RZ 3387 (# 93433) 12/17/01
Electrical Engineering 200 pages

Research Report

On Packet Switch Design

Cyriel Minkenberg

IBM Research
Zurich Research Laboratory
8803 Rüschlikon
Switzerland

On Packet Switch Design

Cyriel Minkenberg

IBM Research, Zurich Research Laboratory, 8803 Rüschlikon, Switzerland

Abstract

Growth in communication network capacity has been fueled by rapid advances in fiber-optic transmission technologies. In contrast, growth in the capacity of switching and routing nodes has grown at a much slower pace, to the point where they are currently the bottlenecks that limit network capacity. The maximum capacity (throughput) a packet switch can achieve is largely determined by its architecture. It is widely recognized that conventional architectures cannot be scaled to implement the multi-terabit/s fabrics that will soon be required. Therefore, the problem this dissertation addresses is that of scalable high-capacity packet switches with a strong focus on performance.

First, a comprehensive overview of current packet-switch architectures is given, in which the pros and cons of existing approaches are identified. The main contribution of this dissertation is the introduction and evaluation of a novel architecture that combines the strengths of input-queued switches using virtual output queuing (VOQ) and shared-memory output-queued switches. When compared to purely input-queued switches or combined switches with a limited speed-up, the need for centralized arbitration is removed, whereas when compared to traditional purely output-buffered switches the proposed architecture requires only a relatively small amount of costly output buffers. Performance of the proposed architecture is shown to be high and robust under a wide variety of traffic patterns, particularly when compared with existing architectures. A detailed study on the implementation aspects of the proposed architecture demonstrates its feasibility.

A VOQ-based multicast approach that integrates unicast and multicast queuing and scheduling disciplines is introduced, and is shown to outperforms the existing dedicated-multicast-queue approaches by far, and a straightforward, inexpensive implementation is described.

Finally, a method to support long packets is introduced. Because these long packets have to share resources in the switch (the output queues and the shared memory), deadlocks can occur if no precautions are taken. We demonstrate how a suitable modification of the flow-control mechanism can eliminate all possible deadlocks in the presence of both unicast and multicast traffic, as well as with multiple traffic priorities. To assess its complexity, a possible implementation is described in detail.

This dissertation* has been performed in the context of the PRIZMA project (the IBM family of high-speed packet-switch chips). Its main role within this project has been system-level modeling and simulation. The main contributions in this context are that it has greatly increased understanding of the proposed system in terms of system-level architecture and performance, and, as a direct result of this increased insight, has paved the way for further architectural improvements.

*PhD Thesis, Eindhoven University of Technology, The Netherlands, 2001.

Contents

Abbreviations

ATM	Asynchronous Transfer Mode
CIOQ	Combined Input- and Output-Queued
CMOS	Complementary Metal-Oxide Semiconductor
CNN	Cellular Neural Networks
CPI	Clustered Processor Interconnect
DD	Delay Distribution
DEC	Digital Equipment Corporation
DWDM	Dense Wavelength-Division Multiplexing
FARR	Fair Arbitrated Round Robin
FCFS	First-Come First-Served
FIFO	First-In First-Out
FIRM	FCFS In Round-robin Matching
Gbps	Gigabits per second
GBps	Gigabytes per second
GTI	Gran Turismo Injection
HoL	Head-of-Line
IBM	International Business Machines
i.i.d.	independently and identically distributed
IP	Internet Protocol
IQ	Input Queue/Queued/Queuing
IQD	Input Queue size Distribution
iSLIP	Iterative SLIP (see also SLIP)
LCFS	Last-Come-First-Served
LOOFA	Lowest Output queue Occupancy First Algorithm
LRU	Least Recently Used
MEMS	Micro-Electro-Mechanical System
MFT	Memory Full Threshold
NHD	Networking Hardware Division
OCF	Oldest Cell First
OQ	Output Queue/Queued/Queuing
OQT	Output Queue Threshold
PIM	Parallel Iterative Matching
PPS	Parallel Packet Switch
SLIP	(Not an abbreviation, “slip” refers to the desynchronization of grant arbiters that is characteristic of the algorithm)
QoS	Quality of Service
VOQ	Virtual Output Queue(d/ing)
WBA	Weight-Based Algorithm
WDM	Wavelength-Division Multiplexing
WFQ	Weighted Fair Queuing
RAR	Replication At Receiving
RAS	Replication At Sending
RR	Round Robin
RRM	Round-Robin Matching
RPA	Reservation with Preemption and Acknowledgment
PRIZMA[1]	Parallel Routing In Zurich’s Modular Architecture; Packetized Routing Integrated Zurich Modular Architecture; Packetized Routing In Zurich’s Modular Approach

[1] A concensus on the precise meaning of this acronym has not yet been reached.

Chapter 1

Introduction

This chapter starts with the motivation of the work presented in this dissertation, and proceeds to introduce the main topic and detail the design task. Finally, the organization of the dissertation is explained.

1.1 Motivation

The need to communicate

Throughout history, humans have always had a need to communicate. Because of the central role communication plays in human society, inventions that impact communications also tend to have a drastic impact on society as a whole. Next to the invention of the printed word (Johannes Gutenberg, 1446), the telegraph (Samuel Morse, 1835) and the telephone (Alexander Bell, 1876) represent the greatest revolutions in communications. The next revolution, brought about by the merging of digital computer and communications technologies, has opened up an entire new dimension of communication, with the Internet, and in particular the bandwidth-hogging World Wide Web, as one of its most prominent exponents. With the arrival of the Information Age as successor to the Industrial Age, demand for communication bandwidth has been growing explosively, best exemplified by the exponential growth curves of the Internet. More importantly, this trend shows no sign whatsoever of slowing down. On the contrary, numerous new applications are being proposed that will drive demand for communication bandwidth at an even faster pace. The widespread deployment of applications such as audio- and video-streaming, the conduction of business-to-consumer and business-to-business transactions, and the integration of wireless communications with the Internet are expected to be a driving force behind this trend.

Structure of communication networks

The infrastructure that has to support this communication is faced with rapidly increasing demand, which is the main motivation behind this work. Before going deeper into details, let us have a closer look at the general structure of communication networks. From the end-user's viewpoint, it would be ideal to have a dedicated connection between any pair of end-users in a network, because this offers known, deterministic latency, throughput and availability characteristics. A pair of communicating users can never interfere with communication between any other pair. While these are certainly highly desirable features, the price to be paid is also very high: the complexity in terms interconnection links in such network with N users is on the order of N^2, which may be feasible for a network having just a few users, but for networks such as the public telephone system or the Internet with tens of millions or even hundreds of millions of users, such a solution is clearly both impractical and prohibitively expensive.

To overcome this interconnection complexity hurdle, we make the following assumptions:

- End-user bandwidth demand varies strongly over time.
- Given a sufficiently large group of M end users, the probability that the aggregated bandwidth demand from this group exceeds a certain threshold T can be made "acceptably" small (in circuit-switching this is called the *blocking* probability).

Now, in general the sum of the requested user bandwidth is much smaller than the total bandwidth available to the group, which offers the potential for substantial cost savings. To take advantage of this, traffic from this group of end users is aggregated *(multiplexed)* onto a single link of bandwidth T that will be shared by all of the users in the group.

Statistical multiplexing

This principle is called *statistical multiplexing* and is one of the most important design principles of any communication network, because it constitutes the key enabling factor for cost-effective deployment of ubiquitous high-speed communication networks.

Users with bandwidth demands that do not satisfy the above assumptions may be better off not sharing their bandwidth, which is why telecommunication operators also offer leased lines.

Applying the principle of statistical multiplexing boils down to segmenting the network. This is done by first assigning groups of users that will share a link, and then interconnecting the groups by means of *switching nodes* or *switches* for short. The main task of these nodes is to forward information arriving on their input links to the correct output links. In early telephony exchanges this was accomplished by means of electro-mechanical relay switches, hence the term "switching", which is still in use today. In the context of packet-switched networks, the nodes are typically called *routers*. The interconnecting transmission links are often called *trunks*. In large networks, there may be multiple hierarchical stages of segmentation, so that switches in a lower stage share links through the higher stages to achieve additional efficiency benefits from statistical multiplexing, see Fig. 1.1.

The great bandwidth gap: Optics vs. electronics

As described above, the main two components of a communication network are the transmission links ("transport") and the switching nodes. The transmission part of current high-bandwidth networks is largely fiber-optic. Advances in fiber-optic transmission technologies such as wavelength-division multiplexing (WDM) and dense WDM (DWDM) have greatly pushed the envelope of bandwidth available in fibers by multiplexing large numbers of separate channels onto a single fiber. Each individual channel typically operates at the Optical

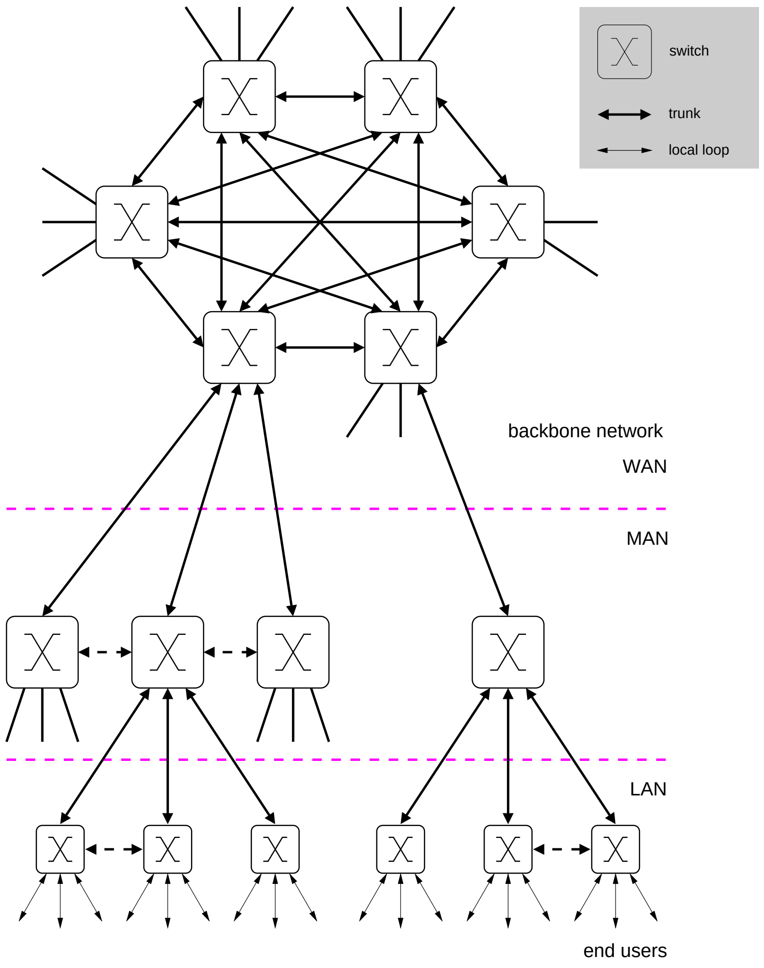


Figure 1.1: Hierarchical network structure. The switches act as statistical multiplexers in and between stages.

Carrier (OC-x) rates OC-48 (2.5 Gb/s), OC-192 (10 Gb/s), or even OC-768 (40 Gb/s). Using state-of-the-art DWDM techniques, a single fiber can carry over 5 terabit of data per second.[1]

Although silicon technologies have also advanced rapidly, the gap between the data rates that optical transmission technology can deliver and those that electronic switches can process is widening at an alarming rate. Worse still, in the foreseeable future certain fundamental physical limits of traditional CMOS silicon technology are expected to be reached. Unlike when bipolar silicon technology ran out of steam and CMOS was already around and mature enough to take over, there is no such emerging technology in sight yet that can replace CMOS within, say, the next five to ten years.[2] This means that the switching and routing nodes of the network are the bottleneck of the network, and will remain so for some time to come. Thus, it is of utmost importance that the switch be as efficient as possible in transferring packets from in- to outputs, to avoid further reduction in the usable link rate, i.e. as little bandwidth as possible should go unused and its throughput must be close to 100%.

[1]Current laboratory DWDM setups [Bigo00] can multiplex as many as 128 colors, each color carrying an OC-768 channel, on one fiber, which adds up to 5.12 Tb/s.

[2]The so-called double-gate transistor technology may provide further leeway.

As an alternative to electronic switches, all-optical switches have the advantages that light can be routed through free space, and that light can be carried over long distances with very little loss in signal power. With transport being almost entirely in the optical domain, such all-optical switches have the additional advantage of eliminating the expensive conversion from the optical to the electrical domain and back. However, they have two distinct disadvantages. First, most current optical switches can only switch fairly slowly (in the 1 to 10 ms range; faster optical switches do exist, but are prohibitively expensive), which prevents per-packet processing. Turner's "burst switching" scheme [Turner98, Turner99, Chen00b] aims at circumventing this problem by switching at the granularity of bursts, i.e., large, aggregated chunks of data that can be routed as a whole. Second, compared to its electronic equivalent, optical storage of information is very cumbersome and impractical.

Still, a number of all-optical switches are being developed, such as micro-electro-mechanical-system (MEMS) switches, thermo-optic switches, "bubble" (waveguide) switches, and liquid-crystal switches [Yao00, Bishop01].

Until optical buffering becomes feasible, optical switching granularity is improved, and optical switch technology becomes cost-competitive with its electronic counterpart, electronic switches will continue to play an important role. Hybrid solutions combining the strengths of the two technologies will certainly also be proposed in the meanwhile. By their respective natures, all-optical switches are better suited to circuit switching and electronic switches to packet switching.

Circuit and packet switching

Conventional circuit-switched networks such as the telephone system are unsuitable for data and computer communication because of the fundamentally different characteristics of the types of traffic. In contrast to the constant rate of voice traffic, data and computer communication traffic is typically very bursty. This makes it inherently unsuitable to be switched by a circuit-switched network, because such a network (a) employs dedicated communication paths, leading to poor utilization, (b) is based on fixed-rate bandwidth allocation, implying inflexibility of coping with widely varying bandwidth demands, and (c) incurs high call setup and teardown overheads, making it unsuitable for short-duration calls (cell-based networks using virtual circuit switching also suffer from this last disadvantage). These realizations led to the development of packet-switched techniques and networks, which overcome the afore-mentioned shortcomings, at the price of (a) complex routing and control, and (b) variable, unpredictable network performance in terms of loss rates, latency, jitter, and throughput. The key difference between switching nodes in a circuit- and a packet-switched network is that in the former, *no queuing* is required in the node because a dedicated path exists and arriving data is guaranteed to be able to leave (almost) immediately.[3] In the latter, however, packets from different sources can simultaneously contend for the same outgoing bandwidth, so that buffering is required to store packets that cannot be transmitted immediately. As we shall see later, the queuing discipline employed in a packet switch is paramount to its performance.

A strong argument for circuit switching is that moving from a packet-switched network to a circuit-switched network obviates the need for buffering, thus, perhaps, enabling fully optical

[3]Some buffering may be required, such as in a time slot interchanger (TSI), but this buffering is strictly limited and deterministic.

switching.

For the finer details of both circuit- and packet-switched networks, the reader is referred to [Stallings92] for example. A useful book on computer networks in general is [Tanenbaum96].

On packets, cells, and frames

Current state-of-the-art semiconductor technologies allow the implementation of packet switching at transmission rates up to several Gb/s per link in VLSI hardware. For implementational complexity reasons, packet switching at the hardware level is usually tailored towards fixed-length packets. Therefore, in this dissertation, the term *packet* will be used for fixed-size data units, analogous to the term *cell*, which originates in the 53-byte data unit employed in Asynchronous Transfer Mode (ATM) networks. Typical packet sizes are in the range of 32 to 256 bytes. One *packet cycle* equals the duration of a single packet $T = L/B$, where L is the length of a packet in bits, and B the link rate in bits per second. When referring to data units of variable size, in the range of, say, 64 to 2,048 bytes, we will employ the term *frame*.

