffers, by using a low number of internal buffers per output irrespective of N. On the other hand, it overcomes the scheduling complexity experienced by unbuffered crossbars by means of distributed and pipelined scheduling algorithms. We proposed a class of distributed and pipelined round robin scheduling algorithms for the PBC architecture. In particular, the DROP-PR scheme was shown to have optimal performance under different traffic patterns and switch sizes.

The experimental results showed that a PBC switch with 8 internal buffers per output exhibits ideal performance, irrespective of the switch size, N. We also showed a design trade off between using the bufferless iSLIP algorithm with 4 iterations, or using the PBC with 4 internal buffers per fabric output. This trade off affords a switch designer wider performance and cost based choices. We believe that the PBC architecture has good potential to become the architecture of choice for next generation routers. The reason for this is not only because the PBC achieves the best of both the unbuffered and fully buffered crossbars, but also because it can provide the opportunity to implement optimal bufferless scheduling algorithms in a pipelined and distributed fashion.

Chapter 7

Conclusions

Buffered crossbar (CICQ) switches are considered viable and practical architectures for the design of high performance routers. A CICQ switch has good potential in overcoming the scheduling bottleneck experienced by alternative switching architectures. However, the scheduling simplicity comes at the cost of an expensive and complex buffered crossbar fabric chip in terms of on chip memory and flow control signaling. Additionally, the CICQ switching has so far been studied only in the context of unicast traffic scheduling.

This dissertation studies the CICQ switching architecture and addresses its scalability and performance issues. To address the scalability limitations, we have proposed a CICQ switching architecture where the schedulers are all embedded within the buffered crossbar fabric chip. This results in an optimized flow control mechanism and allows the design of scalable switching. We showed that this architecture is able to provide performance guarantees. We have also studied the problem of multicast as well as the integration of unicast and multicast flows scheduling for the CICQ architecture. To reduce the internal memory limitation, we proposed a partially buffered crossbar switching architecture, wherein only a small fixed number of internal buffers are used inside the buffered crossbar fabric chip.

This chapter is structured in three sections. Section 7.1 summarizes the work presented in this dissertation. In Section 7.2, we present the mains contributions of the dissertation. Finally, Section 7.3 lists some future directions and open issues worthy of further research and investigation in the context of buffered crossbars.

7.1 Summary

The dissertation begins by providing the background of the work. A survey of existing switching architectures was given in Chapter 2. The survey focussed on the IQ crossbar fabric switching architecture because of its similarity to the CICQ switch. We have described the CICQ switching architecture and outlined its main limitations, motivating the work in this dissertation.

Chapter 3 describes the design and implementation of a set of embedded schedulers within the buffered crossbar fabric chip. This is a novel class of algorithms, where the arbitration process is fully based on the internal buffer information. This was motivated by the observation that the buffered crossbar fabric chip is I/O pin count constrained, implying the existence of unused area on the chip. When the schedulers are located inside the crossbar chip, there is no longer a requirement for the flow control to carry the availability of every crosspoint. Instead, the index of a new arriving cell will be forwarded to the crossbar chip, resulting in optimized flow control between the crossbar fabric chip and the input line cards. For a 32×32 switching system, our embedded CICQ switching architecture achieves up to 70% saving of chip I/O flow control pins when compared to existing CICQ switch architectures. This has the benefit of speeding up the scheduling time while using a *limited* number of flow control signals, resulting in more scalable crossbar switches. It also improves the performance of the scheduling algorithms, since there are many algorithms that base their decisions on the internal buffers and when embedded within the crossbar chip would have faster decisions and cheaper access to resources. The experimental results showed that our set of algorithms outperform existing algorithms under various traffic settings. To show the feasibility of the embedded scheduling architecture, we implemented a 24×24 buffered crossbar core with each port running at 10 Gbps, achieving an aggregated switching bandwidth of 240 Gbps.

Although our devised embedded schedulers were shown to provide high performance under a wide range of unicast traffic patterns, they do not provide performance guarantees. In Appendix A, we devised a set of embedded schedulers for a buffered crossbar that can mimic an ideal OQ switch. We showed that our proposed fabric, when running twice as fast as the external line rate, can emulate an ideal output queued switch. Our results applied to the class of OQ switches that use FIFO output scheduling discipline.

Chapter 4 addresses the problem of multicast traffic flows scheduling. We proposed a multicast buffered crossbar switching architecture based on input FIFO queues along with appropriate scheduling. We showed that our architecture exhibits better performance than existing architectures. We further improved our multicast switching architecture by using a small number of input queues per port of the switch. We devised a multicast cell assignment algorithm to map incoming multicast traffic to the input queues. Our algorithm was shown to assign traffic more efficiently, fairly and faster than existing algorithms. We used simple round robin for cell scheduling and showed its superiority to alternative proposals. Our study showed an interesting trade off between the number of input multicast queues and the size of the internal buffers. This results not only in better performance, but also in significantly reduced scheduling complexity, hence faster and more scalable switching.

In Chapter 5, we proceeded to scheduling more realistic traffic flows: the combination of unicast and multicast traffic. We proposed a buffered crossbar based architecture, along with the appropriate scheduler, that efficiently supports both unicast and multicast flows. Our scheduler, while based on a fanout splitting policy, tends to not exhaust the serial links between the line cards and the fabric core when servicing multicast traffic. We employed plain round robin scheduling, both at the traffic level as well as the queue level, and showed higher performance than previous algorithms. In order to verify the feasibility of our design, we implemented our integrated scheduler for a 16×16 switching system running a 20 Gbps port speed, allowing every switch port to forward more than 47 million ATM cells per second.

Chapter 6 describes a novel variation to the CICQ switching architecture that overcomes the buffered crossbar internal memory limitation. We proposed a *partially* buffered crossbar switching architecture that is designed to be a good compromise between the two extreme cases of unbuffered crossbars and fully buffered crossbars. The proposed partially buffered crossbar is based on few internal buffers per fabric output, making its cost comparable to unbuffered crossbars. It overcomes the centralized IQ crossbar scheduling bottleneck by using distributed and pipelined schedulers, as in fully buffered crossbars, making it a practical and low cost architecture for ultra high capacity networks.

7.2 Contributions

The main contributions of this dissertation are as follows:

- **Unicast Scheduling:** We have designed and implemented a novel class of unicast scheduling algorithms for the CICQ switching architecture.

These algorithms make their scheduling decisions based only on the internal buffers information. We have proposed an embedded scheduling architecture, where all the input and output schedulers are embedded within the buffered crossbar fabric chip. Embedding the schedulers inside the crossbar results in *optimized flow control* between the crossbar fabric chip and the input line cards. We showed that, for a switch with N ports, a flow control of $2N\log N$ signals is sufficient for efficient scheduling as opposed to using N^2 flow control signals. For a typical 32×32 switching system, our proposed embedded CICQ switching architecture achieves up to 70% saving in chip I/O flow control pins when compared to existing CICQ switch architectures. This has the benefit of speeding up the scheduling time while using a *limited* number of flow control signals, resulting in more scalable buffered crossbar switches.

- **Providing Performance Guarantees:** we described a set of embedded schedulers for a buffered crossbar that can mimic an ideal OQ switch. We showed that our proposed buffered fabric, using a speedup of two, can emulate an ideal FIFO OQ switch.

- **Multicast Scheduling:** We studied the problem of multicast traffic flows scheduling in CICQ. We proposed a multicast FIFO based buffered crossbar switching architecture along with appropriate scheduling. We showed that our architecture outperforms existing architectures. We further extended our multicast switching architecture and used a small number of input queues per port of the switch. We devised a multicast cell assignment algorithm to map incoming traffic to input queues. Our algorithm was shown to assign traffic more efficiently, fairly and quickly than existing algorithms. The experimental results showed that the number of input multicast queues can be reduced for a little increase in the size of the internal buffer memory per crosspoint. This results in higher switching performance in terms of cell delay as well as reducing the scheduler complexity, providing faster and more scalable switching.

- **Integration of Unicast and Multicast Flows:** We studied the problem of multicast traffic flows scheduling in CICQ switches. We proposed, designed and implemented an integrated scheduling algorithm capable of scheduling unicast and multicast flows simultaneously. Our design was implemented for a 16×16 CICQ switch running a 20 Gbps port speed, resulting in every switch port capable to forward more than 47 million ATM cells per second.

- **Partially Buffered Crossbar Switches:** We proposed a novel variation to the CICQ switching architecture that overcomes the buffered crossbar excessive internal memory requirement. We proposed a *partially* buffered crossbar switching architecture that is designed to be a good compromise between the two extreme cases of unbuffered crossbars and fully buffered crossbars. The proposed partially buffered crossbar is based on few internal buffers per fabric output, making its cost comparable to unbuffered crossbars. It overcomes the centralized IQ crossbar scheduling bottleneck by using distributed and pipelined schedulers as in fully buffered crossbars making it a practical and low cost architecture for high speed and capacity networks. Experimental results suggested that using 8 internal buffers per crossbar output is sufficient to achieve ideal performance for any switch size, N.

7.3 Future Research Directions

The increasing need for terabit switches and routers means that future commercial packet switches must be implemented with reduced scheduling complexity, low cost and scalability while providing performance guarantees. In this dissertation, we have proposed several schemes for the buffered crossbar architecture in order to meet next generation routers requirements. We conjecture that there are a number of other research directions that require more investigation. These directions include the following:

- **Embedded Multicast Scheduling:** The work in this dissertation has focussed on embedding only unicast scheduling algorithms inside the buffered crossbar chip. It would be interesting to explore implementing a scheduler capable of supporting all types of traffic flows (including multicast and/or the integration of both unicast and multicast flows) inside the buffered crossbar fabric. This would provide scalability of the buffered crossbar fabric architecture irrespective of the traffic type. Regarding the integrated traffic scheduling; although in this dissertation we considered multicast fanout-splitting policy due to its high throughput, adopting non fanout-splitting policy is better as it is more bandwidth efficient. We believe that a carefully designed CICQ switch architecture, owing to the existence of its internal buffers, can adopt non fanout-splitting discipline without throughput degradation. One way to address this is by dedicating internal buffers for multicast traffic and crossing every multicast packet only once over the serial links between the input line cards and the buffered crossbar core.

- **Providing Performance Guarantees:** Current VLSI technology is capable of allowing one cell per crosspoint when the number of the switch ports is less than one hundred. The size of memory that can fit on chip is expected to grow. It would be interesting to investigate the possibility of providing performance guarantees (OQ emulation) with a speedup less than two, but with internal buffers of bigger sizes.
- **Partially Buffered Crossbar Switches:** The partially buffered crossbar switching architecture is perhaps the most promising of all. The work in this dissertation has mainly focussed on introducing the concept of the partially buffered crossbar architecture by testing and validating its performance. We believe that all the future directions outlined above can be incorporated in one, practical and optimal architecture such as the partially buffered crossbar. One very important issue is to investigate whether it is possible to map the bufferless MWM optimal scheduling algorithms to the partially buffered crossbar architecture and implement them in a pipelined and distributed fashion. It has so far been not possible to achieve this task for fully buffered crossbars and the reason is mainly attributed to the physically distributed internal buffers as well as their scheduling. We believe that the partially buffered crossbar architecture with its efficient internal buffers sharing, can not only practically run optimal scheduling, but can also provide throughput, rate and delay guarantees as well.

Appendix A

Output Queued Switch Emulation

O**utput**-Queued (OQ) switches are known to be of optimal performance amongst all queueing approaches. However, an OQ switch is not scalable due to the high memory bandwidth limitation (see Section 2.3.3). While it has been shown that we can emulate an OQ using a more scalable crossbar switch (i.e., Input-Queued (IQ) switch) and a small speedup, the algorithms proposed were impractical (see Section 2.3.5). More recent work has shown that a conventional CICQ can practically emulate an OQ switch (see Section 2.5.2). In this appendix, we show a similar result to the related work presented in Section 2.5.2, however with different architecture and scheduling. In particular, we propose the following:

- A 1-cell internally buffered CICQ switch with embedded schedulers (as proposed in Chapter 3) that can exactly emulate a FIFO OQ switch. The embedded CICQ switch core has a speedup of two.
- A set of embedded scheduling algorithms. The input scheduling is named Most Current Arrival First (MCAF) with an output scheduling scheme named Lowest Time to Leave First (LTF). Our algorithm, MCAF_LTF, particularly its output scheduler is simpler than previously proposed algorithms.

The remainder of the appendix is structured as follows: Section A.1 describes the architecture and illustrates the definitions used thereafter. Section A.2 presents the MCAF_LTF scheme and provides sufficient proof for OQ emulation by CICQ switch with a speedup of 2.

A.1 Switch Model and Definitions

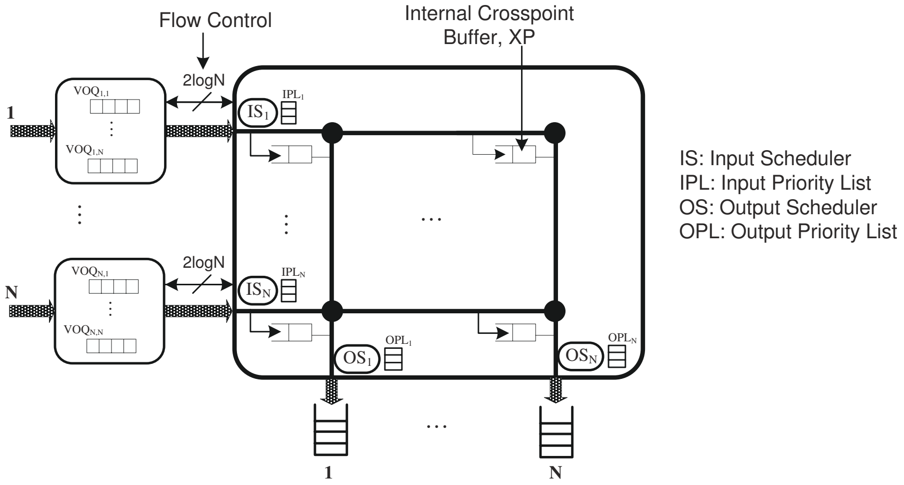


Figure A.1: CICQ Switching architecture with embedded schedulers and output queues.

The embedded CICQ switching architecture that we propose here is similar to the architecture we described in Section 3.3. However, since we use a speedup of 2, queueing is now required in the inputs as well as in the outputs, as depicted in Figure A.1. With a speedup of 2, each time slot is divided into 4 phases as depicted in Figure A.2. These phases are described below:

- *Arrival phase:* All arrivals occur during this phase.
- *First scheduling phase:* a scheduling cycle is performed during this phase.
- *Second scheduling phase:* a second scheduling cycle is performed during this phase.
- *Departure phase:* All cell departures occur during this phase. The end of this phase coincides with the end of a time slot.

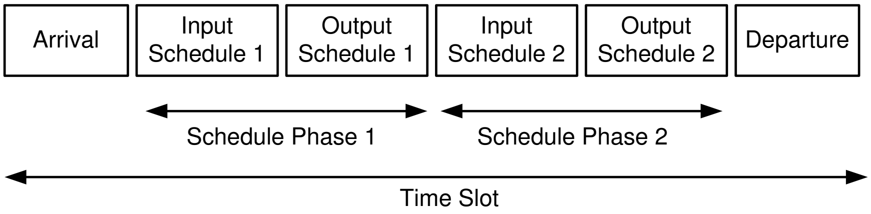


Figure A.2: Scheduling phases in an embedded CICQ Switch.

In the following section, we provide some definitions necessary for the derivation of the OQ emulation. These definitions are similar to those presented in [35]:

1. **Input Priority List (IPL):** Each input scheduler maintains an input priority list, IPL, of all cells queued in its corresponding input port. The IPL determines the departure order of cells from the input to the internal buffers.

2. **Shadow OQ Switch:** A theoretical OQ switch that determines the departure order and time of each cell from the CICQ to emulate an OQ.

3. **Time-to-Leave (TTL):** Equals the departure time slot of cell c, specified by the shadow OQ switch. Note that all cells, destined for the same output, must have distinct TTLs.

4. **Output Priority List (OPL):** Each output scheduler, *j*, maintains an output priority list, OPL, of all cells queued at the column buffer $CXPB_j$. The OPL at an output scheduler is composed a FIFO queue and a PIFO queue. The ordering of cells in the FIFO and PIFO queues determines the departure order of cells from the internal buffers to the output queue. Cells are inserted in the PIFO queues based on their TTL field. A cell c, destined to a PIFO queue, is inserted ahead of all cells with a greater TTL and behind all cells with smaller TTL.

5. **Input Thread (IT):** The input thread of a cell c, *IT(c)*, is equal to the number of cells ahead of c in its input priority list. *IT(c)* is defined for each cell queued at an input port. It is influenced by the arrival and input scheduling phases. A newly arriving cell may cause *IT(c)* to increment. However, an input scheduling phase may cause *IT(c)* to decrement. If c is transferred to the internal buffers, its *IT(c)* becomes zero .

6. **Output Cushion (OC):** The output cushion of a cell c is equal to the number of cells at c's output queue with lower TTL than c. Unlike *IT(c)*, *OC(c)* is influenced by the output scheduling and the departure phases, respectively. An output scheduling phase may cause *OC(c)* to increment. Conversely, a departure phase may cause *OC(c)* to decrement. *OC(c)* does not change during an input scheduling phase.

7. **Slackness (L):** Every time slot, the slackness of cell c, *L(c)*, equals the output cushion of cell c minus its input thread. That is,

   $$L(c) = OC(c) - IT(c)$$

   The slackness is defined for cells queued either at an input port or at a crosspoint buffer .

The slackness of a cell c determines the urgency of c's transfer from its incoming port to its outgoing port [35]. Recall that emulating OQ means that every cell must reach its output queue on or before its time to leave as specified by the shadow OQ. The OQ emulation process is highly influenced by the slackness of every cell, c, inside the system. Any increase in *L(c)* is translated by either an increase in *OC(c)* or a decrease in *IT(c)*. In both cases, *L(c)* increases and there is no fear for c of reaching its output on time. Any decrease in *L(c)*, however, is translated by either a decrease in *OC(c)* or an increase in *IT(c)*. In both cases, *L(c)* decreases and c should be urgently transferred to its output queue before it misses its time to leave. As a result, in order for the OQ emulation to occur, we have to ensure that the slackness of every cell inside the switch is positive and non-decreasing.