Note that these definitions may differ from those employed elsewhere in the literature.

Switch and adapter functionality

Fig. 1.2 depicts a fictitious network, consisting of four switches (represented by the rounded boxes), a number of trunks interconnecting the switches (represented by the thick two-headed arrows, indicating bi-directional communication), and a collection of various data sources and sinks, including speech-, audio-, video- and data-communication, which are all being transported simultaneously across the same network. Typically, each data source/sink connects to the network through an *adapter*, whose job it is to translate the incoming and outgoing data streams at both the logical and physical levels. This often entails segmenting the incoming data stream, which can take virtually any shape, into (fixed-length) data units (packets) that the network can transport, and then transmitting these packets using the protocols and the physical layer technology that the network the adapter is connected to understands. An adapter at the receiving end will have to reassemble the segments into the original data stream to reconstruct the signal. The job of the switches it is then to route the incoming packets to their proper destination. Both the adapter function and the routing function of the switch are illustrated in Fig. 1.3.

For the time being, we will not concern ourselves with how the entire route through the network is determined and established; this is a science in its own right. One way is to perform source routing, where the route is determined in advance, and store the local routing tags for each switch in the packet itself so that the packet is self-routing throughout the network. An alternative is to compute the route in advance (static routing), but have the local routing tags inserted before each switch, which requires that either switch or inter-switch adapters perform header translations. A third possibility is to do fully dynamic routing. In an environment where packet order must be maintained, this incurs additional complications. These issues are beyond the scope of this work.

Thus, packet-based switches are required to perform two basic functions:

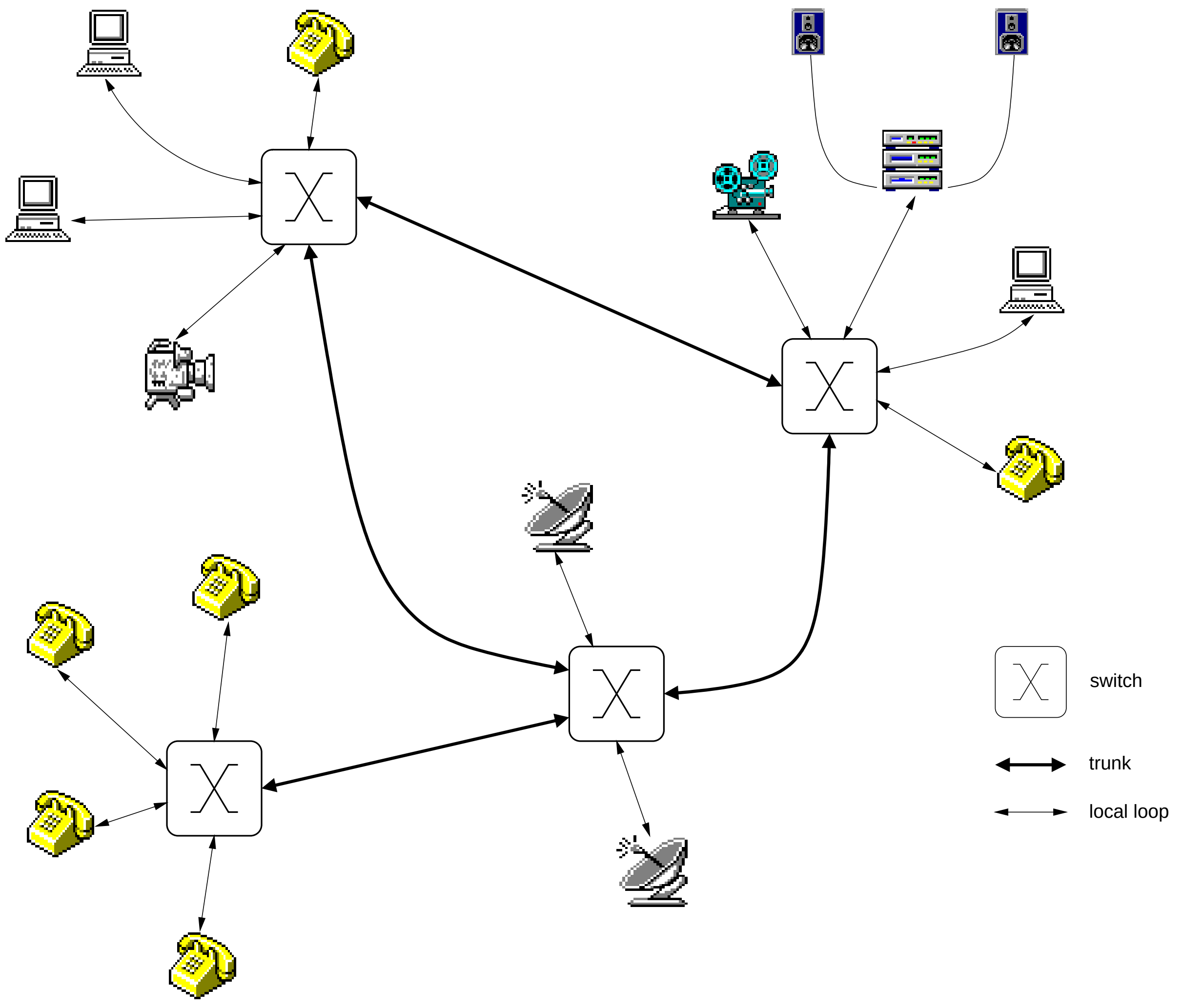


Figure 1.2: A typical communication network consisting of data sources, sinks, trunks and switches.

1. **Routing**: Packets must be forwarded from their arriving input to one or more destination outputs, as indicated by their routing tags.
2. **Queuing**: Owing to the statistical nature of packet traffic, it may happen that packets arriving simultaneously at different inputs want to go to the same output. Only one packet can be routed directly, while the others have to wait. This state of (temporary) output overload is called *output contention*. To resolve the output contention without dropping all those packets that lose the contention, *buffers* are necessary that must be organized such that packet sequence is maintained[4] on a per virtual connection basis, i.e. *sequence integrity* must be maintained.

The way in which the queuing function is implemented is one of the distinguishing characteristics of a packet switch and, as will be seen, the determining factor in a packet switch's efficiency. Therefore, the topic of this dissertation is the design of fixed-length-packet switch

[4]This requirement mainly stems from ATM, but is desirable also in other environments such as IP, where it is not strictly required.

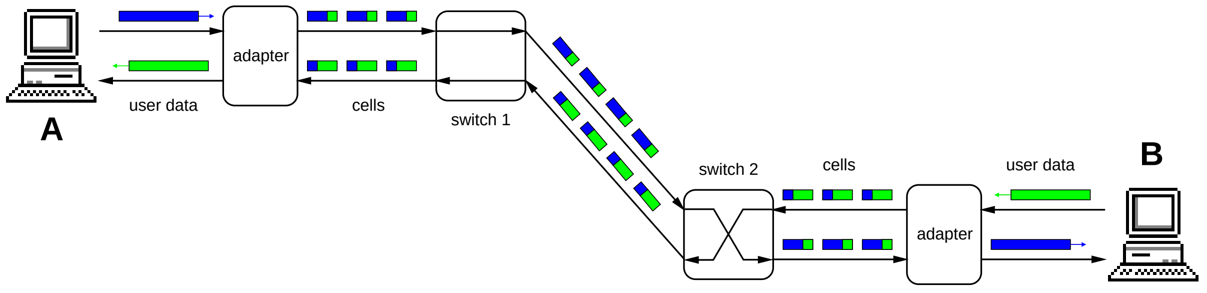


Figure 1.3: Adapter function: Computer A generates data it wants to send to Computer B. The adapter segments the data into smaller, fixed-length units, prefixes these with the address of computer B, and forwards them to the network. The packets reach their destination through switches 1 and 2, and are reassembled into the original stream by the receiving adapter, which hands the data over to the final recipient.

architectures to achieve optimal link utilization (throughput), which, as argued before, is an essential characteristic given current trends, while maintaining fairness among all its inputs and outputs. Furthermore, the architecture must be amenable to implementation at aggregate throughput rates in the range of hundreds of Gb/s to 1 Tb/s range, and allow scaling to even larger systems.

The next section gives a short overview of the problem domain, existing solutions and their drawbacks, and outlines the direction in which this dissertation will proceed.

1.2 The Problem Domain

The two traditional approaches to solving contention in packet switches are input queuing and output queuing. Both methods have been the topic of many research efforts in the past and are therefore well understood. Output queuing has usually been favored, because of its inherent performance advantages over input queuing: theoretically, output queuing offers ideal performance. A packet switch is considered to be ideal if it is *work-conserving*, i.e., no output line should ever be idle as long as at least one packet destined to it is in the switch. Thus, it can easily be verified that an output-queued switch is indeed work-conserving given any traffic pattern. Additionally, as packets are immediately placed in a buffer on the port they are destined to, it is easy to control packet delay and delay variation in an output-queued switch and thus provide QoS guarantees to individual flows.

Output-queued switches are often implemented in a shared-memory architecture rather than with dedicated output queues mainly because of a performance advantage due to increased buffer efficiency: to achieve the same performance fewer buffers per output are needed. However, with ever-increasing bandwidth demands, it is becoming increasingly difficult to realize switches that purely rely on the shared-memory approach because the aggregate shared-memory bandwidth is proportional to both the line rate and the number of ports. This internal *speed-up factor*, inherent to output queuing, is the main implementational challenge for a shared-memory switch. Shared-memory implementations based on SRAMs are limited by the memory access time, which for current state-of-the-art SRAMs is around 5 ns. Assuming a 64-byte wide data path, this limits the achievable aggregate switch throughput to about 51.2 Gb/s. Higher rates can be achieved by more exotic memory implementations, which typically are very costly in terms of silicon (gates and wiring) area, thus strictly limiting the amount of memory that can be implemented on a single chip.

The inherent limitations encountered in implementing shared-memory switches have been the main motivation behind intense research efforts in input-queued architectures over the past several years.

Contrary to its output-buffered counterpart, an input-buffered switch requires only buffers with a bandwidth proportional to the line rate, but *not* to the number of switch ports, seemingly making them inherently more scalable to higher line rates. However, classic input-queued architectures suffer from *head-of-line* blocking, owing to the use of FIFO queues. This means that a blocked packet at the head of the input queue can prevent packets behind it from reaching idle outputs, leading to severe throughput degradation. Recent research has been aimed at eliminating head-of-line blocking in input-queued switches, and raising their level of performance closer to that of output-queued switches.

The solution most commonly adopted employs input queues that are sorted by destination, which gives rise to an architecture with N^2 queues and a centralized scheduler that arbitrates among these queues. This arrangement, often referred to as *virtual output queuing* (VOQ) eliminates head-of-line blocking, but a suitable scheduling algorithm is required to obtain good performance. Unfortunately, the known optimal algorithms are too complex to implement at very high data rates, so sub-optimal, heuristic algorithms of lesser complexity, but also lesser performance, have to be used. The centralized scheduler is the bottleneck in this architecture in terms of implementational complexity. Furthermore, support for multicast and provisioning of QoS classes is substantially more difficult than in an output-queued architecture.

To summarize, we can say that switch architectures relying purely on output queuing cannot scale to high throughput because of memory bandwidth limitations, whereas purely input-queued architectures are faced with a poorly scalable centralized scheduler. With the shared-memory architecture the tradeoff is between memory bandwidth and memory size (which translates into switch performance), while with the input-queued architecture it is between scheduler complexity and switch performance.

These considerations have given rise to a third class of packet-switch architectures: the *combined input- and output-queued* (CIOQ) class. It is within this class of architectures that we expect to be able to find the key to improving on the existing, non-combined architectures.

We conjecture that combining the core concepts of shared-memory output queuing and VOQ input queuing into one switch architecture using a simplified VOQ scheduling algorithm can lead to unique results.

Note that this Section has only provided a brief glimpse at the problem domain. Chapter 2 gives an extensive overview of existing packet-switch architectures and surveys the state-of-the-art within each of the classes of architectures.

1.3 Design Task

The goal is to derive a packet-switch architecture that offers close to ideal performance, guarantees fairness among all its inputs and outputs, behaves robustly under unfavorable traffic patterns, supports multicast traffic and QoS provisioning, can be implemented in VLSI at throughputs up to the Tb/s range at a reasonable cost, and is inherently scalable to more and/or faster ports. As indicated above, we will look at the class of CIOQ architectures to find a candidate architecture that fulfills these requirements.

1.4 Dissertation Organization

This dissertation is structured as follows: Chapter 1 (this one) is a general introduction, containing a brief historical perspective, some background information on communication networks and the role of switching therein, and introduces the problem domain and the mission statement. Chapter 2 gives an overview of existing packet-switch architectures, categorized into three main classes of architectures based on queuing discipline. The state-of-the-art architectures in each class are reviewed and their merits and drawbacks assessed. Chapter 3 presents our novel CIOQ architecture. Its design principles are explained and its operation is detailed. We evaluate its performance under a range of traffic characteristics by means of both analysis and performance simulations, and compare it with existing approaches. Finally, we discuss implementational aspects and consider scalability issues. Chapter 4 proposes a novel method and algorithm for efficient and fair multicast support that can be integrated seamlessly into the switch architecture introduced in Chapter 3. We assess its performance characteristics by means of simulations and propose a straightforward hardware implementation. Chapter 5 describes the details of deadlock-free frame mode operation in the proposed architecture. Chapter 6 presents a practical application of the architecture and methods introduced in the previous two chapters, in the form of the PRIZMA family of packet switches. Finally, Chapter 7 summarizes the conclusions we arrived at in the course of the work presented in this dissertation, and indicates directions for future research.

1.5 Summary

Global bandwidth demand is growing at an exponential rate, fueled mainly by the proliferation of the Internet and its establishment as a worldwide communication medium encompassing a wide spectrum of applications. Optical transmission technologies such as wavelength-division multiplexing have allowed these growth rates to be sustained by offering a way to multiply the available transmission bandwidth in optical fibers by large factors. So far, the mainly electronic routing and switching network nodes have (in a way) been able to keep up by exploiting the advances in silicon technology, but already the available transmission bandwidth is starting to outstrip the available routing and switching capacity many times, a situation that is not likely to improve any time soon. All-optical switches are very promising to achieve very high throughput, but have two distinct disadvantages: first, their switching granularity is very coarse, which prevents per-packet switching, and second, optical buffering is cumbersome and impractical. Therefore, electronic packet switches are expected to continue to play a large role for a long time to come, provided that architectures that can scale to the desired throughput in the multi-terabit range are being developed. Thus, the aim of this dissertation is to develop a high-capacity, scalable, and practically realizable (using current technologies) packet-switch architecture.

Chapter 2

An Overview of Packet-Switch Architectures

We categorize packet-switch architectures based on queuing discipline, and provide an overview of existing packet-switch architectures, with the main focus on providing insight into the most recent developments in each class. The chapter concludes with a discussion on the merits and drawbacks of the most relevant architectures, based upon which the requirements for a new switch architecture, to be further developed in subsequent chapters, are established.

2.1 Classification of Switch Architectures

The purpose of this chapter is *not* to provide a comprehensive overview of all packet-switch architectures proposed over the past 10 to 15 years. This has been done quite adequately by a number of switch architecture overview papers, such as [Hluchyj88b], [Rathgeb88], [Ahmadi89b], [Tobagi90], and [Pattavina93]. A more recent overview focusing especially on space-division architectures is presented in [Awdeh95].

These papers use many different ways to classify switch architectures into categories. Some of these criteria are blocking vs. non-blocking, buffering strategy (input-buffered, output-buffered or combined), lossy vs. lossless, single-stage vs. multi-stage, buffer implementation (partitioned, grouped, shared), time- or space-divided (or combined TST/STS).