During each time slot, every cell, c, can have one of the following statuses: just arrived, selected for input scheduling, not selected for input scheduling (blocked by a flow control), selected for output scheduling, not selected for output scheduling (blocked by a more urgent cell) or departed the switch. Note that we are no longer concerned about any cell that either reaches its output queue (i.e., the status: selected for output scheduling) or departed the switch. The OQ emulation takes place if, irrespective of its status, any cell, c, has a non negative slackness. In the following section, we propose a scheduling scheme along with its complete proof that a CICQ switch running at a speedup of 2 can exactly emulate an OQ switch. In particular, we will show that, upon its arrival, every cell, c, is inserted with a non-negative slackness. Then, ongoing, as long as the cell c is inside the switch and having one of the statuses of interest, its slackness never decreases.

A.2 FIFO Output Queueing Emulation

This section provides the specification of our proposed scheduling scheme along with the sufficient conditions that prove that, with a speedup of two, a CICQ switch can exactly emulate an OQ switch. The specification of each of scheduling phase is as follows:

Input Schedule: MCAF

Each input, i, maintains its IPL as follows:

- If there is a currently arriving cell, c, to a *VOQ*$_{i,j}$.
- Then insert c, just behind the last entry of *VOQ*$_{i,j}$ in the IPL.
- If *VOQ*$_{i,j}$ is eligible
    - . If *VOQ*$_{i,j}$ contains other cells than c

      Move the HoL cell of *VOQ*$_{i,j}$ to the front of IPL and assign c a priority flag ‘P’.
    - . Else, move the HoL cell of *VOQ*$_{i,j}$ to the front of IPL and assign c a priority flag ‘F’.
- Serve cells based on IPL order.

Output Schedule: LTF

Each output, j, maintains its OPL as follows:

- If cell c has a priority flag ‘P’
    - . Then, insert c into the *PIFO*$_j$.
    - . Else, insert c into the tail of *FIFO*$_j$.
- Move the HoL cell of *PIFO*$_j$ or *FIFO*$_j$ based on the Lowest TTL to the front of the OPL.
- Serve cells based on OPL order.

Lemma A.1. *The LTF output scheduling scheme ensures Lowest TTL (LTTL) scheduling property.*

Proof. Every cell, c, sent from an input $VOQ_{i,j}$ to a crosspoint buffer, $XP_{i,j}$, is inserted either at the tail of $FIFO_j$ or in the $PIFO_j$.

- Case 1: Cell, c, is inserted at the tail of $FIFO_j$
  - i) Cell, c, enters the switch at the current time slot.
  - ii) Similar to FIFO OQ, if simultaneous arrivals occur to the same FIFO, tie-breaking is used.
  - iii) For cell c', ahead of c in $FIFO_j$, $TTL(c') < TTL(c)$
  - iv) Combining (i), (ii) and (iii) implies: cells in $FIFO_j$ are ordered by their TTL, and the HoL cell of $FIFO_j$ has the lowest TTL.
- Case 2: cell, c, is inserted in $PIFO_j$
  - 1) By definition 4, inserted cells in the $PIFO_j$ are ordered by their LTTL.
  - 2) The LTF scheme compares the $PIFO_j$ HoL cell with $PIFO_j$ HoL cell and moves the cell with Lowest TTL to the front of the OPL.
  - 3) The LTF output algorithm serves cells based on the OPL order.

Combining (iv), (1), (2) and (3) proves the Lemma. ☐

Theorem A.1. *For a CICQ switch employing the MCAF_LTF scheduling scheme, the slackness of any cell, c, that does not yet reach its output queue, increases by at least 1 during each scheduling phase.*

Proof. We know that any cell, c, that does not yet reach its output queue can only be either at internal buffer or at an input queue. Therefore we have the two followings cases:

- Case 1: Cell, c, is queued at an internal buffer, $XP_{i,j}$
  - i) By Definition 7,

$$L(c) = OC(c) - IT(c)$$

    ii) By Definition 5, since cell c is queued at the internal buffer, then *IT(c)* $= 0$.
    iii) If c ends the scheduling phase at the internal buffer, we know that cell, c', such that *TTL(c')* $<$ *TTL(c)* has been selected for output scheduling (Lemma A.1). Hence, *OC(c)* increases by 1

- Combining (i), (ii) and (iii) yields: *L(c)* increases by 1. (1)

- Case 2: cell, c, is queued at an input queue,$VOQ_{i,j}$

  In this case, there are two possibilities: either $VOQ_{i,j}$ is eligible or it is backlogged.

  - Case a: $VOQ_{i,j}$ is eligible
    i) During the input scheduling phase either c is chosen or a cell c' ahead of c in the IPL is chosen to be transferred to the internal buffer.
    ii) If c is chosen, $IT(c)$ becomes zero, and therefore decreases at least by 1.
    iii) If c' is chosen, *IT(c)* decreases by 1.
    iv) By definition 6, *OC(c)* remains unchanged during input scheduling.
  - Combining (i), (ii), (iii) and (iv) yields: *L(c)* increases by 1. (2)
  - Case b: $VOQ_{i,j}$ is is backlogged by an internally queued cell c'
    i) Both c and c' belong to the same FIFO $VOQ_{i,j}$.
    Hence, *TTL(c')* $<$ *TTL(c)*.
    ii) During an output scheduling phase, either c' or c" such that:

$$TTL(c") < TTL(c') < TTL(c)$$

    is sent to the output. In either cases, *OC(c)* increases by 1.
    iii) During an input scheduling phase, *IT(c)* either decreases or remains unchanged.
  - Combining (i), (ii) and (iii) yields: *L(c)* increases by 1. (3)

- The combination of (1), (2) and (3) yields: the slackness of any cell, c, that does not yet reach its output queue, increases at least by 1 during each scheduling phase. Hence, the proof of Theorem A.1 is complete.

□

Theorem A.2. *Consider a CICQ switch with a speedup of 2 that employs MCAF_LTF scheduling policy. For every time slot and for every cell, c, that does not yet reach its output queue, the slackness never decreases.*

Proof. For the CICQ switch operating at speedup of 2, the time slot is divided into an arrival phase, two scheduling phases and a departure phase.

i) During an arrival phase, *IT(c)* can increase by at most 1 (in case the newly arriving cell is more urgent than c). The possibility that *IT(c)* increases by at most 1 causes *L(c)* to decrease by 1.

ii) During a departure phase, *OC(c)* decreases by exactly 1, since a cell in its output queue left the switch. The decrease of *OC(c)* by 1 causes *L(c)* to decrease by exactly 1.

iii) From Theorem A.2, we know that the slackness of any cell, c, that does not yet reach its output queue, increases at least by 1 each scheduling phase. Since we have a speedup of 2, every time slot contains two scheduling phases. Hence, every time slot, *L(c)* increases by at least 2.

Summing over (i), (ii) and (iii) results in a non decreasing slackness, *L(c)*. ☐

Now, as we proved that the slackness never decreases from time slot to the next, we need to ensure that any arriving cell, c, must be inserted into the IPL with a non negative slackness.

Lemma A.2. *The MCAF_LTF scheduling scheme satisfies the non-negative slackness (NNS) insertion property for a CICQ switch running at a speedup of 2.*

Proof. (*by induction*)
Suppose that lemma 2 held up until time slot t. We show that lemma 2 holds at time slot $t + 1$.
At time slot $t + 1$, a new arriving cell, c, can arrive to either an empty or a non empty VOQ, as follows:

- Case a: Cell c arrives to an empty *VOQ*$_{i,j}$
  Based on MCAF scheme, an empty VOQ to which arrival occurs become highest priority and has the following:

$$\begin{aligned} IT(c) = 0, OC(c) \geq 0, \ \ \textit{thus} : \\ L(c) = OC(c) - IT(c) \geq 0 \end{aligned} \tag{4}$$

- Case b: Cell c arrives to a non empty $VOQ_{i,j}$
  We know that the NNS property held up until the end of time slot t. Suppose that cell, c', behind which c is inserted had a positive slackness of *L*$_t$*(c')* at time t. From Theorem A.2, we know that the slackness never decreases from time slot to the next. This implies:

  $$\begin{aligned} L_{t+1}(c') &= OC_{t+1}(c') - IT_{t+1}(c') \\ &\geq L_t(c') \end{aligned} \tag{5}$$

  Cell c is behind c', that is:

  $$IT_{t+1}(c) = IT_{t+1}(c') + 1 \tag{6}$$

  Both, c and c', are destined to the same output and *TTL(c')* < *TTL(c)* implying:

  $$OC_{t+1}(c) \geq OC_{t+1}(c') + 1 \tag{7}$$

  (5), (6) and (7) imply:

  $$\begin{aligned} L_{t+1}(c) &= OC_{t+1}(c) - IT_{t+1}(c) \\ &\geq (OC_{t+1}(c') + 1) - (IT_{t+1}(c') + 1) \\ &\geq L_{t+1}(c') \\ &\geq L_t(c') \end{aligned} \tag{8}$$

Combining (4) and (8) results in a non-negative slackness insertion policy. Hence, the proof of Lemma A.2 is done. ☐

Having showed that a CICQ using MCAF_LTF scheduling scheme and a speedup of 2 satisfies the non-negative slackness insertion policy and a non-decreasing slackness from time slot to the next, we are ready to prove our main theorem.

Theorem A.3. *A CICQ switch employing the MCAF_LTF scheduling policy with two times speed up can exactly emulate a FIFO_OQ switch.*

Proof. (*By induction*)
Suppose that the CICQ has emulated a FIFO_OQ switch up until the departure phase of time slot t. We show that any cell, c, such that *TTL(c) = t +1* reaches its output queue on or before the second scheduling phase of time slot $t+1$ as follows:

- Case a: Cell c is queued at an internal buffer, $XP_{i,j}$
    - i) There are no cells left inside the switch with $TTL < t+1$.
    - ii) *TTL(c) = t +1* and the LTF scheme ensures LTTL scheduling (Lemma A.1).
- (i) and (ii) result in c being scheduled during the first output scheduling phase of time slot $t+1$. (9)
- Case b: cell, c,is queued at an input queue, $VOQ_{i,j}$
    - i) Cell c has the lowest *TTL*, hence $OC_{t+1}(c) = 0$.
    - ii) Cell c was inserted with *NNS* (Lemma A.2).
    - iii) Since c has the lowest *TTL*, all cells with lower TTL than c are gone from the system. Thus the internal fabric $XP_{i,j}$ must be available.
- (i) and (ii)imply $IT_{t+1}(c) = 0$, and thus c must be in the front of the IPL. (10)
- (iii) and (10) result in $VOQ_{i,j}$ being eligible and cell c must be transferred to the internal buffer during the first input scheduling phase (11).
- (iii) and (11) result in cell c being chosen by the output scheduler during the first output scheduling phase.

□

In our embedded CICQ switch, the arrival phase and the first scheduling phase can be performed simultaneously to reduce the round trip delay of input scheduling, as in Figure A.3. In this case, the embedded CICQ will emulate an OQ with one time slot delay. The proof remains the same, by always considering only the cells that reach the switch input queues one scheduling phase earlier. The reason for this is because the first scheduling phase excludes currently arriving cells, which are considered starting from the following input scheduling phase onwards.

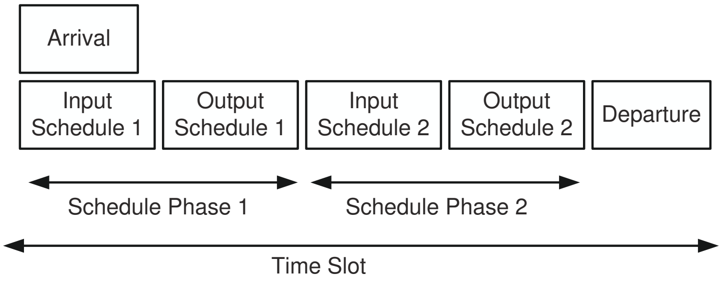


Figure A.3: Scheduling phases in embedded CICQ Switch with parallel arrival and input scheduling phases.

Appendix B

Performance Simulation Environment

This appendix describes the simulation environment used to evaluate the performance of the switching systems studied in this dissertation. First, we introduce the software simulation tool, along with a generic switching system and provide some definitions for its inputs and outputs. We, then, illustrate and describe the traffic models employed to run the experiments. Finally, we explain the indices used to evaluate the performance of the scheduling algorithms and the switching architectures studied in this dissertation.

B.1 Simulation Environment

Throughout all the simulation studies in the dissertation, we used the SIM [108] simulator. SIM is a slotted-time simulator written in ANSI C and was designed for simulating fixed-size switching architectures, such as ATM switches. Instead of discrete-time simulation (event-driven), the simulation in SIM progresses on a time-slot basis (ATM cell). Every time-slot consists of three main steps: *(i)* check the arrival of new cells; *(ii)* schedule the transfer of cells from the inputs of the switch to the output of the switch; and *(iii)* schedule the departure of cells from the outputs of the switch. SIM is structured in modules and each module performs a specific task, such as traffic generator, input queues, fabric switch, scheduling algorithm, and output queues.

We have modified SIM in order to incorporate our scheduling algorithms, queueing structures and the different buffered crossbar configurations. We

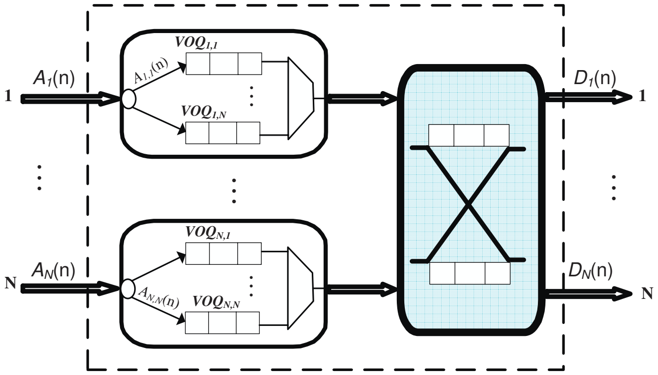


Figure B.1: The dynamics of a generic switch.

have included the queueing structure to support the multicast k input queues studied in Chapter 4. We have included two new modules for input scheduling and output scheduling respectively, used throughout the whole dissertation. We have also included two new modules to support the buffered crossbar switch fabric used throughout the dissertation as well as the partially buffered crossbar switch introduced in Chapter 6.

A generic switch with N inputs and N outputs is depicted in Figure B.1. $\mathcal{A}_{i,j}(n)$ denotes the number of arrivals to input i of cells destined to output j at time-slot n, while $\mathcal{A}_i(n)$ is the aggregate number of arrivals to input i during time-slot n. Every time-slot, at most one cell can arrive at each input. The arrival rate of $\mathcal{A}_{i,j}(n)$ is denoted by $\lambda_{i,j}$. $\mathcal{D}_{i,j}(n)$ denotes the number of departures from output j of cells arriving from input i while $\mathcal{D}_j(n)$ is the aggregate number of departures from output j at time-slot n. Similarly, every time-slot, at most one cell can depart from each output. In order to gather performance results and related statistics, we run SIM for one million time-slots and we gather the data when fourth the simulation time has elapsed. In all the simulations, we consider *admissible* arrival process and i.i.d Bernoulli traffic as defined below.

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

Definition B.2. *Arriving Traffic is said to be 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.*

B.2 Traffic Scenarios

A traffic scenario is generally characterized by two random processes to model the spatial and temporal characteristics. The temporal process refers to the inter-arrival times of successive cell arrivals. This is characterized by the arrival frequency, or input load. The spatial process of traffic is characterized by the distribution of arriving cells over the output destinations. Throughout our simulations, we used both uniform and non-uniform traffic models as described in the next section.

B.2.1 Uniform Traffic

Internet traffic is a mix of all types of traffic. Uniform traffic constitutes a large part of this traffic. The two widely used uniform traffic patterns are Bernoulli uniform and Bursty uniform. We describe each of them below.

Bernoulli Uniform Traffic

This a common test-bed traffic scenario used for evaluating the performance of switch performance. In every time-slot, a cell is generated with probability ρ (also known as the normalized switch input load). Since the traffic is uniform, ρ=λ. The output destination is uniformly distributed over all, N, outputs.

Bursty Uniform Traffic

Bursty traffic is a commonly used traffic model due to its close approximation of Internet traffic. Real network traffic is highly correlated from cell to cell [40] and so in practice, cells tend to arrive in bursts. When data traffic arrives at the input ports of a router, it is often segmented into cells, such as video and/or audio frames, forming a burst or a set of bursts. Therefore, studying the performance of a switching system under bursty traffic is very important. Bursty traffic can be modeled by a two-state Markov-chain consisting of an ON (busy) and an OFF (idle) state. During the ON state, cells (all with the same destination) are generated every time-slot. During an OFF state, no traffic is generated. The busy and idle periods contain a geometrically distributed number of cells, known as the burst size and denoted by b.

B.2.2 Non-Uniform Traffic

Traffic non-uniformity refers to the variation in the distribution of input traffic over the destination output ports. Internet traffic is, generally, non-uniform and asymmetric. Many Internet traffic examples confirm this property, such as client-server applications, where a number of clients communicate with a small number of servers. Since it is nearly impossible to simulate all such workloads, there exist some representative and commonly used non-uniform traffic models. In our simulations, we used two known non-uniform models which we describe next.

Diagonal Traffic

The Diagonal traffic is defined as in the following traffic matrix, for 4×4 switch:

$$\lambda(Diagonal) = \frac{1}{3}\begin{pmatrix} 2\rho & \rho & 0 & 0 \\ 0 & 2\rho & \rho & 0 \\ 0 & 0 & 2\rho & \rho \\ \rho & 0 & 0 & 2\rho \end{pmatrix}$$

This is a very skewed and critical traffic, in the sense that input i has cells only for output i and output $|i+1|$. A diagonal load has $\lambda_{i,i} = \frac{2\rho}{3}$, $\lambda_{i,|i+1|} = \frac{\rho}{3}\ \forall\ i$ and $\lambda_{i,j} = 0$ for all other i and j.

Unbalanced Traffic

The unbalanced traffic is defined by using an unbalanced probability, ω. For a $N \times N$ switch, the traffic load at each input port is defined by ρ. Then, for each input port s and output port d, the traffic load, $\rho_{s,d}$, is given by:

$$\rho_{s,d} = \begin{cases} \rho\left(\omega + \frac{1-\omega}{N}\right) & \text{if } s = d; \\ \rho\,\frac{1-\omega}{N} & \text{otherwise}. \end{cases}$$

Note that when $\omega = 0$, the load is uniform over all outputs and when $\omega = 1$, the traffic is totally unbalanced (only the diagonal). In our experiments, we set ω to be 0.5 because this value corresponds to the pattern with the lowest performance (the hardest to schedule).