For the purpose of this dissertation many of these categorizations focus too strongly on implementation details. For instance, whether a switch achieves its function in a time- or space-divided manner is not relevant from a performance perspective, as one can theoretically map either implementation to the other, resulting in the same external behavior. In the following overview of packet-switch architectures, the main focus will be on the queuing discipline. Of course, it can be argued that the choice of queuing discipline is also an implementation issue, and from a high-level point of view—regarding the switch as a black box that moves data from A to B—this is certainly true. However, for the purpose of this dissertation, the correct abstraction level is that of the internal switch architecture, and thus the queuing discipline, because this is the determining factor of switch performance. Moreover, in this overview, we shall not concern ourselves with specific hardware implementation details of these architectures.

The remainder of this chapter is organized as follows: In Section 2.1.1 we distinguish between blocking and non-blocking architectures. Section 2.2 briefly reviews a not so useful architecture with no buffers at all, merely as a reference. Sections 2.3 and 2.4 treat the classic input- and output-queued architectures respectively. The principle of shared queuing is reviewed in Section 2.5. Because of the resurgent interest in input-queued architectures, a large number of papers on this topic have been published in recent years. Given the significance of the results of this research, and as there is no comprehensive overview available in the literature today, Section 2.6 provides a comprehensive survey of this new generation of input-queued architectures. The latest development trends in the field are towards new combined input- and output-queued (CIOQ) architectures, which will be surveyed in Section 2.7. Finally, in Section 2.8 we discuss the merits and drawbacks of the most widely used architectures today, and summarize the key points that can be gleaned from the overview presented.

2.1.1 Blocking and non-blocking switch architectures

In packet-switching architectures two forms of *blocking* can be distinguished. First, owing to the statistical, non-deterministic nature of packet arrivals at the switch, more than one packet destined for the same output may arrive at the inputs simultaneously. Of these packets only one can be forwarded to the output, while the others have to wait. Therefore, the switch has to be equipped with buffers to accommodate packets that cannot be forwarded immediately. This inevitable form of blocking is called *output contention*. The second form of blocking is related to the internal structure of a packet switch, e.g., in a multi-stage switch contention for fabric-internal links may occur, even among packets destined for different output ports. Architectures that exhibit this behavior, e.g. Banyan networks, are referred to as *blocking*. Clearly, this property is undesirable, as it has highly adverse effects on switch performance.[1]

A switch architecture is called *non-blocking* when it satisfies the requirement that an arriving packet destined for any output to which no other packets are destined can be forwarded immediately, regardless of the destinations of all other arriving packets. From here on, we will focus on this type of switch architecture.

2.1.2 Sequence-preserving

Depending on the switch-fabric application, there may be a strict requirement regarding the order in which packets leave the switch. In particular, certain applications, such as ATM, impose the strict requirement that packets belonging to the same flow *must* be delivered in the order they arrived in. For the switch, this means that sequence of packets belonging to a given combination of input port, output port, and virtual lane must be maintained.

2.1.3 Single-stage vs. multi-stage switches

Fig. 2.1 depicts a $KN \times KN$ Benes multistage switch fabric consisting of three stages of K $N \times N$ switches each. For large networks, the multistage approach is much cheaper in terms of the required number of switch elements than single-stage port expansion. In general, for

[1]For Banyan networks, the blocking property can be avoided by preceding the Banyan with a Batcher bi-tonic sort network. The *Starlite* switch [Huang84] was the first to employ the Batcher-Banyan self-routing structure.

an $M \times M$ fabric, with $M = KN$, KS switch elements are required for an S-stage fabric compared to K^2 for a single-stage fabric (see Section 6.3.2).

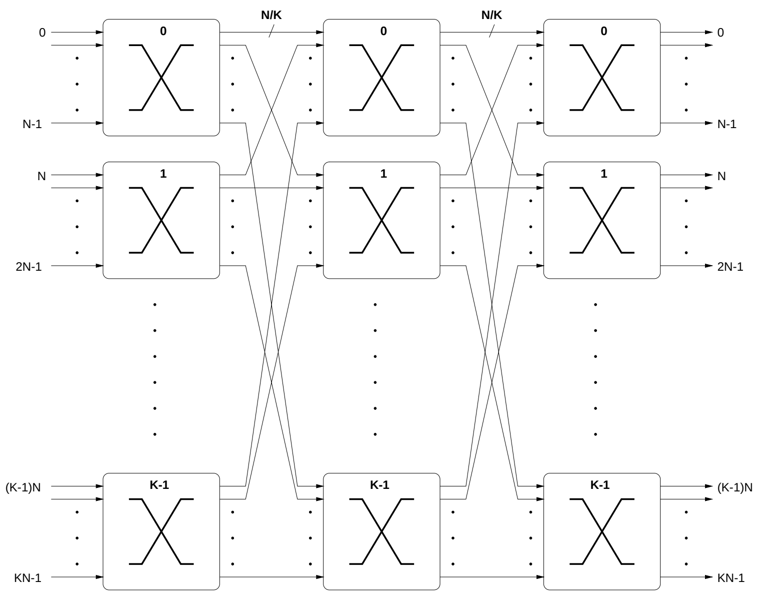


Figure 2.1: A $KN \times KN$ multistage network consisting of $3K$ $N \times N$ switch elements.

We will only consider single-stage architectures here. Single-stage architectures have a strong performance advantage over multi-stage architectures, but inherently are not as scalable. Therefore, as bandwidth demands continue to rise, there will be a growing need for multi-stage fabrics because these can scale to many times more ports than any single-stage architecture can. However, we consider the multi-stage problem as partly orthogonal—after all, a multi-stage fabric is a cascade of single-stage switches. Still, the extension of a given single-stage architecture to multi-stage is far from trivial. To convey the complexity of making multi-stage fabrics work, let us briefly mention the main issues:

- Network topology: which topology—e.g. Benes, Banyan, BMIN, Hypercube, Torus, etc.—is the right one for a given application?

- Performance: both in terms of delay because more stages means more latency and throughput because multi-stage fabrics are often internally blocking owing to a combination of the fabric's interconnection topology, static routing, or higher-order HoL blocking [Jurczyk96].

- Fabric-internal routing: static routing is easy to implement, but leads to poor performance under unfavorable traffic patterns because it cannot adapt to congestion situations, whereas adaptive routing can improve performance, but is expensive to implement and may lead to out-of-order delivery. Either source routing or per-hop look-up is required.

- Flow control: to maintain performance, prevent higher-order HOL blocking, and guarantee fabric-internal losslessness, proper flow control is required. To obtain maximum performance, global end-to-end flow control may be necessary, which presents significant scaling problems.

- Multicast support: multicast traffic requires both proper duplication strategies and destination set encoding with corresponding lookup tables in every switch to maintain scalability (carrying full bitmaps is clearly not feasible for fabrics with hundreds of ports).
- Practical implementation and cost issues.

Regardless of the many obstacles, we conjecture that the insights and results obtained in this dissertation for single-stage fabrics will also prove valuable in designing high-performance multi-stage fabrics. We also conjecture that the boundaries of single-stage architectures will be pushed to their limits, before the transition to multi-stage fabrics is made.

2.1.4 Buffer placement

In the switch architecture the choice of buffer (or *queue*) placement to resolve output contention is of great importance for the overall switch performance. Throughout this work, we assume time-slotted operation, where time is divided into fixed-length slots called *packet cycles*. A packet cycle is the transmission time of one packet. One packet cycle equals L/B, where L is the length of a packet in bits and B is the link rate (port speed) in bits per second (b/s). To evaluate switch performance we will employ the following metrics:

- Average **throughput**: this is defined as the average number of packets that exit the switch during one packet cycle divided by the number of switch output ports. Equivalently, this is expressed as the fraction of time the output lines are busy (output utilization).
- Average **packet delay**: The average delay encountered by a packet while traversing the switch (usually expressed in packet cycles).
- **Packet-loss rate**: The percentage of arriving packets that are dropped by the switch owing to lack of buffer space.
- **Packet delay variation**: The distribution of packet delays around the mean, also referred to as *jitter*.

The two classic queuing solutions are *output queuing* and *input queuing*, which will be treated in Sections 2.3 and 2.4, respectively.

2.2 No Queuing

Here, we will have a brief look at the simplest way of queuing: no queuing at all, i.e. any packet that cannot be forwarded immediately is simply discarded. This case has been studied by Patel in 1981 [Patel81] and the outcome is that under the assumption of i.i.d. random sources,[2] the

[2]A common assumption is that the switch is fed by traffic sources that are independent and identically distributed (i.i.d.) on all inputs, with each source generating a packet in any given cycle with probability p (and therefore probability $1 - p$ of not generating a packet), with uniformly distributed destinations (each packet is destined for any of N outputs with equal probability $1/N$). This type of traffic is commonly referred to as *Bernoulli* or *uniform random* traffic, with input load $p * 100\%$. See also Appendix C.4.

maximum throughput reaches 63%, while 37% of all packets are discarded (see Appendix A for a derivation of this result). Obviously, this packet-loss rate is absolutely inacceptable for any practical purposes, so we will have to look at switch architectures that incorporate some form of queuing.

2.3 Input Queuing

First, we formally define the concept of input queuing, see Definition 1.

Definition 1 (Input Queuing) *A switch belongs to the class of* **input-queuing** *switches if the buffering function is performed* **before** *the routing function.*

The classic and simplest solution to resolve output contention, other than the no-queuing option, is to place buffers at the switch fabric inputs and storing incoming packets that cannot be forwarded immediately in these buffers, see Fig. 2.2. The reason input queuing has been and still is quite popular is that the bandwidth requirement on each buffer is only proportional to the port speed and not to the number of switch ports. Only one read and one write operation are required per packet cycle on each buffer, so that with a port speed of B, the total bandwidth requirement equals $2B$ per buffer and $2NB$ aggregate for a switch of size $N \times N$.

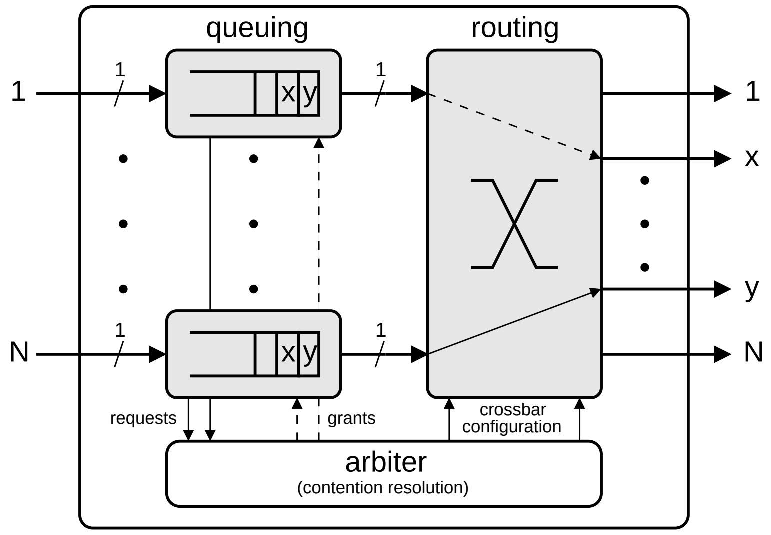


Figure 2.2: Input queuing with FIFO queues. Note the HoL blocking on input 1 of the packet destined to output x owing to contention for output y.

Of course, output contention must still be resolved. This task is executed by the *arbiter* unit depicted in Fig. 2.2. In each packet cycle, a selection is made from the set of head-of-queue packets to be forwarded to the outputs during this cycle. This selection must satisfy the condition that at most one packet is forwarded to any single output. Additionally, the selection policy must be designed such that fairness among inputs and outputs is guaranteed. Once the selection has been made, the selected packets are removed from their buffers and forwarded through the routing fabric, which can be a simple crossbar. The *three-phase algorithm* first presented in [Hui87] is an example of such a selection algorithm.

In an input-queued switch with FIFO queues, an input has at most one packet that is eligible for transmission, namely the HoL packet. When output contention occurs (multiple inputs have a packet for the same output), one and only one of these inputs is granted permission to transmit according to a certain rule. Various propositions to arbitrate amongst HoL packets destined to the same output have been proposed and investigated (A more detailed overview with references can be found in [Awdeh95, Section 3.2]):

- Random selection: An input is selected randomly from the set of contending inputs. This requires implementing (pseudo-)random selections.
- Round-robin selection: The input queue beyond the one served last receives highest priority. Each output must maintain a pointer to the input it served last.
- Longest-queue selection: The input with the largest number of waiting packets is served, which entails keeping track of input queue occupancy data.
- Least recently used selection: The input that has been served least recently is served. This requires that service times be maintained for each input.

In [Karol87] it is shown that the input queuing scheme limits switch throughput to a theoretical maximum of merely $2 - \sqrt{2} \approx 58.6\%$ of maximum bandwidth under the assumption of independent, uniform Bernoulli traffic sources. Owing to the increased correlation in the traffic arrivals caused by the buffering, adding buffers will actually *degrade* throughput compared to the no-buffer case! Additionally, it has been shown that the HoL selection policy does not affect throughput, although it may affect average delay to a small degree [Iliadis90a, Iliadis91b].

Hui and Arthurs [Hui87] derived the following expression (Eq. 2.1) for the average delay D as a function of the average input load p, assuming an infinite number of switch ports ($N \to \infty$), infinite buffering and a random selection policy:

$$D = \frac{(2-p)(1-p)}{(2-\sqrt{2}-p)(2+\sqrt{2}-p)} - 1, \tag{2.1}$$

with $0 \leq p < 2 - \sqrt{2}$.

The essential cause of this low maximum throughput is that a blocked packet at the head of an input queue can prevent other packets behind it destined to idle outputs from being forwarded. This phenomenon is widely known as *head-of-line (HoL) blocking*, see also Fig. 2.2.

Li showed that throughput decreases with increasingly correlated traffic (burstiness), to as little as 50% for strongly correlated traffic [Li92]. Cao [Cao95] also analyzed an input-queued packet switch under correlated traffic, and derived an approximate closed form solution for the maximum throughput, $\lambda = N/(2N-1)$, under the key assumption that the correlation is strong (the probability that more than one burst ends on more than one input simultaneously is negligible).

In [Tobagi90], [McKeown97b] and [Chuang98] the term *work-conserving* is used to qualify switches.

Definition 2 (Work Conserving) *A switch that is* **work conserving** *will serve any output for which at least one packet is available in the switch.*

This is a crucial definition because it really highlights the essence of what a switch should strive to achieve: *to keep its output lines busy!* By definition, a switch that is truly work-conserving achieves maximum throughput and lowest delay, which confirms the importance of this concept. However, note that additional requirements, in particular fairness, may conflict with strict work conservation.

From the HoL blocking example, it immediately becomes clear that this type of input-queued switch is *not* work conserving—there is a packet waiting for output x, but it cannot be served because it is blocked by a packet destined to output y.

2.3.1 Implementations

GIGAswitch

DEC's GIGAswitch[3] [Souza94] is an example of a crossbar-based, input-queued packet switch; it has 36 inputs and 36 outputs, each running at 100 Mb/s, with a single FIFO queue per input. To alleviate HoL blocking, outputs can be grouped together into so-called *hunt groups* to form a single logical high-bandwidth port. Packets addressed to a particular hunt group may be transmitted on any of its corresponding physical ports. Out-of-order delivery may therefore occur if no special precautions are taken.

2.3.2 Improvements

The original three-phase algorithm for input-queued packet switches, described by Hui and Arthurs [Hui87], is based on FIFO queues and consists of request, grant, and send phases. As pointed out, it has been shown to lead to HoL blocking, which limits the maximum throughput to merely 58% [Karol87]. Spurred on by this discovery of the inherent performance disadvantage of FIFO-based input buffering compared to output buffering, various approaches were proposed to overcome this limitation. The general approach is to make more than one packet from each input queue eligible for transmission. This section will give an overview of the early approaches, which were mainly based on scheduling packets into the future, or on the use of random access buffers instead of FIFOs.