B.3 Performance Indices

Three common metrics are, generally, used to evaluate the performance of a switching system: the average cell delay, the switch throughput and the input queues occupancies (for buffer sizing and cell loss ratio). We used these metrics to evaluate the studied switching systems. We describe each of these metrics below.

Average Cell Delay

The delay of a cell is the time duration the cell spends inside the switch queues until it reaches its output port. Depending on the switch architecture used (see Figure B.1), the delay of a cell can refer to the time spent inside the input VOQs (in the case of IQ bufferless switches) or the time spent inside the input VOQs as well as the internal buffers (in the case of CICQ buffered switches). In our study, we consider the average delay over all cells. Note that the average cell delay can be referred to as the mean cell delay, or the mean (average) cell latency. The average cell delay is important as it indicates the efficiency of a scheduling algorithm (or of a switching architecture).

Switch Throughput

The switch throughput is defined as the ratio between the output load and the input load of the switch. The maximum throughput is defined as the maximum input load after which the switch becomes unstable. Instability means that the input load is higher than the throughput of the switch, hence queues will keep growing indefinitely. The maximum throughput is also known as the saturation throughput of the switch and indicates the switch capacity. If the saturation throughput of a switch with a given scheduling algorithm equals to one, which is the maximum value due for a speedup of one, then the given scheduling algorithm is said to achieve 100% throughput. Given two scheduling algorithms both of which can achieve 100% throughput, the one with the shorter average cell delay is preferable. A scheduling algorithm is considered stable if it provides 100% throughput and it keeps the input buffer size bound in number of cells.

Input Queues Occupancies

The input queues occupancies metric indicates the stability of the switching system both with respect to the scheduling algorithm used and/or the underlying switching fabric topology. The input queues occupancies can also serve as an indication on the input buffer size needed to prevent cell loss. We used the L^2 norm vector representing the input VOQs occupancies [104]. Let $VOQ_{i,j}(n)$ be the number of cells queued in $VOQ_{i,j}$ at time slot n. The L^2 norm[1] vector at time slot n is denoted by $\|L(n)\|$ and defined as follows:

$$\|L(n)\| = \sqrt{\sum_{i=1}^{n}\sum_{j=1}^{n}\big(VOQ_{i,j}(n)\big)^2}$$

As it was shown in [104], the input queues occupancies can serve to prove the stability of the scheduling algorithm. That is if, under a scheduling algorithm X, one can show that $E(\|L(n)\|) < \infty$, then it can be concluded that X is stable. Here, $E(x)$ refers to the expected value of x.

---

[1] Also known as the Euclidean norm.

Bibliography

[1] W. Bux, W. Denzel, T. Engbersen, A. Herkersdorf, and R. Luijten, “Technologies and Building Blocks for Fast Packet Forwarding,” *IEEE Communications Magazine*, vol. 39, no. 01, pp. 70–77, January 2001.

[2] L. Roberts, “Beyond Moore’s Law: Internet Growth Trends,” *IEEE Computer Magazine*, vol. 33, no. 1, pp. 117–119, January 2000.

[3] The Internet Systems Consortium, Inc. (ISC), ”http://www.isc.org”.

[4] L. G. Roberts, “Data by the Packet,” *IEEE Spectrum*, vol. 11, no. 2, pp. 46–51, February 1974.

[5] B. Leiner, V. Cerf, D. Clark, R. Khan, L. Kleinrock, D. Lynch, J. Postel, L. Roberts, and S. Wolff, “The Past and Future History of the Internet,” *Communications of the ACM*, vol. 40, no. 2, pp. 102–108, February 1997.

[6] L. Kleinrock, “Information Flow in Large Communication Nets,” *RLE Quarterly Progress Report*, July 1961.

[7] M. Gerla and L. Kleinrock, “Flow Control: A Comparative Survey,” *IEEE/ACM Transactions On Networking*, vol. COM-28, no. 04, April 1980.

[8] G. Pfisher and A. Norton, “Hot Spot Contention and Combining in Multistage Interconnection Networks,” *IEEE Transactions On Computers*, vol. C-34, no. 08, pp. 934–948, October 1985.

[9] M. Katevenis, “Fast Switching and Fair Control of Congested Flow in Broad-Band Networks,” *IEEE Journal in Selected Areas in Communications*, vol. 05, no. 08, pp. 1315–1326, October 1987.

[10] U. Black, *ATM: Foundation for Broadband Networks*. Prentice Hall, 1995.

[11] R. Y. Awdeh and H. T. Mouftah, “Survey of ATM Switch Architectures,” *Computer Networks and ISDN Systems*, vol. 27, no. 12, pp. 1567–1613, September 1995.

[12] D. Clark, “The Design Philosophy of the Darpa Internet Protocols,” *ACM SIGCOMM*, pp. 106–114, August 1988.

[13] S. Keshav and R. Sharma, “Issues and Trends in Router Design,” *IEEE Communications Magazine*, vol. 36, no. 05, pp. 144–151, May 1998.

[14] A. Tanenbaum, *Computer Networks*, Third Edition ed. Prentice Hall, 1996.

[15] L. Roberts, “Packet Switching or Optical Switching?” *IEEE Internet Computing*, vol. 04, no. 01, January/February 2000.

[16] P. Gupta, “Algorithms for Routing Lookups and Packet Classifcation,” Ph.D. dissertation, Stanford University, 2000.

[17] D. Shah, S. Iyer, B. Prabhakar, and N. McKeown, “Maintaining Statistics Counters in Router Line Cards,” *IEEE Micro*, pp. 76–81, January/February 2002.

[18] N. McKeown, “A Fast Switched Backplane For A Gigabit Switched Router,” *Business Communications Review*, vol. 27, no. 12, 1997.

[19] J. Hui, “Switching and Traffic Theory for Integrated Broadband Networks,” *Kluwer Academic Publishers*, January 1990.

[20] A. Pattavina, “Switching Theory: Architectures and Performance in Broadband ATM Networks,” *Wiley, John & Sons*, December 1997.

[21] C. Minkenberg, “On Packet Switch Design,” Ph.D. dissertation, Technische Universiteit Eindhoven, September 2001.

[22] X. Li, L. Mhamdi, J. Liu, K. Pun, and M. Hamdi, *Architectures of Internet Switches and Routers*. Springer-Verlag, September 2006.

[23] J. H. Patel, “Performance of Processor-Memory Interconnections for Multiprocessors,” *IEEE Transactions on Computers*, vol. 30, no. 10, pp. 771–780, 1981.

[24] C. Clos, “A Study of Nonblocking Switching Networks,” *Bell Systems Technical Journal*, vol. 32, p. 406–424, 1953.

[25] V. Benes, “Optimal Rearrangeable Multistage Connecting Networks,” *Bell Systems Technical Journal*, vol. 43, pp. 1641–1656, 1964.

[26] S. Iyer and N. McKeown, “Analysis of the Parallel Packet Switch Architecture,” *IEEE/ACM Transactions on Networking*, vol. 11, no. 2, pp. 314–324, 2003.

[27] N. Chrysos and M. Katevenis, “Scheduling in Non-Blocking Buffered Three-Stage Switching Fabrics,” *IEEE Infocom*, April 2006.

[28] T. Aramaki, H. Suzuki, S. Hayano, and T. Takeuchi, “Parallel ’ATOM’ Switch Architecture For High Speed ATM Networks,” *IEEE International Conference on Communications (ICC)*, pp. 250–254, 1992.

[29] M. Devault, J. Y. Cochennec, and M. Servel, “The Prelude ATD Experiment: Assessments and Future Prospects,” *IEEE Journal on Selected Areas in Communications*, vol. 06, no. 09, pp. 1528–1537, December 1998.

[30] Cisco Systems, “Cisco 12000 Gigabit Switch Router.”

[31] Lucent Technologies, “Performance Optimized Ethernet Switching,” *Cajun White Paper*, 1997.

[32] K. Yoshigoe, “Design and Evaluation of the Combined Input Crossbar Queued (CICQ) Switch,” Ph.D. dissertation, University of South Florida, August 2004.

[33] A. Demers, S. Keshav, and S. Shenker, “Analysis and Simulation of a Fair Queuing Algorithm,” *Internetworking: Research and Experience*, vol. 01, no. 01, pp. 3–26, September 1990.

[34] H. Zhang, “Service Disciplines for Guaranteed Performance Service in Packet-switching Networks,” *Proceedings of the IEEE*, vol. 83, no. 10, pp. 1374–1396, October 1995.

[35] S. Chuang, A. Goel, N. McKeown, and B. Prabhakar, “Matching Output Queueing with a Combined Input Output Queued Switch,” *IEEE Journal on Selected Areas in Communications*, vol. 17, no. 06, pp. 1030–1039, December 1999.

[36] M. Karol, M. Hluchyj, and S. Morgan, "Input Versus Output Queuing on a Space-Division Packet Switch," *IEEE Transactions on Communications*, vol. 35, no. 09, pp. 1337–1356, December 1987.

[37] S. Q. Li, "Performance of a Non-blocking Space-division Packet Switch with Correlated Input Traffic," *IEEE Globecom*, pp. 1754 – 1763, 1989.