Port grouping

Pattavina proposed the grouping of multiple physical outputs into one logical output [Pattavina88]. Such *channel grouping*[4] results in a switch with fewer, but faster ports. The arbitration will redirect a request for a logical port to the first corresponding idle physical port, so that HoL blocking is reduced. This principle is applied in the GIGAswitch architecture [Souza94]. Obara [Obara91, Obara92a] proposed a similar approach to improve performance; see the following subsection.

[3]GIGAswitch is a trademark of Digital Equipment Corporation.

[4]This is similar to the PRIZMA link-paralleling concept, see Section 6.6, although the link-paralleling concept is designed to maintain also the packet sequence.

Scheduling packets into the future

Obara and Yasushi [Obara89] proposed a distributed contention-resolution mechanism that dynamically allocates transmission times to packets stored in the input queues. The algorithm consists of request, arbitration, and transmission phases. In the request phase, each input sends a request for its HoL packet to the requested output. Each output j counts the requests directed to it, R_j. Output j keeps track of the next available transmission slot T_j. It assigns transmission slots T_j through $T_j + R_j - 1$ to the requesting inputs, and updates T_j to $T_j + R_j$. Upon reception of a transmission slot, an input checks its transmission table; if the assigned slot is not yet reserved, the packet is removed from the queue and will be transmitted at the assigned timeslot. Otherwise, the packet remains at HoL and has to retry in the next slot. This scheme reduces the HoL blocking, because packets may be scheduled for future timeslots, thus bringing forward new work (HoL packets) sooner than the three-phase algorithm. However, as timeslots assigned by an output may have already been reserved at the requesting input, slots may be wasted, thus limiting maximum throughput. Approximate analysis and simulation [Obara89] show that maximum throughput under uniform i.i.d. traffic is about 76% (for a 16×16 system), up from 58% for conventional FIFO buffers.

Matsunaga and Uematsu [Matsunaga91] also propose a scheme based on timeslot reservations. Contention resolution is performed by a number of reservation tables RT_i, $1 \leq i \leq N$, one per input. RT_i indicates the reservation state of the outputs at $t + i - 1$, where t is the current timeslot. Each RT contains one entry per output, indicating whether the output is reserved at that time. Each input i attempts to match a packet in its queue to a non-reserved slot in the corresponding table RT_i, and reserves the corresponding entry when successful. The packet selected is removed from the queue and will be transmitted at $t + i$. Each timeslot, RT_i is copied into RT_{i-1}, RT_1 is discarded, whereas RT_N is set to all non-reserved. The approach requires FIRO (First In, Random Out) buffers, because the algorithm performs a search through each input buffer (up to a certain depth d) to find a packet that can occupy a non-reserved slot in the reservation table. For a search-depth $d = 16$ and a 16×16 system, the maximum throughput under uniform i.i.d. traffic is about 90%. Special care must be taken to prevent unfairness among inputs because input 1 has a much smaller chance of reserving a slot than input N, owing to the way the tables are shifted from N to 1 as time progresses.

In [Obara91] an improvement to the algorithm of [Obara89] is presented. To reduce the probability of assigned timeslots being wasted, the inputs of the switch can be grouped into groups of size k. From each group of inputs, k packets may depart in one timeslot. Thus, up to k conflicting timeslot assignments can be managed by an input group, which drastically reduces the probability of a timeslot being wasted. Simulations have shown that maximum throughput increases to over 90% for a group size of 4 for a 64×64 switch under uniform i.i.d. traffic. A further improvement, proposed in [Obara92a], is the grouping of outputs as well as inputs, thus creating a combined input- and output-queued switching fabric.

[Obara92b] addresses the problem of what is termed "turn-around time" (TAT) between input controllers and a centralized contention-resolution unit. The TAT equals the amount of time it takes to complete the scheduling of one packet (request-acknowledge). By pipelining the requests to the contention-resolution unit, throughput degradation due to long TAT can be prevented.

In [Karol92] a different improvement to the scheduling algorithm of [Obara89] is presented, which attempts to reduce the amount of wasted timeslots by *recycling* them.

Window-based approach

The window-based approach was proposed by Hluchyj and Karol in [Hluchyj88b]. The strict FIFO queuing discipline of the input queues is relaxed, allowing other packets than the ones at HoL to contend for idle outputs. Each input queue first contends with its HoL packet. The HoL packets that "win" the contention and their respective outputs are removed from the contention process. In the next iteration, inputs that lost contention contend for any remaining idle outputs with the next packet in their queue. This process is repeated until either all outputs have been matched or w rounds of contention have been completed, where w is the window depth; $w = 1$ corresponds to strict FIFO queuing. For uniform i.i.d. traffic this approach improves throughput considerably, even for small w. For a 16 $\times$ 16 switch, throughput increases to 77% for a window size $w = 3$, and to 88% for $w = 8$ [Hluchyj88b].

However, as traffic exhibits a more bursty nature, the number of packets destined to different outputs within the window w decreases rapidly, so that a much larger window is required to achieve the same throughput, which requires more processing power in the scheduling unit.

Note that the window-based schemes essentially search the input queue for packets destined to idle outputs. Typically, the maximum input queue size Q is much larger than the number of outputs N. Therefore, it makes sense to sort the input queue based on the destination of the packets; this way, only N packets need to be considered, instead of up to Q. This arrangement was proposed as early as 1984 [McMillen84], and has received a significant amount of attention in recent years. In Section 2.6 we will study this arrangement in depth.

2.3.3 Other variations

Other solutions involve running the switch fabric at higher speeds than the in- and output lines, or using multiple switch fabrics in parallel. These include the following (see also [Liew94] and [Awdeh95, Section 3.4]):

1. *Output-port expansion*: Output-port expansion with a factor F requires an $N \times NF$ switch fabric, where each group of F physical output ports corresponds to one logical output. This way, up to F packets can be delivered to an output in each cycle. This approach is conceptually similar to output-channel grouping.

2. *Input-port expansion*: Input-port expansion with a factor F requires an $NF \times N$ switch fabric, where each group of F physical input ports corresponds to one logical input. This allows F packets from each input to contend in each cycle. This differs from the windowing scheme in that more than one packet can be selected from the same input.

3. *Switch speed-up*: If the switch operates at a speed F times faster than the links, F packets can be transported from a single input, and also F packets can be delivered to an output in one cycle. (More conventional speed-up schemes allow multiple packets to be delivered to one output, but allow only one packet per input to be transmitted [Oie89]). As multiple packets are being delivered to an output in one cycle, some degree of output queuing is also required. Therefore this type of switch really belongs to the combined input/output queuing class as described in Section 2.7.

2.4 Output Queuing

We formally define the concept of output queuing, see Definition 3.

Definition 3 (Output Queuing) *A switch belongs to the class of* **output-queuing** *switches if the buffering function is performed* **after** *the routing function.*

The alternative classic solution to input queuing is output queuing, where the buffers are placed at the switch fabric outputs, see Fig. 2.3. Theoretically, output queuing offers maximum, ideal performance, because

- HoL blocking does not exist, thus lifting the throughput limitation from which input queuing suffers, and
- contention is reduced to a minimum—only output contention occurs, which is inavoidable because of the statistical nature of packet-switched traffic, but there is no input contention, which leads to lower average delays also at low utilization than in input queuing.

It can be shown theoretically that a non-blocking output-queuing switch having queues of infinite size offers the best performance in terms of both throughput and delay. In fact, this type of switch is the only type that is truly *work conserving* in the sense of Definition 2 under *any* traffic pattern.

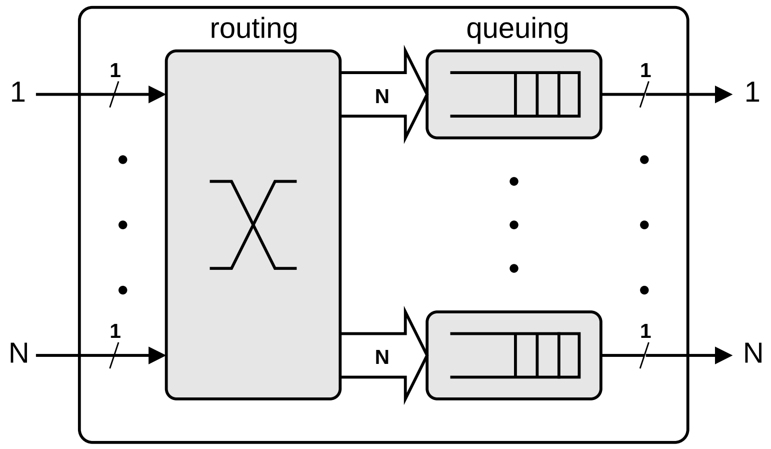


Figure 2.3: Output Queuing: routing takes place before buffering. Note that the write bandwidth requirement on the output queues equals N writes/cycle.

However, output queuing comes at a significant cost. The bandwidth requirement on a single buffer is now proportional to both port speed and number of ports, because in one packet cycle N packets may arrive destined for the same output, so that N writes (N arriving packets) and one read (one departing packet) are required. Thus, the bandwidth requirement equals $(N+1)B$ per output queue, with B being the port speed and N the number of input ports. Thus, the aggregate bandwidth equals $N(N+1)B$, which is quadratic in the number of ports, and therefore output queuing is inherently less scalable to more and/or faster ports than input queuing.

The average queuing delay D of an output-queued switch fabric with infinite-sized queues under Bernoulli traffic, as $N \to \infty$, equals [Hluchyj88b]

$$D = \frac{p}{2(1-p)}\ , \tag{2.2}$$

where p $(0 \leq p < 1)$ is the average input load. The delay versus input load curves for both input and output queuing (Eqs. 2.1 and 2.2) are displayed in Fig. 2.4.

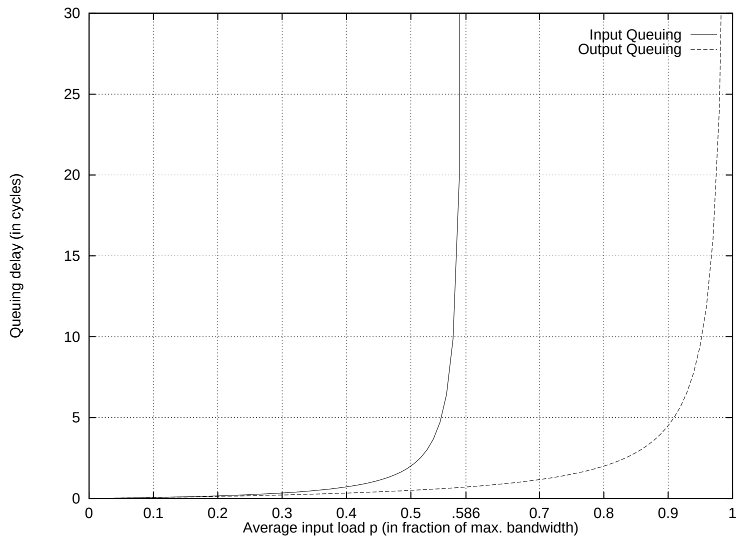


Figure 2.4: Average queuing delay of input vs. output queuing.

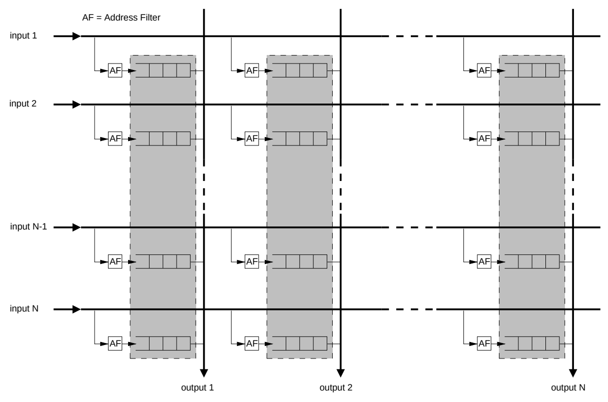
As already pointed out, the major drawback is that both the speed of the internal interconnect and the memory bandwidth have to be N times higher than the external line speed in order to be able to transport N packets to a single output queue in one cycle, whereas an input-queued switch suffers no such limitation. We call this the *internal speed-up factor* (see also Section 2.7). This requirement makes output queuing in general an expensive solution because here buffers with an access rate N times faster than in an input-queued architecture are required, making the output-queued solution less suitable for very-high-speed packet switches. Additionally, it does not scale well to larger switches, as the speed-up factor increases with switch size.

2.4.1 Examples

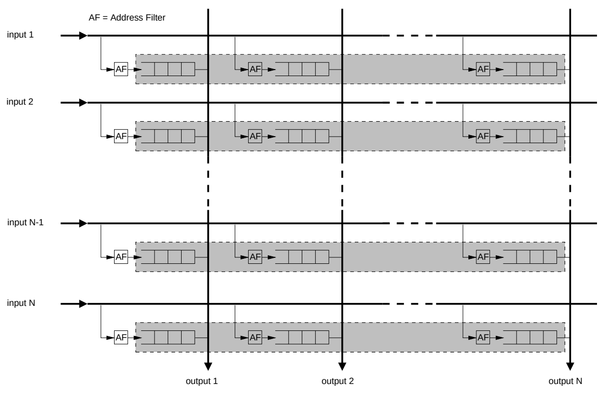
Examples of output-queued switch architectures are the Buffered Crossbar, the Knockout, the Gauss, and the ATOM switch.

Buffered Crossbar switch

One of the oldest packet-switch architectures is the buffered crossbar, see the schematical representation depicted in Fig. 2.5. A 1982 patent [Bakka82] describes a switching circuit employing this architecture. It has resurfaced numerous times, for instance in the *Bus Matrix switch* (see Fig. 2.5a), which has been implemented by Fujitsu, described in [Nojima87]. In [Rathgeb88] it is referred to as the *Butterfly switch*. Gupta et al. [Gupta91b] introduce a variant called *limited intermediate buffer switch*, combining single-packet crosspoint buffers with input queuing. Remarkably, [DelRe93] presents the buffered crossbar architecture as a "novel" one under the name of *multiple queuing*.



(a) Bus Matrix switch architecture, a.k.a. the Butterfly switch. Some form of arbitration (not shown in the picture) is required to select which of the queues connected to the same output line is allowed to transmit. The shaded areas indicate buffer sharing per output.



(b) Bus Matrix switch with buffer sharing per input.

Figure 2.5: Buffered Crossbar switches.

That it indeed belongs to the class of output-queued switches may not be apparent at first,[5] but consider that when a packet is stored in one of the crosspoint queues, it has already arrived at its destination output port. The routing function is actually performed by the address filters. Therefore, routing is done before queuing, classifying this architecture as an output-queued one.

Logically, it can be regarded as an output-queued switch with distributed output queues, or conversely, an output-queued switch can be viewed as a bus matrix that shares the buffer space available for one output line, indicated by the shading in Fig. 2.5a. Note that the bandwidth requirement in one shaded area (one output queue) equals N writes (N incoming packets) and one read (1 outgoing packet).

The main drawback of the architecture is that N^2 separate buffers have to be implemented. Also, each crosspoint buffer and therefore the input-output pair associated with it have a fixed amount of buffer space. This is also regarded as a drawback, as no buffer sharing among inputs and/or outputs is possible. In [Gupta91b], the former problem is tackled by reducing the size of each buffer to one. Alternatively, the buffer space of all queues belonging to the same input can be pooled into one shared buffer to achieve better buffer efficiency, see Fig. 2.5b. Both the case of pure crosspoint buffering and crosspoint buffering with shared buffers per input were studied analytically in [DelRe93]. Results indicate that delay-throughput performance under uniform Bernoulli traffic is equal to that of classical output queuing—this comes as no big surprise, because it can easily be seen that, given infinite buffers, this architecture is in fact work-conserving, just like the classical output-queued architecture.