[38] C. Y. Chang, A. J. Paulraj, and T. Kailath, "A Broadband Packet Switch Architecture with Input and Output Queuing," *IEEE Globecom*, 1994.

[39] T. Anderson, S. Owicki, J. Saxe, and C. Thacker, "High Speed Switch Scheduling for Local Area Networks," *ACM Transactions on Computer Systems*, pp. 319–352, 1993.

[40] N. McKeown, "Scheduling Algorithms for Input-Queued Cell Switches," Ph.D. dissertation, University of California at Berkeley, May 1995.

[41] H. C. Chi and Y. Tamir, "Symmetric Crossbar Arbiters for VLSI Communication Switches," *IEEE Transactions on Parallel and Distributed Systems*, vol. 04, no. 01, pp. 13–27, January 1993.

[42] N. McKeown, B. Prabhakar, and M. Zhu, "Matching Output Queueing with Combined Input and Output Queueing," *35th Annual Allerton Conference on Communications, Control and Computing*, October 1997.

[43] P. Krishna, N. S. Patel, A. Charny, and R. J. Simcoe, "On the Speedup Required for Work-Conserving Crossbar Switches," *IEEE Journal on Selected Areas in Communications*, vol. 17, no. 06, pp. 1528–1537, June 1999.

[44] B. Prabhakar and N. McKeown, "On the Speedup Required for Combined Input and Output Queued Switching," *Automatica*, vol. 35, no. 12, pp. 1909–1920, September 1999.

[45] A. Mekkittikul, "Scheduling Non-Uniform Traffic In High Speed Packet Switches and Routers," Ph.D. dissertation, Stanford University, November 1998.

[46] M. A. Marsan, A. Bianco, E. Leonardi, and L. Milia, "RPA: A Flexible Scheduling Algorithm for Input Buffered Switches," *IEEE Transactions on Communications*, vol. 47, no. 06, p. 1921–1933, May 1999.

[47] P. Giaccone, “Queueing and Scheduling Algorithms for High Performance Routers,” Ph.D. dissertation, Politecnico Di Torino, February 2002.

[48] R. E. Tarjan, “Data Structures and Network Algorithms,” *Society for Industrial & Applied Mathematics*, 1983.

[49] J. E. Hopcroft and R. Karp, “An $N^{5/2}$ Algorithm For Maximum Matching In Bipartite Graphs,” *Society for Industrial & Applied Mathematics Journal on Computing*, vol. 02, p. 225–231, 1973.

[50] H. C. Chi and Y. Tamir, “Starvation Prevention For Arbiters Of Crossbars With Multi-Queue Input Buffers,” *IEEE International Conference on Communications (ICC)*, pp. 1646–1650, June 1992.

[51] D. Serpanos and P. I. Antoniadis, “FIRM: A Class of Distributed Scheduling Algorithms for High-Speed ATM Switches with Input Queues,” *IEEE Infocom*, 2000.

[52] A. Mekkittikul and N. McKeown, “A Starvation-Free Algorithm For Achieving 100% Throughput in an Input-Queued Switch,” *International Conference on Computer Communications and Networks (ICCCN)*, pp. 226–231, October 1996.

[53] N. McKeown, “iSLIP Scheduling Algorithm for Input-Queued Switches,” *IEEE/ACM Transactions On Networking*, vol. 07, no. 02, pp. 188–201, April 1999.

[54] Y. Jiang and M. Hamdi, “A Fully Desyncronized Round-Robin Matching Scheduler For A VOQ Packet Switch Architecture,” *IEEE Workshop on High Performance Switching and Routing*, pp. 407–411, 2001.

[55] R. Bakka and M. Dieudonne, “Switching Circuits for Digital Packet Switching Network,” *United States Patent 4,314,367*, February 1982.

[56] S. Nojima, E. Tsutsui, H. Fukuda, and M. Hashimmoto, “Integrated Packet Network Using Bus Matrix,” *IEEE Transactions on Communications*, vol. 05, no. 08, pp. 1284–1291, October 1987.

[57] A. K. Gupta, L. O. Barbosa, and N. D. Gorganas, “16x16 Limited Intermediate Buffer Switch Module for ATM Networks for B-ISDN,” *IEEE Globecom*, pp. 939–943, December 1991.

[58] ——, “Limited Intermediate Buffer Switch Modules and Their Interconnection for B-ISDN,” *IEEE International Conference on Communications (ICC)*, pp. 1646–1650, June 1992.

[59] M. Lin and N. Mckeown, “The Throughput of A Buffered Crossbar Switch,” *IEEE Communications Letters*, vol. 9, no. 5, pp. 465–467, May 2005.

[60] M. Nabeshima, “Performance Evaluation of Combined Input-and Crosspoint-Queued Switch,” *IEICE Transactions On Communications*, vol. B83-B, no. 3, March. 2000.

[61] M. Katevenis, G. Passas, D. Simos, I. Papaefstathiou, and N. Chrysos, “Variable Packet Size Buffered Crossbar (CICQ) Switches,” *IEEE International Conference on Communications (ICC 2004)*, vol. 02, pp. 1090–1096, June 2004.

[62] K. Yoshigoe, A. Jacob, and K. J. Christensen, “The RR/RR CICQ Switch: Hardware Design for 10-Gbps Link Speed,” *IEEE International Performance, Computing, and Communications Conference*, pp. 481–485, April 2003.

[63] F. Abel, C. Minkenberg, P. Luijten, M. Gusat, and I. Iliadis, “A Four-Terabit Packet Switch Supporting Long Round-Trip Times,” *IEEE Micro*, vol. 23, no. 1, pp. 10–24, January/February 2003.

[64] R. R. Cessa, “Design and Analysis of Reliable High-Performance Packet Switches,” Ph.D. dissertation, Plytechnic University, April 2001.

[65] S. T. Chuang, “Providing Performance Guarantees with Crossbar-Based Routers,” Ph.D. dissertation, Stanford University, December 2004.

[66] T. Javadi, R. Magill, and T. Hrabik, “A High-Throughput Algorithm for Buffered Crossbar Switch Fabric,” *IEEE International Conference on Communications (ICC)*, pp. 1581–1591, June 2001.

[67] R. Rojas-Cessa, Z. J. E. Oki, and H. J. Chao, “CIXB-1: Combined Input One-Cell-Crosspoint Buffered Switch,” *IEEE Workshop on High Performance Switching and Routing (HPSR)*, pp. 324–329, 2001.

[68] K. Yoshigoe and K. J. Christensen, “A Parallel-Polled Virtual Output Queued Switch with a Buffered Crossbar,” *IEEE Workshop on High Performance Switching and Routing*, pp. 271–275, 2001.

[69] L. Mhamdi and M. Hamdi, “Practical Scheduling Algorithms for High-Performance Packet Switches,” *IEEE International Conference on Communications (ICC)*, vol. 03, pp. 1659–1663, May 2003.

[70] R. R. Cessa, E. Oki, and H. J. Chao, “On the Combined Input Crosspoint Buffered Packet Switch with Round-Robin Arbitration,” *IEEE Transactions on Communications*, vol. 53, no. 11, p. 1945–1951, November 2005.

[71] B. Magill, C. Rohrs, and R. Stevenson, “Output Queued Switch Emulation by a Buffered Crossbar Fabric,” *IEEE Journal on Selected Areas in Communications*, vol. 17, no. 06, pp. 1030–1039, May. 2003.

[72] S. Chuang, S. Iyer, and N. McKeown, “Practical Algorithms for Performance Guarantees in Buffered Crossbars,” *IEEE Infocom*, March 2005.

[73] J. Turner, “Strong Performance Guarantees for Asynchronous Crossbar Schedulers,” *IEEE Infocom*, pp. 1–11, April 2006.

[74] D. Pan and Y. Yang, “Localized Asynchronous Packet Scheduling For Buffered Crossbar Switches,” *Proceedings of the 2006 ACM/IEEE Symposium on Architecture for Networking and Communications Systems (ANCS 06)*, pp. 153–162, December 2006.

[75] L. Mhamdi, M. Hamdi, C. Kachris, S. Wong, and S. Vassiliadis, “High-Performance Switching Based on Buffered Crossbar Fabrics,” *Computer Networks*, vol. 50, no. 13, pp. 2271–2285, September 2006.

[76] Xilinx Inc., “Virtex-4 Family Overview,” http://www.xilinx.com, March 2005.

[77] T. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction To Algorithms*. The MIT Press, Cambridge, Massachusetts, March 1990.

[78] H. J. Chao, “Next Generation Routers,” *Proceedings of the IEEE*, vol. 90, no. 9, pp. 1518–1558, September 2002.

[79] L. Mhamdi and M. Hamdi, “MCBF: A High-Performance Scheduling Algorithm for Buffered Crossbar Switches,” *IEEE Communications Letters*, vol. 07, no. 09, pp. 451–453, September 2003.

[80] R. Rojas-Cessa, Z. Guo, and N. Ansari, “On the Maximum Throughput of a Combined Input-Crosspoint Buffered Packet Switch,” *IEICE Transactions on Communications*, vol. E89-B, pp. 3120 – 3123, November 2006.

[81] Xilinx Inc., “Virtex-4 RocketIO Multi-Gigabit Transceiver,” http://www.xilinx.com, March 2005.