[5]It was not apparent to the authors of [DelRe93], who classified it as an input-queued architecture.

Knockout switch

An example of an output-queued architecture is the Knockout switch [Yeh87], which derives its name from the key design principle it is based on: the *Knockout principle*.

The principle states that for a given switch size $N \times N$, it is possible to select a value L such that the probability that more than L packets arrive at one output simultaneously becomes arbitrarily small, with $L \ll N$. Applying this principle to packet switches, the Knockout switch can accept up to L packets for one output at a time, so that output queuing is achieved with an internal speed-up factor of only L, instead of N for conventional switches. The price paid for this is the inherent lossiness of the fabric in case more than L packets arrive at an output queue, which is a major drawback. Moreover, the dimensioning of L to achieve a given loss ratio strongly depends on traffic characteristics. The random i.i.d. arrivals assumed in [Yeh87] are not realistic, and burstiness and non-uniformity can lead to much higher loss ratios, thus requiring much larger values of L. In [Yoon95] the Knockout switch is studied under non-uniform traffic patterns. The results indicate that packet-loss rate rises sharply (several orders of magnitude) beyond a certain load in the case of hot-spot traffic, and that under non-uniform traffic the packet-loss rate strongly depends on the number of concentrator outputs L. Therefore, it is difficult to select a reasonable value of L such that the Knockout switch achieves acceptable packet-loss rates under any traffic pattern.

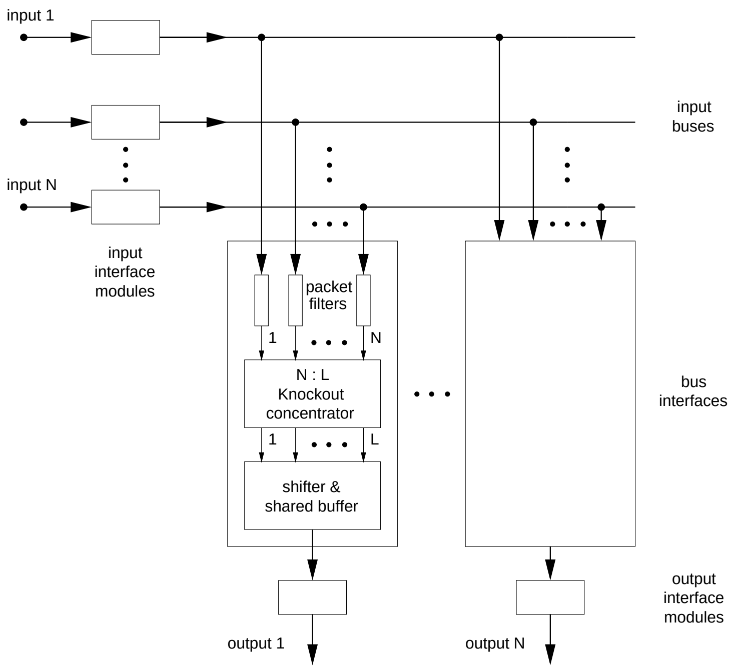


Figure 2.6: The Knockout Switch, from [Eng88b].

Fig. 2.6 displays the Knockout switch architecture. Incoming packets are transmitted via broadcast busses to the outputs. At each output a filter selects only those packets that are destined for this particular output. Next, the arriving packets are led through an N to L concentrator, which implies that packets will get lost here if more than L arrive at an output simultaneously, and are then written to a set of L specially organized FIFO queues that act as one FIFO with L inputs and one output, in order to achieve the required output-queue write bandwidth.

How to incorporate multicast services into the Knockout switch architecture is described in [Eng88a], and [Eng88b] describes a photonic version of the Knockout switch.

The Knockout principle can also be generalized to *groups* of outputs, instead of just individual outputs. This *generalized knockout principle* is proposed in [Eng89] as a technique to build large, scalable switch fabrics.

Gauss switch

The Gauss switch [DeVries90a, DeVries90b, Aanen92] is also based on the Knockout principle, and works in almost exactly the same way. The main difference is that the Knockout principle is not implemented using concentrator networks, but rather using a shift register to access the output buffer operated at a speed-up $L < N$. To access this shift register, packets are stored in small intermediate buffers, from where they attempt to grab a free slot on the shift register. When a packet does not succeed in accessing the shift register, and a new packet arrives from the same input in the next time slot, one of the two must be dropped. Therefore, the presence of these intermediate buffers offers an advantage in packet-loss rate compared to the Knockout switch, as shown in [Aanen92].

ATOM switch

Another example of an output-queued switch is NEC's ATOM switch [Suzuki89]. The ATOM architecture is conceived as a multistage configuration. The switch elements have dedicated FIFO buffers at each output, with a time-division-multiplexed bus of bandwidth NB to transfer all arriving packets from the inputs to the queues, which enables easy implementation of multicast service. Support for priority classes is provided by maintaining a linked list per priority in each output buffer. Clearly, the main drawback of the architecture is that the TDM bus bandwidth does not scale. Furthermore, as no flow control is provided, the static buffer allocation leads to poor performance under bursty or non-uniform traffic.

2.4.2 Variations

- *Output bandwidth expansion*: Running the outputs at a speed of s times the input link speed allows more than one packet to be removed from an output queue and delivered to an output link. Equivalently, the number of ports per output can be increased by a factor s.
- *Output buffer sharing*: Sharing all buffers among all outputs will cause no packets to be lost or blocked until there are $N * b$ packets waiting in total, where b is the available buffer space per output. This is a form of shared queuing, which Section 2.5 will look at in greater detail.

2.5 Shared Queuing

Both the input- and the output-queued architectures treated so far have assumed that each input c.q. output queue has a fixed amount of buffer space that is dedicated to this queue. This strategy may lead to suboptimum use of resources: for instance, in case some ports are heavily loaded while others are mostly idle, the former may obviously benefit from temporarily getting more resources, while the latter may get by with less. From this realization the concept of *shared queuing* was born.

This concept implies that buffer resources in a switch fabric are pooled together. All logical input or output queues or groups thereof can use resources from a buffer pool until the pool is exhausted. The aim is to achieve better resource efficiency.

Although the sharing concept has proven effective especially in reducing packet-loss rates, it can lead to unfairness, as packets from one particular input or destined to one particular output can monopolize the shared resource, thus actually causing performance degradation, as was also pointed out in [Hluchyj88b] and [Liew94]. This issue will be treated in Chapter 3.

The following two sections provide some examples of both shared input and shared output queuing. Practical implementations incorporating the shared-queuing approach include the Starlite switch, the Vulcan switch, the ATLAS switch, the PRIZMA switch, the Swedish ATM switch core [Andersson96], and the Hitachi ATM switch [Kuwahara89, Kozaki91].

2.5.1 Shared input queuing

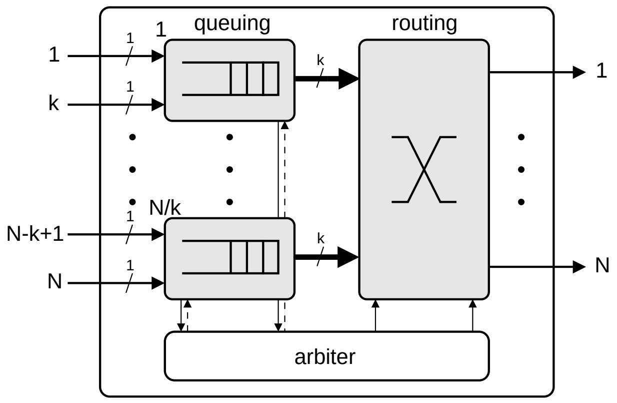


Figure 2.7: Grouped Input Queuing, from [Tao94].

In [Tao94] an input-buffered scheme is introduced that groups sets of k inputs together, providing a shared buffer per group, see Fig. 2.7. In each cycle, up to k packets are selected from each group. Note that the queuing discipline is no longer strictly FIFO, so that HoL blocking is alleviated. The packets to be forwarded are selected using the following algorithm:

1. Assign the “idle” flag to all output ports;
2. Randomly choose a grouped queue;
3. From this grouped queue, choose up to k packets having differing, idle destinations in a queuing-age sequence. Mark the selected outputs “occupied”.
4. Repeat step 3 for the next grouped queue, until either all grouped queues have been processed or all output ports are marked “occupied”.

This algorithm is very similar in operation to the RPA algorithm described in [Ajmone97] (see also Section 2.6) when $k = 1$, except that RPA is explicitly based on virtual output queuing, whereas this is implicit in this approach: although the packets in the shared buffer are not sorted by destination, packets to any destination can depart from any grouped queue at any time. Clearly this algorithm is not very efficient: in the worst case, an entire grouped queue must be searched in each of a maximum of N iterations.

Simulation results for grouping factor $k = 2$, 4, and 8 show that using this scheme throughput drastically improves compared to FIFO input queuing. However, what the authors [Tao94] do not explicitly mention is that this is not due to the grouping but rather to the non-FIFO character of the grouped queues. The grouping effect, as also shown in [Tao94], leads to much lower loss rates, in particular when considering bursty traffic.

Starlite switch

The Starlite switch [Huang84], developed at AT&T Bell Laboratories, presents an example of shared input queuing.[6] There is no dedicated buffer per input, but rather one buffer that stores all packets that could not be delivered owing to output contention. Excess packets are dropped. The buffered packets are fed back into the switch (hence the name ‘shared recirculating queue’) through M additional input ports, dedicated to serving packets being recirculated. This reduces the effective size of the switch because an $(N + M) \times (N + M)$ size switch is required to realize an $N \times N$ switch, and may cause packets to be delivered out of sequence. The size of the recirculation buffer and the number of input ports M dedicated to it can be tuned to meet specific packet loss requirements.

2.5.2 Shared output queuing

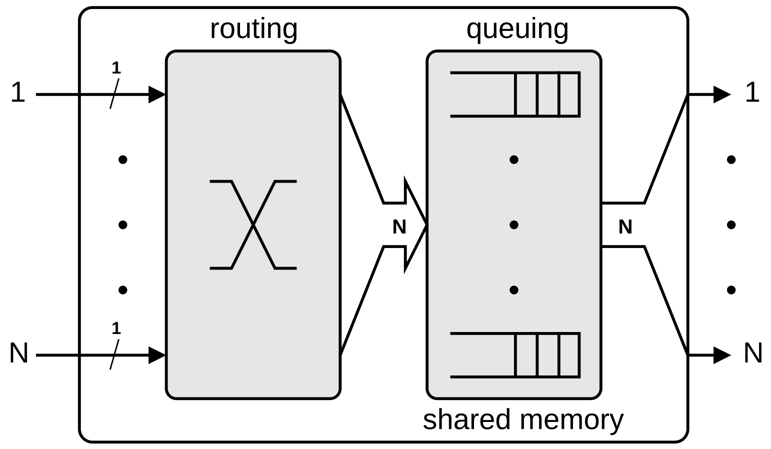


Figure 2.8: Shared output queuing, logical structure.

[6]This architecture employs a form of shared input queuing because queuing is in fact done before switching, although packets are passed through the routing fabric before being buffered. However, once buffered, they need to be routed *again*.

Most practical implementations of output-queued switch fabrics employ a shared output buffer approach, where all output queues share one common memory. This approach has a significant advantage over dedicated output queues from an implementational viewpoint: the aggregate bandwidth through the shared memory equals $2NB$, which is in fact equal to the aggregate bandwidth requirement for the input-queued architecture, instead of $N(N+1)B$ in case of dedicated output buffers, see Fig. 2.8. This tremendous saving can be understood easily by realizing that no switch, regardless of its architecture, ever needs to receive or transmit more than N packets in a single cycle.

In addition, by sharing the available memory space among all outputs better memory efficiency can be achieved, which has been demonstrated in [Eckberg88, Hluchyj88b, Iliadis91b, Liew94]. These results have shown that much improved packet-loss rates can be achieved by sharing a given amount of total buffer space, because a packet will only be lost when the entire shared buffer space is full [Eckberg88, Hluchyj88b]. Additionally, in [Iliadis91b] it is shown analytically that, given a combined input- and output-queued switch system employing back-pressure, a fully shared output buffer of size Nb offers superior throughput compared to N dedicated output buffers of size b each, see also Section 2.7.3.

Shared-memory implementation

Although a shared-memory implementation reduces the aggregate memory bandwidth to $2NB$ compared to dedicated output queuing, the implementation of the shared memory itself is the greatest challenge in terms of implementation. The aggregate bandwidth requirement equals that of an input-queued switch with identical throughput, but in the input-queued switch the memory is distributed over N inputs, which typically are also separated physically (on separate chips and boards), whereas in the shared-memory switch the entire bandwidth is funneled through a single memory. This is the main drawback of the shared-memory architecture. To obtain the desired bandwidth, several approaches are possible, for example employing wide memory, or employing parallel (interleaved) memory banks, pipelining memory access, or combinations thereof, as illustrated by the following examples of such switches.

Hitachi switch

Hitachi [Kuwahara89, Kozaki91] developed a 32×32 shared-buffer ATM switch, with a port speed of 155.52 Mb/s (OC-3). The shared buffer is organized in the form of a bit-slice with eight memories; the datapath to each memory is 54 bits wide (54 bits times 8 memories yields the internal 54 byte format); the cycle time of each memory is 38.6 ns, and it can store 4096 packets. Support for traffic classes is provided in the form of two delay classes and two loss rate classes, whereas multicast requires additional circuitry. The 32×32 switch is implemented in a set of 0.8 μm CMOS VLSIs on one printed circuit board.

The switch is intended for use as a building block for large multistage fabrics (up to 1024 ports), but owing to the complete lack of flow control, performance in such a system will be uncontrollable and therefore unpredictable.

Vulcan switch

In [Stunkel95], the high-performance switch architecture used to interconnect parallel processors in the IBM SP2 supercomputers is described. The heart of the switching network is the Vulcan switch, see Fig. 2.9.

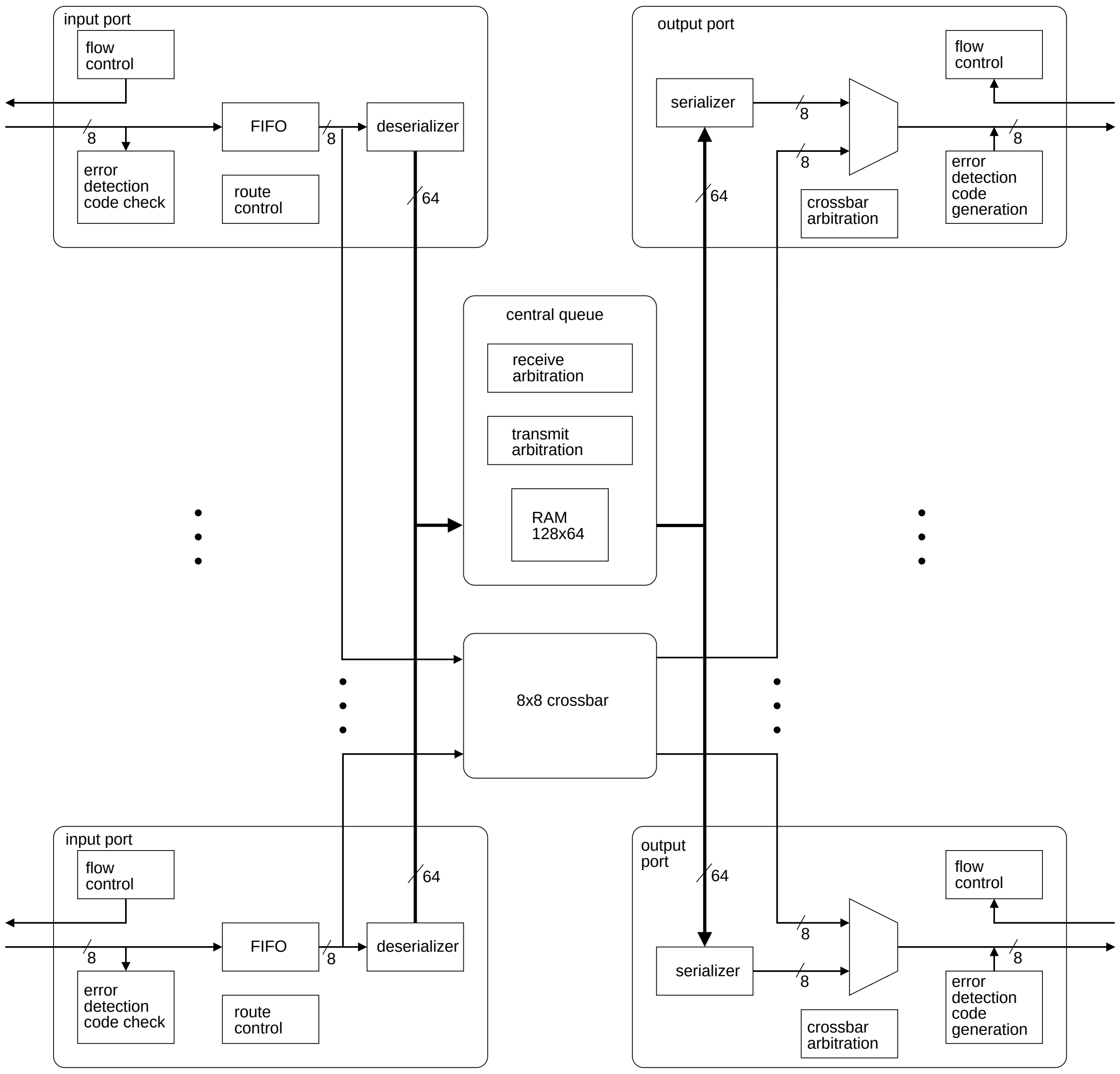


Figure 2.9: Vulcan switch architecture, from [Stunkel95].