[82] P. Gupta and N. McKeown, “Design and Implementation of a Fast Crossbar Scheduler,” *IEEE Micro*, vol. 19, no. 01, January/February 1999.

[83] B. Prabhakar, N. McKeown, and R. Ahuja, “Multicast Scheduling for Input-Queued Switches,” *IEEE Journal on Selected Areas in Communoications (JSAC)*, vol. 15, no. 15, pp. 885–866, June 1997.

[84] M. A. Marsan, A. Bianco, P. Giaccone, E. Leonardi, and F. Neri, “Multicast Traffic in Input-Queued Switches: Optimal Scheduling and Maximum Throughput,” *IEEE/ACM Transactions on Networking*, vol. 03, no. 11, pp. 465–477, June 2003.

[85] M. Guo and R. Chang, “Multicast ATM Switches: Survey and Performance Evaluation,” *Computer Communication Review*, vol. 28, no. 02, pp. 98–131, April 1998.

[86] M. A. Marsan, A. Bianco, P. Giaccone, E. Leonardi, and F. Neri, “Optimal Multicast Scheduling in Input-Queued Switches,” *IEEE International Conference on Communications (ICC)*, 2001.

[87] S. Gupta and A. Aziz, “Multicast Scheduling for Switches with Multiple Input-Queues,” *Proceedings of Hot Interconnects*, pp. 28–33, 2002.

[88] B. Prabhakar and N. McKeown, “Designing a Multicast Switch Scheduler,” *Proceedings of the 33rd Annual Allerton Conference on Communication, Control, and Computing*, pp. 984–993, Oct. 1995.

[89] J. Y. Hui and T. Renner, “Queuing Analysis for Multicast Packet Switching,” *IEEE Transactions on Communications*, vol. 42, no. 02, pp. 723–731, February 1994.

[90] A. Bianco, P. Giaccone, E. Leonardi, F. Neri, and C. Piglione, “On the Number of Input Queues to Efficiently Support Multicast Traffic in Input Queued Switches,” *IEEE Workshop on High Performance Switching and Routing (HPSR)*, pp. 111–116, June 2003.

[91] M. Song and W. Zhu, “Throughput Analysis for Multicast Switches with Multiple Input Queues,” *IEEE Communications Letters*, vol. 08, pp. 479—-481, July 2004.

[92] P. Giaccone and E. Leonardi, “Asymptotic Performance Limits of Switches with Buffered Crossbars Supporting Multicast Traffic,” *IEEE Infocom*, pp. 1–10, April 2006.

[93] S. Sun, S. He, Y. Zheng, and W. Gao, “Multicast Scheduling in Buffered Crossbar Switches with Multiple Input Queues,” *IEEE Workshop on High Performance Switching and Routing (HPSR)*, pp. 73–77, May 2005.

[94] J. D. C. Little, “A Proof of the Queueing Formula $L = \lambda W$,” *Operations Research*, vol. 9, pp. 383–387, 1961.

[95] M. Andrews, S. Khanna, and K. Kumaran, “Integrated Scheduling of Unicast and Multicast Traffic in an Input-Queued Switch,” *IEEE Infocom*, pp. 1144–1151, 1999.

[96] M. Song and W. Zhu, “Integrated Queuing and Scheduling for Unicast and Multicast Traffic in Input-Queued Packet Switches,” *IASTED International Conference on Communication and Computer Networks (CCN 2004)*, November 2004.

[97] L. Mhamdi and M. Hamdi, “Scheduling Multicast Traffic in Internally Buffered Crossbar Switches,” *IEEE International Conference on Communications (ICC)*, pp. 1103–1107, June 2004.

[98] C. Minkenberg, “Integrating Unicast and Multicast Traffic Scheduling in A Combined Input- and Output-Queued Packet-Switching System,” *International Conference on Computer Communications and Networks (ICCCN)*, pp. 127–234, 2000.

[99] E. Schiattarella and C. Minkenberg, “Fair Integrated Scheduling of Unicast and Multicast Traffic in an Input-Queued Switch,” *IEEE International Conference on Communications (ICC)*, vol. 01, pp. 287–292, June 2006.

[100] B. Prabhakar, N. McKeown, and R. Ahuja, “Multicast Scheduling for Input-Queued Switches,” *IEEE Journal in Selected Areas in Communications (JSAC)*, pp. 2021–2027, June 1997.

[101] K. Yoshigoe, K. Christensen, and A. Jacob, “The RR/RR CICQ Switch: Hardware Design for 10-Gbps Link Speed,” *IEEE International Performance, Computing, and Communications Conference*, pp. 481–485, April 2003.

[102] N. McKeown, M. Izzard, A. Mekkittikul, B. Ellersick, and M. Horowitz, “The Tiny Tera: A Packet Switch Core,” *IEEE Micro*, pp. 26–33, January/February 1997.

[103] C. Minkenberg and T. Engbersen, “A Combined Input And Output Queued Packet-Switched System Based On A Prizma Switch-On-A-Chip Technology,” *IEEE Communications Magazine*, vol. 38, no. 2, pp. 70–77, December 2000.

[104] N. McKeown, A. Mekkittikul, V. Anantharam, and J. Walrand, “Achieving 100% Throughput in Input-Queued Switch,” *IEEE Transastions On Communications*, vol. 47, no. 08, 1999.

[105] M. Katevenis, G. Passas, D. Simos, I. Papaefstathiou, and N. Chrysos, “Variable Packet Size Buffered Crossbar (CICQ) Switches,” *IEEE International Conference on Communications (ICC 2004)*, vol. 2, pp. 1090–1096, June 2004.

[106] N. Chrysos and M. Katevenis, “Scheduling in Switches with Small Internal Buffers,” *IEEE Globecom*, pp. 614–619, November 2005.

[107] R. Rojas-Cessa and Z. Dong, “Combined Input-Crosspoint Buffered Packet Switch with Flexible Access to Crosspoint Buffers,” *IEEE International Caribbean Conference on Devices, Circuits and Systems, Playa del Carmen*, April 2006.

[108] “SIM,” High-Performance Networking Group, Stanford University http://klamath.stanford.edu/tools/SIM/.

List of Publications

Book Chapters

1. X. Li, L. Mhamdi, J. Liu, K. Pun, and M. Hamdi. **Architectures of Internet Switches and Routers**. *Springer-Verlag*, pp. 1–38, ISBN: 978-1-84628-273-7, September 2006.

International Journals

1. L. Mhamdi, G. N. Gaydadjiev, and S. Vassiliadis. **Efficient Multicast Support in High-Speed Packet Switches**. *Journal of Networks*, Vol. 2 No. 3, pp. 28–35, June 2007
2. L. Mhamdi, M. Hamdi, C. Kachris, S. Wong and S. Vassiliadis. **High-Performance Switching Based on Buffered Crossbar Switches**. *Computer Networks*, 50(13): 2271–2285, September 2006.
3. L. Mhamdi and M. Hamdi. **MCBF: A High-Performance Scheduling Algorithm for Buffered Crossbar Switches**. *IEEE Communications Letters*, 07(09): 451–453, September 2003.

Conference Proceedings

1. L. Mhamdi and S. Vassiliadis. **Hybrid Scheduling in High-Speed Packet Switches**. *IEEE International Conference on Signal Processing and Communications (ICSPC 2007)*, To appear.
2. L. Mhamdi and S. Vassiliadis. **Integrating Uni- and Multicast Scheduling in Buffered Crossbar Switches**. *IEEE Workshop on High Performance Switching and Routing (HPSR 2006)*, pages 99–104, June 2006.

3. L. Mhamdi, C. Kachris and S. Vassiliadis. **A Reconfigurable Hardware Based Embedded Scheduler for Buffered Crossbar Switches**. *ACM/SIGDA Fourteenth International Symposium on Field Programmable Gate Arrays (FPGA 2006)*, pages 143–149, February 2006.
4. L. Mhamdi and S. Vassiliadis. **Multicast Traffic Scheduling Based On High-Speed Crossbar Switches**. *Proc. of the 17th Annual Workshop on Circuits, Systems and Signal Processing, (ProRisc 2006)*, pages 313–318, November 2006.
5. L. Mhamdi and S. Vassiliadis. **A Practical Scheduler for High-Speed Switches and Internet Routers**. *Proc. of the 16th Annual Workshop on Circuits, Systems and Signal Processing, (ProRisc 2005*, pages 398–403, November 2005.
6. L. Mhamdi and M. Hamdi. **Scheduling Multicast Traffic in Internally Buffered Crossbar Switches**. *IEEE International Conference on Communications (ICC 2004)*, pages 1103–1107, June 2004.
7. L. Mhamdi and M. Hamdi. **Output Queued Switch Emulation By a One-Cell-Internally Buffered Crossbar Switch**. *IEEE Global Telecommunications Conference, (GLOBECOM 2003)*, pages 67–72, June 2003.
8. L. Mhamdi and M. Hamdi. **CBF: A High-Performance Scheduling Algorithm for Buffered Crossbar Switches**. *IEEE Workshop on High Performance Switching and Routing, (HPSR 2003)*, pages 3688–3693, December 2003.
9. L. Mhamdi and M. Hamdi. **Practical Scheduling Algorithms for High-Performance Packet Switches**. *IEEE International Conference on Communications, (ICC 03)*, pages 1659–1663, May 2003.