It is an 8×8 output-queued switch, with an aggregate throughput of 2.56 Gb/s. In parallel computing applications low latency is very important, therefore the shared memory is augmented by a crossbar to achieve low latency for packets that experience no contention. Packets that experience contention are stored in a *central queue*, implemented in a 1 kB large dual-ported memory. The central queue maintains eight lists, one for each output. Space for each of these lists is allocated dynamically upon request, without per-input or per-output limitation.

Output buffering is achieved by means of pipelining: the datapath through the central queue is

64 bit wide, enough to transport eight data units (called *flits* in SP2 terminology) simultaneously. To achieve the desired bandwidth, each input port that wants to write to the central queue must deserialize eight flits and write them at once. To this end, each Vulcan input port has a FIFO buffer. At the output side, these flits have to be serialized again.

To prevent data loss, the Vulcan switch employs credit-based (actually called *tokens*), point-to-point flow control, enabled by the presence of the input buffers on each input port. This scheme enables gapless transmission on long links and allows Vulcans to be cascaded directly together, which is a very useful property when building the large switch systems the Vulcan is intended for. The credit protocol is very simple: each output port starts with a number of credits (31) and removes one credit for each flit transmitted. It can keep transmitting as long as the credit counter is greater than zero. The receiving input buffer returns one credit to the sender for every flit that leaves the input buffer.

ATLAS I switch

The ATLAS I switch [Katevenis96a, Katevenis96b, Katevenis97, Katevenis98, Kornaros99], developed by FORTH and the University of Crete, is a 16×16 single-chip output-queued shared-memory switch with a port speed of 622 Mb/s. The memory is implemented using parallel memory banks that are accessed in a pipelined fashion. This scheme is described in detail in [Katevenis95], and was used earlier in the Telegraphos I, II, and III switches. It supports three service classes, multicast, and link bundling. Its most distinguishing feature is the *credit-based* flow control, which enables the modular construction of large multistage networks while maintaining high performance [Katevenis96b].

PRIZMA switch

An example of shared output queuing is encountered in the PRIZMA switch [Denzel95]. This concept is represented logically in Fig. 2.8, whereas Fig. 2.10 shows the physical implementation. Here, all packets are stored in a central, shared memory having a size of M packets, and the output queues only contain addresses that point to the corresponding packets in the shared memory. Essentially, the control- and data-paths are thus separated, a crucial feature of PRIZMA, as shall be explained shortly.

The original PRIZMA architecture is based on the separation of the data and control paths. The packet payload data is routed through the data path only. It consists of the shared memory and fully parallel input and output routing trees (one routing tree per input and one per output) that allow parallel, simultaneous access to any set of N different memory locations on both the read and the write side. This arrangement overcomes the bottleneck of a shared medium such as a bus for the data stream.

The data section is controlled by a control section to achieve the desired switching function. It consists of N output queues, one free address queue, and demultiplexers to route incoming addresses to the correct output queues. The free queue initially contains all shared memory addresses. It provides every input with a memory address to store the next incoming packet at. When a packet arrives, the corresponding input stores it at the designated memory address, forwards the address to the control section along with the packet's header, which contains destination and priority information. The control section routes the address to one or more output

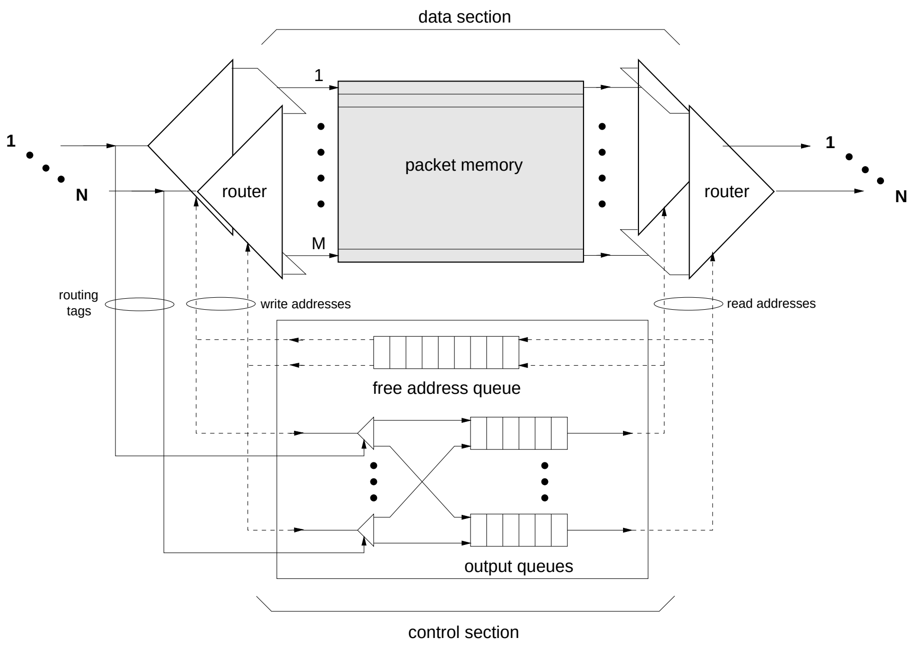


Figure 2.10: Shared output queuing, as implemented in the PRIZMA switch, [Denzel95].

queues, according to its destination, where it is appended at the tail of the queue(s). The input then requests a new address from the free queue, and will receive one, if present.[7] As there are N parallel input routers, this process can be executed in parallel for all inputs.

Each output queue removes the address at its head and uses this address to configure the corresponding output router, which will fetch the data from the memory at the requested address. When completed, the address is returned to the free queue, to be used again for newly arriving packets. Because of the parallel output routers all output queues can transmit simultaneously.

By means of output-queue thresholds queues can be prevented from using up an unreasonable portion of the shared memory, which could lead to performance degradation on other outputs. The use of multiple “nested” output-queue thresholds has been studied in [Minkenberg96].

In the PRIZMA architecture, the bandwidth burden is shifted onto the control section, but because the control section only handles addresses (pointers to packets in the shared memory), the overall bandwidth requirement is much less stringent. Given that the size of a pointer is $\log_2 M$ bits, the aggregate bandwidth requirement on the dedicated output queues that store the addresses equals $N(N+1)B\frac{\log_2 M}{L}$, with L being equal to the size (in bits) of one packet. To illustrate how the separation of control and data paths relaxes the overall bandwidth requirement, consider that for a typical packet switch with $N = 32$, $M = 1024$, $L = 64$ bytes $= 512$ bits, this constitutes a bandwidth reduction by a factor of more than 50.

Additionally, multicast can easily be supported with this architecture [Engbersen92]. An incoming multicast packet is stored only once in memory, whereas its address is duplicated to each destination output queue. An occupancy counter associated with each memory address keeps track of how many copies still have to be transmitted; this counter is initialized to the number of destinations (the multicast fanout). Once it reaches zero, the address is freed and

[7]If no free addresses are available, the next incoming packet on this input will be lost. This situation should be avoided by employing proper flow-control mechanisms.

returned to the free queue.

The PRIZMA architecture also allows multiple, identical switch chips to be operated in parallel, thus multiplying the throughput by effectively widening the datapath, see Section 6.3.1. Flow control, priority support, expansion modes, etc. will be treated elsewhere, see Chapters 3 and 6.

2.6 Virtual Output Queuing

The afore-mentioned limitations of output-queued and shared-memory switches, in particular the limited scalability due to memory bandwidth constraints, rekindled interest in input-queued, crossbar-based architectures. The performance limitation caused by HoL blocking in FIFO input queues has been reduced by schemes such as those described in Section 2.3.2. Several of these schemes attempt to improve performance by searching the input buffer for packets destined to idle outputs. As already noted in Section 2.3.2, this search would be much more efficient if the input buffers were organized according to outputs.

This arrangement is called *virtual output queuing* (VOQ). The name refers to the sorting of packets per output in each input buffer, i.e., at each input a separate queue is maintained for each output, see Fig. 2.11. The first work describing an input-buffered switch using this arrangement, to my knowledge, dates back to a 1984 patent by McMillen [McMillen84]. Another early reference describing a switch system using this queuing arrangement is [Mehmet89]. The McMillen patent describes the VOQ mechanism, but calls it *multiple queuing*.

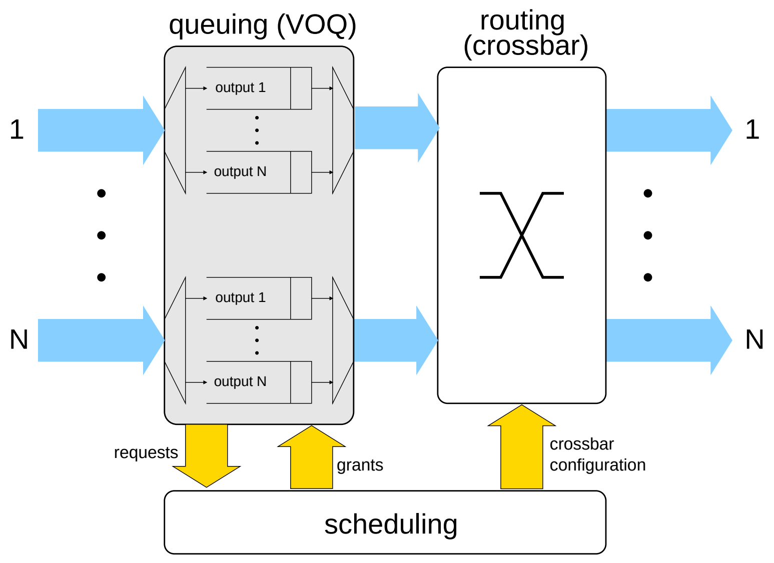


Figure 2.11: The VOQ architecture. There are N^2 VOQs at the input side, and the routing fabric is a simple crossbar. In each cycle, the scheduling unit collects the requests, executes some algorithm to decide which VOQs may forward a packet, returns the grants and configures the crossbar. The granted VOQs remove their HoL packets and forward them to the outputs.

This arrangement is sometimes also referred to as *advanced input queuing* [LaMaire94], but henceforth the term *virtual output queuing* will be used exclusively.

Although VOQ means maintaining N^2 queues in total, each individual queue can remain FIFO. This arrangement is comparable to output queuing in that packets are queued per output. However, the output queues are distributed over the inputs. This approach introduces a new problem: as there are now (a maximum of) N^2 HoL packets instead of just N, the problem of selecting which packets to transmit becomes much more complex. The performance of this architecture is determined by the arbitration algorithm.

Both conventional input-queued switches (see Section 2.3) and VOQ switches based on crossbars can only transmit one packet from each input and receive one packet at each input in one cycle ($S_i = S_o = 1$). The key difference between the two is that whereas the former has only one packet per input eligible for transmission, an input with VOQ can dispatch as much as N requests (one per output). The scheduler has to solve the problem of matching the inputs with the outputs in an optimal and fair manner. The remainder of this section will give a comprehensive overview of the techniques and algorithms proposed so far, and evaluate them in terms of performance, fairness, and implementational complexity. Fig. 2.12 shows a taxonomy of matching algorithms for VOQ switches.

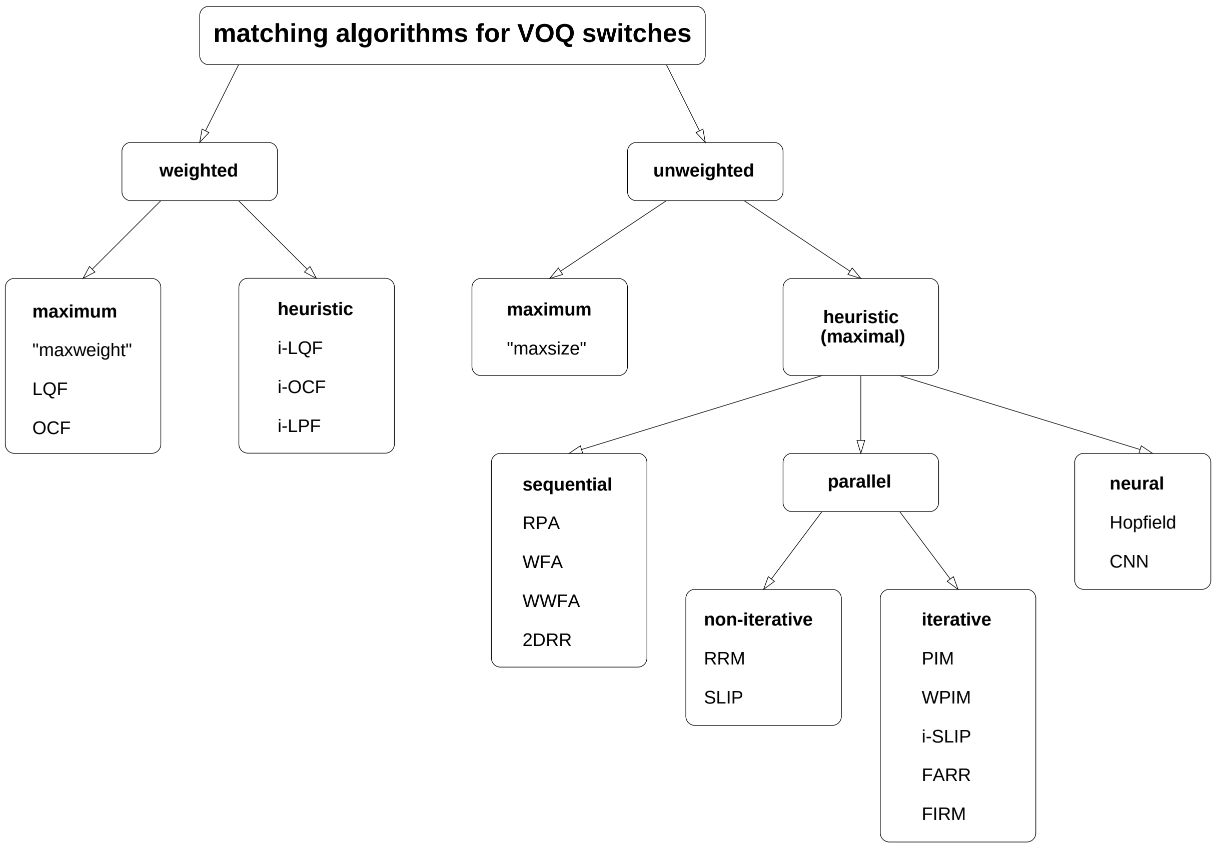


Figure 2.12: A taxonomy of matching algorithms for VOQ switches.