Samenvatting

Verscheidene voorstellen om geschikte architecturen voor hoge snelheid *packet switches* (*IP routers* en *ATM switches*) te vinden zijn door academici en industrie onderzocht en geïmplementeerd. De onderverdeling van deze architecturen kan gebeuren op basis van verschillende attributen, zoals wachtrijmethoden, arbitrage algoritmen en/of de interne topologie. De meeste hoge snelheid switches en Internet routers van vandaag de dag gebruiken een *crossbar fabric* zonder buffers op de kruispunten. Het ontwerpen van op crossbars gebaseerde routers die schaalbaar zijn en gegarandeerde prestatie leveren, is moeilijk met de huidige technologie. Dit wordt toegeschreven aan de hoge complexiteit van berekeningen in de centrale crossbar arbiter en aan de natuur van de op crossbars gebaseerde switchingarchitectuur zelf.

Dit proefschrift bestudeert het arbitrage probleem in crossbar switches met buffers, waar de kruispunten een kleine buffer hebben. De arbitrage van *unicast*- en *multicast*verkeersstromen en tevens de integratie van beiden worden behandeld. Een aantal gedistribueerde en parallelle arbitragealgoritmen met bijbehorende architecturen worden beschreven. Deze algoritmen zijn ontworpen om praktisch implementeerbaar te zijn en om schaalbaar te zijn met het aantal poorten van een router en met de lijnsnelheid.

Een klasse van unicastarbitragealgoritmen wordt beschreven, die alleen de status van de interne buffers gebruiken. Een switchingarchitectuur wordt voorgesteld, waar alle arbiters in de crossbar chip geïntegreerd zijn. Verder wordt beschreven hoe de voorgestelde architectuur gagarandeerde prestaties kan leveren. Met een zogenaamde versnelling van een factor twee is de voorgestelde architectuur in staat een ideale switch met wachtrijen op de uitgangen te emuleren.

Het probleem van arbitrage van multicastverkeersstromen wordt ook bestudeerd. Een gebufferde crossbararchitectuur gebaseerd op ingangs-multicast eerst-komt-eerst-maalt rijen met de bijbehorende arbitrage wordt beschreven. Deze architectuur presteert beter dan bestaande architecturen. De architectuur voor de multicast switch wordt verder verbeterd door een klein aantal buffers per ingang te gebruiken. Er wordt een algoritme voor cel plaatsing bepaald dat inkomend verkeer aan ingangsbuffers toewijst. Er wordt aangetoond dat dit algoritme verkeer efficiënter, eerlijker en sneller kan toewijzen dan bestaande algoritmes. Deze studie laat een interessante afweging zien tussen het aantal ingangsbuffers voor multicast en de grootte van de interne buffers. Dit zorgt zowel voor verdere prestatieverbetering van de switch als voor een vermindering in de arbitrage complexiteit, met als gevolg een snellere en beter schaalbare switcharchitectuur.

Voorts wordt de arbitrage van meer realistische verkeersstromen bestudeerd: de combinatie van unicast en multicast. De architectuur van een op crossbars gebaseerde switch wordt beschreven, vergezeld van bijbehorende arbitrage die unicast en multicast efficiënt ondersteunt. Hoewel de voorgestelde arbiter op *fanout splitting* is gebaseerd, nijgt deze multicast verkeer te kunnen behandelen zonder de verbinding tussen de lijnkaarten en de kern van de switch te overbelasten. Er wordt aangetoond dat deze architectuur betere prestaties levert dan bestaande architecturen.

Als laatste wordt een variant op de architectuur voor de gebufferde crossbar switch bestudeerd. Er wordt een gedeeltelijk gebufferde crossbararchitectuur voorgesteld. Deze is ontworpen om een goed compromis te zijn tussen de twee uitersten van bufferloze crossbars en volledig gebufferde crossbars. De gedeeltelijke gebufferde crossbar is gebaseerd op een paar interne buffers per crossbar uitgang, waardoor de kosten vergelijkbaar zijn met een architectuur zonder buffers. Het knelpunt van de gecentraliseerde crossbar arbitrage wordt verholpen door deze arbitrage gedistribueerd en *pipelined* te doen zoals ook wordt gedaan in volledig gebufferde crossbars. Hierdoor wordt de architectuur goedkoop en praktisch voor gebruik in netwerken met extreem hoge capaciteit.

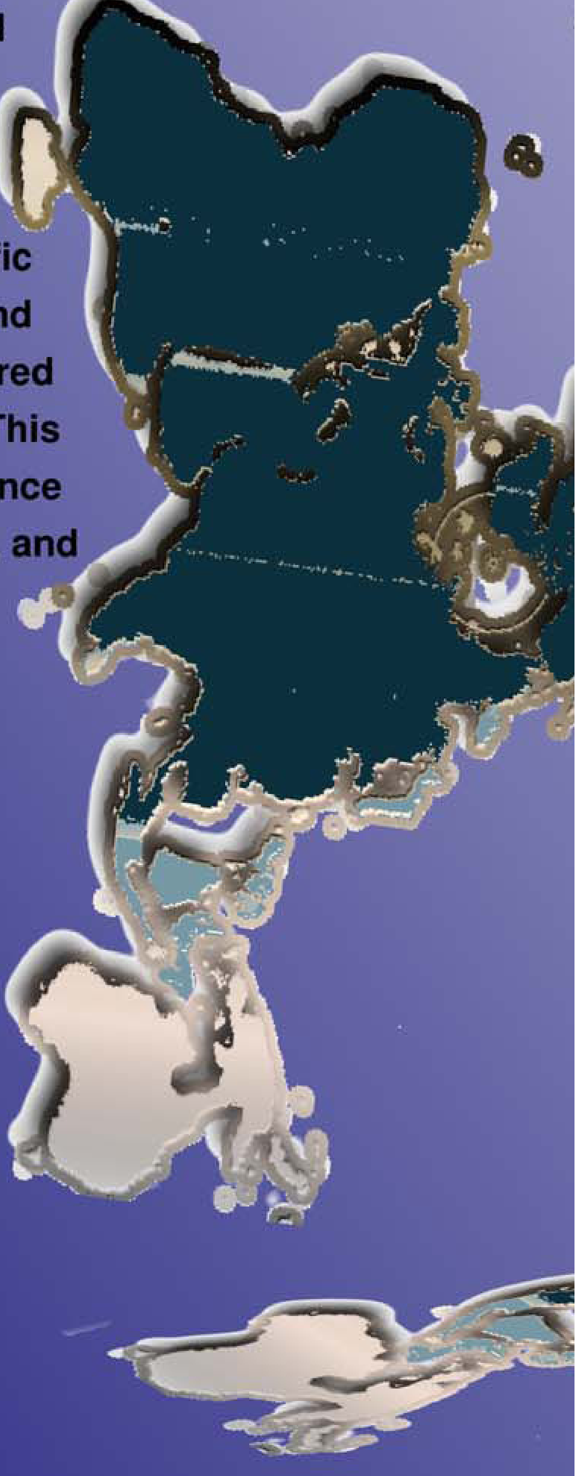
Curriculum Vitae

Lotfi Mhamdi was born on the 16th of November 1975 in Sidi Bouzid, Tunisia. He received the Bachelor of Science (BS) degree from South University, Tunisia, in 2000. In the same year, he was admitted to the Computer Science department at the Hong Kong University of Science and Technology (HKUST), Hong Kong, as a postgraduate student. In November 2002, he obtained the Master of Philosophy (MPhil) degree in computer science from HKUST. From 2001 until 2003, he has been a teaching assistant for various bachelor level computer science courses at HKUST. During the same period, he has also been a research assistant within the Area Of Excellence in Information Technology (AOE-IT) project, Hong Kong.

In 2004, Mr. Mhamdi joined the Computer Engineering (CE) Laboratory of Delft University of Technology (TU Delft), The Netherlands, as a researcher. He has been working towards the PhD degree under the guidance of Prof. dr. Stamatis Vassiliadis. His research work spans the area of high-speed networks including the design, analysis, scheduling and management of high-speed switches and Internet routers.

Mr. Mhamdi is a reviewer for most of the major IEEE ComSoc as well as various Computer Architecture conferences and journals. He served as the publicity chair of the International Conference on Design and Technology of Integrated Systems in nanoscale era (DTIS 2007). He is a technical program committee member of the IEEE International Workshop on High Performance Switching and Routing (HPSR 2008), the IEEE International Conference on Communications (ICC 2008) and the ACM/IEEE International Symposium on Networks-on-Chip (NoCS 2008). He is a member of IEEE.

High performance routers are the basic building blocks of the Internet. Most high performance routers built today use crossbars and a centralized scheduler. Due to their high scheduling complexity, crossbar-based routers are not scalable and cannot keep pace with the explosive growth of the Internet. This dissertation studies a slight variation to the crossbar, a buffered crossbar switching architecture. A scalable buffered crossbar switch design is proposed, using embedded unicast scheduling. Furthermore, an efficient buffered crossbar-based switching architecture is described for multicast traffic support as well as the integration of unicast and multicast traffic flows. Finally, a partially buffered crossbar switching architecture is proposed. This architecture is shown to exhibit high performance comparable to that of fully buffered crossbars, and a low cost, comparable to that of unbuffered crossbars.



TU Delft
Technische Universiteit Delft

ISBN: 978-90-807957-2-3