Note that the VOQ mechanism need not be employed with input-buffered or crossbar-based switching fabrics exclusively. The 1984 McMillen patent for instance describes a system where up to N packets can be selected from an input simultaneously, so that centralized arbitration is not required; the cost of this is that each input must provide a read bandwidth of N times the line rate, which poses a similar limitation as faced by an output-queued switch. That the VOQ mechanism is also useful in combination with output-queued switches is one of the main topics of this dissertation, and will be demonstrated in Chapter 3.

2.6.1 Bipartite Graph Matchings

The scheduling problem faced by crossbar-based, VOQ switches can be mapped to a bipartite graph-matching problem, see Fig. 2.13a. Assuming an $N \times N$ switch, the VOQ occupancies at time t are given by $Q_{ij}(t)$, with $1 \leq i, j \leq N$. In each packet cycle, we construct a graph $G = (V, E)$ that consists of a set V of $2N$ vertices, partitioned into two sets, namely N inputs (on the left) and N outputs (on the right). The set of edges E has one edge connecting vertex i of the inputs to vertex j of the outputs for each $Q_{ij}(t) > 0$, that is, for each non-empty VOQ the graph has a corresponding edge.

A matching M on this graph G is any subset of E such that no two edges in M have a common vertex. The practical interpretation of the matching is that for each edge (i, j) that is part of the matching one packet is transferred from input i to output j. The definition of the matching guarantees that only one packet per input and output needs to be transferred, thus satisfying the bandwidth restrictions imposed by the crossbar.

A distinction is made between *maximum* and *maximal* matchings. A maximum matching is the largest size matching that can be made on a given graph. A maximal matching is a matching to which no further edges can be added without first removing an already matched edge. Therefore, any maximum match is also a maximal match, but not vice versa. On a given graph, there can be multiple maximum and multiple maximal matches. Figs. 2.13b and 2.13c illustrate the difference.

Tarjan explains in [Tarjan83, Chapter 9] how a bipartite graph-matching problem can be translated into an equivalent network flow problem.

The bipartite graph-matching problem can naturally be mapped to a *request matrix* $R = [r_{ij}], 1 \leq i, j \leq N$, where $r_{ij} = 1$ if there is an edge between input i and output j in the graph, 0 otherwise. Solving the matching problem now boils down to finding a subset M of the set of requests $\{r_{ij}\}$ that satisfies conditions 2.3 and 2.4, which express that at most one request can be selected per row and per column.

$$(\forall i : 0 \leq i < N : \sum_{(i,j) \in M} r_{ij} \leq 1) \tag{2.3}$$

$$(\forall j : 0 \leq j < N : \sum_{(i,j) \in M} r_{ij} \leq 1) \tag{2.4}$$

Eq. 2.5 shows the request matrix R and the maximum and maximal matching matrices corresponding to Fig. 2.13.

$$R = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix} \quad M_{maximum} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \quad M_{maximal} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{2.5}$$

2.6.2 Maximum Size Matching

A maximum size matching M on the graph G described above is one that maximizes the number of edges in M. The fastest known solution to the maximum size matching problem executes in $O(N^{5/2})$ time [Dinic70, Hopcroft73]. In [McKeown95] a proof is given that maximum size matchings can achieve 100% throughput under uniform, identical and independently distributed (i.i.d.) Bernoulli traffic. However, it has been shown in [McKeown96a] that this no longer holds under non-uniform traffic, and moreover, that maximum size matchings can lead to instability and unfairness under admissible traffic, whereas under inadmissible traffic they can even lead to starvation. Therefore, maximum size matchings must be considered unsuitable.

2.6.3 Maximum Weight Matching

By assigning weights $w_{ij}(t)$ to the edges of G we can perform maximum weight matching on G, by finding a matching M that maximizes the total weight $W(t) = \sum_{(i,j)\in M} w_{ij}(t)$. Maximum size matching clearly is just a special case of maximum weight matching with all weights $w_{ij}(t) = 1$. The weights are a function of time t, as the state of the system variables from which the weights are computed change over time.

Tassiulas and Ephremides [Tassiulas92, Tassiulas98] showed, initially in the context of multi-hop packet radio networks, that maximum throughput is achieved if a maximum weight policy is adopted using the queue occupancies as weights, i.e., $w_{ij}(t) = Q_{ij}(t)$. McKeown et al. [McKeown96a] arrived at the same conclusion with their *longest queue first* (LQF) algorithm, which is proven to be stable under the condition of admissible, i.i.d. arrivals. However, LQF can be shown to lead to starvation of short queues. Mekkittikul and McKeown proposed the *oldest cell first* (OCF) algorithm in [Mekkittikul96], which uses the age of the HoL packets as weights, thus guaranteeing that no queue will remain unserved indefinitely, because the weight of unserved HoL packets will increase until they eventually are served. This algorithm is also shown to be stable under all i.i.d. arrivals.

The most efficient known algorithm that solves the maximum weight optimization problem requires $O(N^3 \log(N))$ execution time [Tarjan83, Chapter 8], which must be considered infeasible to implement at high port speeds. Therefore, the proposed algorithms are not practical to implement. Tassiulas proposed [Tassiulas98] a randomized algorithm of linear complexity that employs an incremental updating rule, using the heuristic that the matching obtained for the previous cycle is likely to be a good matching also for the current cycle, as the state of the input queues changes only slightly. Basically, in each cycle a random matching (distributed such that there is a non-zero probability that the optimum, maximum weight matching is chosen) is selected. If the new matching leads to a higher weight than the previous one, the new one is taken, otherwise the old one is kept. This approach is shown to achieve maximum throughput [Tassiulas98].

McKeown proposed [McKeown95] practically implementable heuristic, iterative versions of LQF and OCF, called i-LQF and i-OCF. These algorithms operate very similar to i-SLIP, with the difference that instead of just one-bit requests, each request carries a weight, equal to the queue length in case of i-LQF and equal to the age of the HoL packet in case of i-OCF. The grant and accept arbiters always select the requests c.q. grants with the largest weight (instead of the RR operation of i-SLIP).

Mekkittikul and McKeown [Mekkittikul98] designed the *longest port first* (LPF) algorithm to overcome the complexity problems of LQF. The weights are equal to the *port occupancy*, which is defined as the sum of the queue occupancies of the corresponding input and output: $w_{ij}(t) = \sum_k Q_{ik}(t) + \sum_l Q_{lj}(t)$. The algorithm is shown to be stable under all i.i.d. arrivals, and has a complexity of $O(N^{2.5})$, which is lower than LQF because LPF is based on maximum *size* matching with a preference to choose the maximum size matching with the largest weight (which could be less than the actual maximum weight matching). A heuristic, iterative version called i-LPF is proposed for implementation in high-speed hardware.

Finally, the i.i.d. arrivals assumption required by [McKeown96a, McKeown99c, Tassiulas92] has been lifted by Dai and Prabhakar, who proved using fluid model techniques that an IQ switch using a maximum weight matching algorithm achieves a throughput of 100%, as long as no input or output is oversubscribed and the input arrivals satisfy the strong law of large numbers, which are very mild conditions that most real traffic patterns satisfy [Dai00].

2.6.4 Heuristic Matchings

As both maximum size and weight algorithms are too complex and therefore unsuitable for application in high-speed packet switches, research has focused on heuristic matching algorithms that execute in linear time. In practice, these algorithm will converge on a *maximal* instead of maximum match, a maximal match being defined as a match to which no edges can be added without first removing previously matched edges. Fig. 2.13 illustrates the difference between a maximum and a maximal match.

Many heuristic matchings operate iteratively, adding edges to the match in each iteration without removing existing ones. If they iterate until no more edges can be added, a maximal match has been achieved, by the above definition.

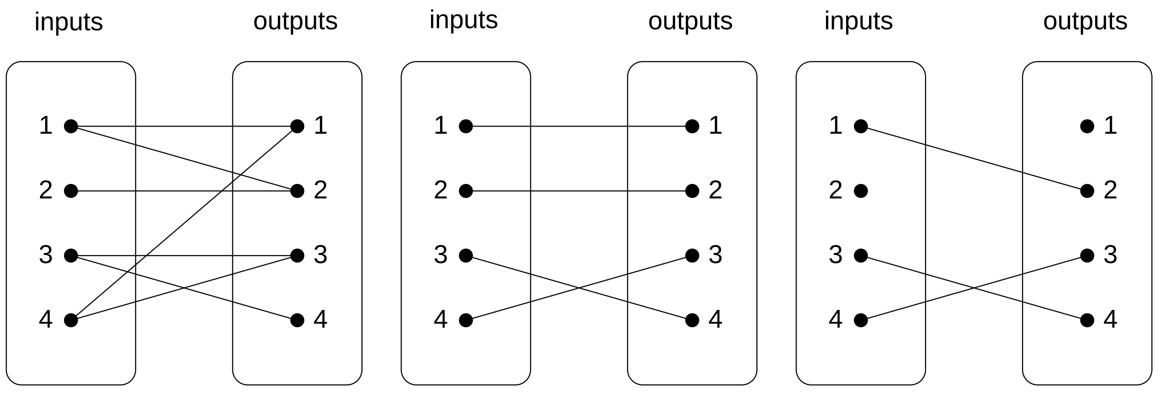
These heuristic algorithms can be divided into three subclasses: *sequential*, *parallel*, and *neural* algorithms. Sequential matching algorithms match one, or a limited number of input and output pair(s) at a time and require $O(N)$ iterations, whereas parallel algorithms can match up to N inputs and outputs simultaneously and require $O(\log N)$ iterations. Parallel algorithms are typically employed iteratively to obtain better performance. Neural algorithms employ a neural network to compute a matching. Subsequently, we will review algorithms from all three classes.

Sequential Matching

The most straightforward way to obtain a maximal match is by sequential matching. This algorithm works as follows: Initially, all inputs and outputs are unmatched. Every input sends a request to each output for which it has at least one packet. The algorithm iterates over all outputs; in each iteration one output gives a grant to one of the inputs it received a request from, excluding inputs that have been already been matched previously.

To ensure fairness and avoid starvation, each output provides grants in a round robin fashion: it keeps a pointer to the input it has granted to last and will give the next grant to the input that appears next in the round robin. Similarly, the sequence in which the outputs are matched is determined in a round-robin fashion.

The main drawback of this algorithm is that although it obtains a maximal match, it requires



(a) Bipartite graph. (b) Maximum match: four connections. (c) Maximal match: only three connections; no more can be added without removing one first.

Figure 2.13: Maximal vs. maximum match on a bipartite graph. Heuristic matching algorithms typically converge on maximal matches, which may lead to lower throughput, as illustrated by (b) vs. (c).

$O(N^2)$ execution time. To improve and achieve linear execution time, parallel matching algorithms have been developed.

In [Tao94] (see also Section 2.5.1) an algorithm for input-buffered switches is described that can be classified as a sequential matching algorithm, although it does not explicitly employ VOQ.

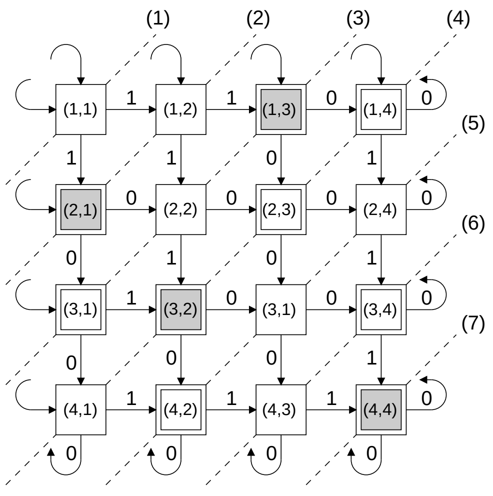
In [Ajmone97, Ajmone98] RPA is presented, a sequential matching algorithm that operates as described above, but with an extension to support multiple traffic classes. The next two sections describe other sequential matching algorithms, namely WFA, WWFA, and 2DRR.

WFA: Wave Front Arbiter

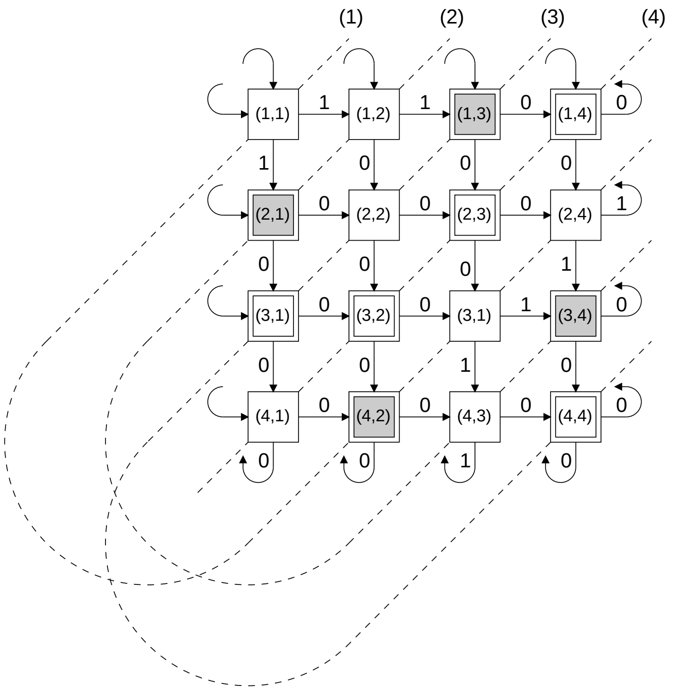
Tamir and Chi [Tamir93] designed a sequential matching algorithm that has better time complexity than the algorithm described, based on the observation that, when looking at a request matrix R (see for instance Eq. 2.5), the matching decisions for requests that lie on the *same diagonal* of the matrix can be made in parallel, as they can never conflict because they all are in different rows and columns. Thus, all matrix diagonals are processed in sequence, excluding inputs and outputs that have already been matched from the process. The time complexity of this process is $2N - 1$, as there are as many diagonals. This arbiter is called the *Wave Front Arbiter* (WFA) because of the way the diagonal moves like a wave through the matrix, see Fig. 2.14a.

A further improvement can be made by observing that the diagonal can be wrapped (see Fig. 2.14b), so that each diagonal covers exactly N elements at a time. The resulting arbiter is called *Wrapped Wave Front Arbiter* (WWFA) and its time complexity is reduced to N. Additionally, it is highly amenable to high-speed implementation.

As an example, Eq. 2.6 shows the matchings resulting from WFA and WWFA, using the request



(a) Wave Front Arbiter.



(b) Wrapped Wave Front Arbiter.

Figure 2.14: Wave Front and Wrapped Wave Front Arbiters, from [Tamir93], showing an example arbitration. The double squares indicate requests, the shaded squares indicate grants.

matrix R which corresponds to Fig. 2.14. Note how both arrive at different matches, although they happen to be of equal size.

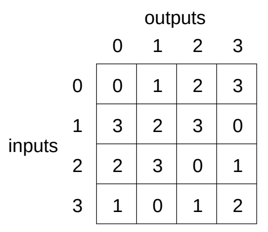
$$R = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix} \quad M_{WFA} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad M_{WWFA} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{bmatrix} \tag{2.6}$$

2DRR Scheduler

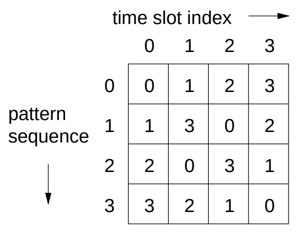
The Two-Dimensional Round-Robin (2DRR) scheduler presented in [LaMaire94] is a generalization of the WFA [Tamir93], although no reference is made to said work.

First, the term *generalized diagonal* is defined: A generalized diagonal is a set of N elements in an $N \times N$ matrix such that no two elements are in the same row or column. Thus, there are $N!$ different generalized diagonals in an $N \times N$ matrix. Note that the wrapped diagonals of the WWFA are also generalized diagonals.

Two matrices are defined, the Diagonal Pattern Matrix $\mathrm{DM}(i, j)$ and the Pattern Sequence Matrix $\mathrm{PM}(k, l)$ with $0 \leq i, j, k, l < N$, see Fig. 2.15. The diagonal pattern matrix defines the generalized diagonals used in the matching process, which is performed in exactly the same way as the WWFA. The pattern sequence matrix is intended to provide fairness among inputs and outputs: if the matching would start with the same diagonal in each time slot, certain connections would always be favored, leading to unfairness. The pattern sequence matrix alleviates this potential unfairness by permuting the sequence of diagonals to be applied each cycle. The authors show that when $N + 1$ is prime, the scheduling algorithm is fair, in that no index in the pattern sequencing matrix is treated preferentially with respect to any other. However, in case $N + 1$ is not prime, this guarantee cannot be made. To enhance the fairness properties of 2DRR, an enhanced version of the algorithm is developed, which uses N different diagonal pattern matrices in consecutive phases (each phase consisting of N timeslots).



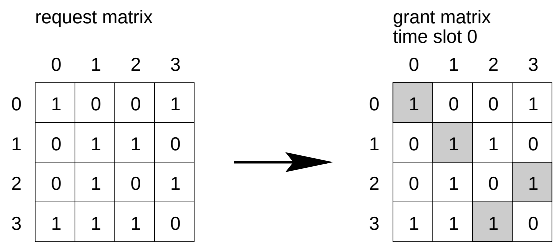
(a) Diagonal pattern matrix. Each set of N matrix entries with the same value constitutes one generalized diagonal. Note the similarity to the WWFA approach.



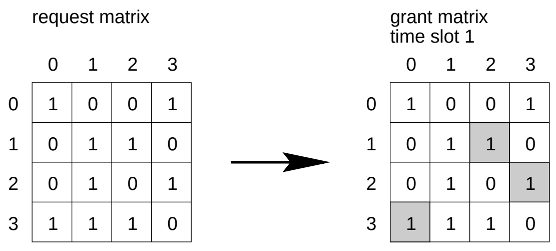
(b) Pattern sequence matrix. Each column defines the pattern sequence to be applied in that timeslot (modulo N). The matrix entries indicate the number of the generalized diagonal to be applied, as defined by the diagonal pattern matrix.

Figure 2.15: 2DRR scheduling algorithm, [LaMaire94]. Shown are (a) an example 4×4 diagonal pattern matrix and (b) a pattern sequence matrix used in the matching process.

Fig. 2.16 shows an example of a request matrix and the resulting grant matrices in time slots 0 and 1, using the DM and PM of Fig. 2.15.



(a) Time slot 0.



(b) Time slot 1.

Figure 2.16: Request and resulting grant matrices in time slots 0 and 1, corresponding to the patterns of Fig. 2.15. Grants are indicated by the shaded squares. Example taken from [LaMaire94].

Maximum throughput of both algorithms is shown to be 100%, and delay characteristics are obtained by means of simulation with i.i.d. Bernoulli traffic. Hardware implementation is shown to be straightforward and amenable to application in high-speed switches.

Parallel Matching: PIM, WPIM

The sequential algorithms described above require $O(N)$ iterations to arrive at a maximal match. A second class of heuristic matching algorithms allows multiple (up to N) input and output pairs to be matched simultaneously, thus converging faster on a maximal match. One such *parallel matching* algorithm is the Parallel Iterative Matching (PIM) algorithm invented by DEC [Anderson93]. Its operation is as follows: Initially, all inputs and outputs are unmatched. In each iteration of the algorithm, the following request, grant, and accept steps are executed:

1. *Request*: Each unmatched input i sends a request to every output j for which it has a packet ($Q_{ij}(t) > 0$).

2. *Grant*: If an unmatched output receives any requests, it *randomly,* in a uniform manner, selects one that will receive the grant.

3. *Accept*: If an input receives any grants, it *randomly,* in a uniform manner, selects one to accept.

As outputs do not coordinate their decisions in the grant step, multiple outputs may grant to the same input. These grant conflicts are resolved in the accept step, but every rejected grant implies a potential reduction in matching size. To increase the size of the matching, multiple iterations are executed, taking into account only the inputs and outputs that remain unmatched after the preceding iteration, adding connections until either no unmatched inputs and outputs remain, or a fixed number of iterations has been completed. In the worst case, it may take N iterations for the algorithm to converge, making PIM no faster than sequential matching. However, owing to the randomness in the algorithm, it can be shown that the algorithm completes in $O(\log N)$ iterations on average. Digital's AN2 prototype switch of size 16×16 uses four iterations fixed, instead of iterating until no more connections can be added [Anderson93].

Performance-wise, PIM is a definite improvement over FIFO queuing, and it executes faster than sequential scheduling. Moreover, it will never starve a connection like maximum size matching does, but it has its disadvantages: With a single iteration, throughput is limited to just 63%, and it may also cause unfairness among connections [McKeown95]. Furthermore, implementing random selections at high speeds is difficult and costly.

WPIM (Weighted PIM) [Stiliadis95] is a variant of PIM that adds support for providing bandwidth guarantees between individual input-output pairs in the presence of ill-behaved flows. Regular PIM always provides equal bandwidth to all input-output pairs because of its statistical nature. If an output is not oversubscribed, that is, the sum of arriving flows does not exceed the output's bandwidth, this is not a problem, but when one input exceeds its allocated share of bandwidth, all others suffer as well. To prevent this, WPIM introduces bandwidth-allocation coefficients that are used by each output to decide individually whether a particular input has exceeded its allocated share of bandwidth—those that have will not receive a grant as long as other inputs have not received their allocated share. Thus, WPIM enables bandwidth provisioning and flow isolation in an input-queued switch using VOQ. Like regular PIM, it completes in $O(\log N)$ iterations.

Round-robin matching: RRM, SLIP and i-SLIP, FARR, FIRM

In [McKeown95] the SLIP and i-SLIP algorithms are presented, which were designed to achieve an improvement in terms of efficiency, fairness and speed compared to existing algorithms. The SLIP algorithm is based on the Round-Robin Matching (RRM) algorithm, described next.

RRM is very similar to PIM, but instead of using random selections, it uses modulo N round-robin arbiters, one for each input and one for each output. Each arbiter maintains a pointer, indicating the element that currently has highest priority. RRM operates as follows:

1. *Request*: Each unmatched input sends a request to each output it has at least one packet for.
2. *Grant*: Each output that has received at least one request selects one request to grant by means of its round-robin arbiter: it chooses the input that appears next in the round robin, starting from the input currently being pointed to. The pointer is advanced (modulo N) to one beyond the input just granted.
3. *Accept*: Similarly, each input that has received at least one grant selects one grant to accept by means of its round-robin arbiter: it chooses the output that appears next in the round robin, starting from the output currently being pointed to. The pointer is advanced (modulo N) to one beyond the output just accepted.

Unfortunately, RRM does not perform very well. Under uniform i.i.d. Bernoulli arrivals, it saturates at an input load of merely 63%, close to that of PIM. The reason for this is that output arbiters tend to *synchronize*, causing multiple arbiters to grant to the same input, which leads to a waste of grants and thus poor throughput.

SLIP is an improvement on RRM, aimed at preventing synchronization of arbiters, hence the name (SLIP is not an acronym). Its operation is very similar to RRM, with only a subtle modification in step 2 of how the pointers are updated:

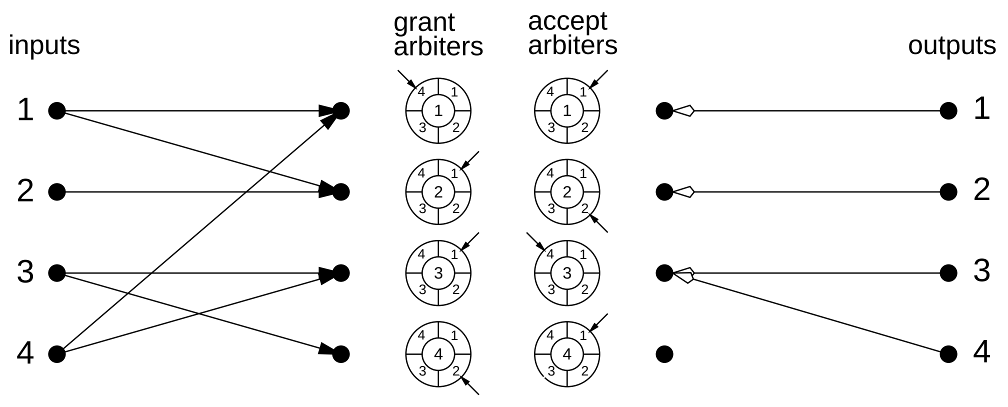
1. *Request*: Same as RRM.
2. *Grant*: Each output that has received at least one request selects one request to grant by means of its round-robin arbiter: it chooses the input that appears next in the round robin, starting from the input currently being pointed to. The pointer is advanced to one beyond the input just granted *if and only if the grant is accepted in step 3.*
3. *Accept*: Same as RRM.

The operation of the SLIP algorithm is illustrated for a 4×4 switch in Fig. 2.17.

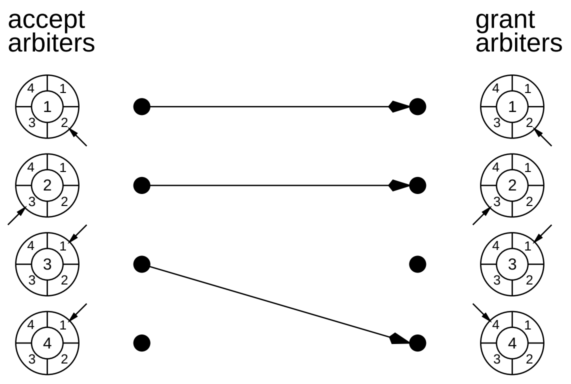
It turns out that SLIP indeed solves the performance limitations of PIM and RRM. Indeed, under uniform i.i.d. Bernoulli arrivals, a maximum throughput of close to 100% is achieved.

One can improve the performance of SLIP further by executing multiple iterations of the algorithm in a single packet cycle, just like PIM does. This variant, called *i-SLIP*, operates as follows:

1. *Request*: Each unmatched input sends a request to each output it has at least one packet for.



(a) Request (b) Grant



(c) Accept

Figure 2.17: One iteration of the SLIP algorithm: (a) Request: each input sends a request to every output it has a packet for, (b) Grant: each output gives a grant to the input that appears next in its grant round robin, (c) Accept: each input accepts the grant from the output that appears next in its accept round robin. All pointers are updated after completion of this step. Note how the grant pointer on output three is *not* updated, because its grant was not accepted.

2. *Grant*: Each output that has received at least one request selects one request to grant by means of its round-robin arbiter: it chooses the input that appears next in the round robin, starting from the input currently being pointed to. The pointer is advanced to one beyond the input just granted if and only if the grant is accepted in step 3 *of the first iteration*.

3. *Accept*: Similarly, each input that has received at least one grant selects one grant to accept by means of its round-robin arbiter: it chooses the output that appears next in the round robin, starting from the output currently being pointed to. The pointer is advanced (modulo N) to one beyond the output just accepted *only if this input was matched in the first iteration*.

Note that this is almost identical to non-iterative SLIP, with the exception of the added condition in steps 2 and 3: the pointers are only updated in the first iteration for reasons of fairness. Compared to SLIP, iterative SLIP improves performance further, and even more so when the number of iterations is increased. On average i-SLIP appears to converge in about $O(\log N)$

iterations, a result similar to PIM. For a detailed analysis of performance and properties of SLIP and i-SLIP, the reader is referred to [McKeown95] and [McKeown99a]. ESLIP [McKeown99b] is a modified version of i-SLIP used in Cisco's high-end switched backplane routers. ESLIP combines unicast and multicast scheduling into a single algorithm.

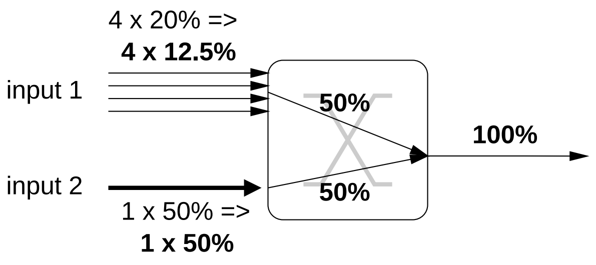
FARR

In [Lund96] the FARR (fair arbitrated round-robin) algorithm is presented. Like PIM and SLIP, FARR is a parallel, iterative matching algorithm. Its main goal is to achieve per-VC fairness, which is lacking from PIM and SLIP. Per-VC fairness is important in case an output is oversubscribed. In such a situation PIM and SLIP will evenly divide bandwidth over all inputs that have traffic for the output in question, which leads to unfairness in the sense of *max-min fairness* as defined in [Hahne91]. FARR overcomes this problem by employing time-stamped requests and favoring those VCs that were served least recently in the arbitration process.

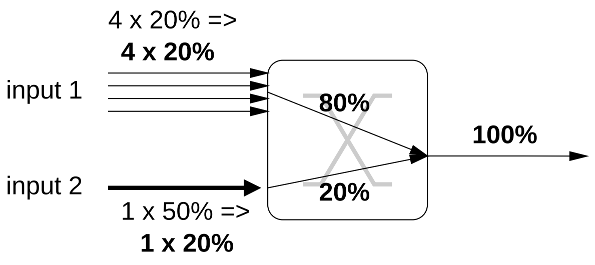
Max-min fairness requires that when K flows compete for a link, they should receive a bandwidth share equal to $\min(B/K, R_k)$ if the link is oversubscribed and R_k if otherwise, with R_k the bandwidth requested by flow k, and B the link bandwidth. This implies that as long as the link is undersubscribed, i.e., $\sum_k R_k \leq B$, every flow will receive its requested share of bandwidth, but when the link is oversubscribed, only those flows that exceed B/K are throttled.

PIM and SLIP do not satisfy this requirement, because they do not distinguish between different flows that arrive on the same input—for these algorithms each input is basically one aggregated flow, and bandwidth is distributed accordingly. That is, PIM and SLIP provide per-link fairness, but not per-flow fairness.

Fig. 2.18 shows an example: There are five flows in total for a given output, four from input 1, with a rate of 20% of the output link each, and one from input 2, with a rate of 50% of the output link. PIM and SLIP will assign 50% to each input, so that flows 1 through 4 each get 12.5% and flow 5 a full 50% (Fig. 2.18a), but according to the max-min fairness rule, each flow should get 20% here (2.18b). Note that this issue is not specific to SLIP or PIM, but to any switch implementing per-input rather than per-flow fairness.



(a) Unfairness: only the flows on the heavier-loaded input are impacted.



(b) Max-min fairness: each flow receives its minimum share of bandwidth.

Figure 2.18: Max-min fairness.

FIRM

In [Serpanos00] an improvement to SLIP is proposed. The difference lies in the way the grant pointers are updated:

1. *Request*: Same as SLIP.

2. *Grant*: Each output that has received at least one request selects one request to grant by means of its round-robin arbiter: it chooses the input that appears next in the round robin, starting from the input currently being pointed to. The pointer is advanced to one beyond the input just granted if and only if the grant is accepted in step 3. *If the grant is* not *accepted, the pointer is advanced to the granted input.*

3. *Accept*: Same as SLIP.

The authors show that this slight modification reduces the maximum waiting time for any request from $(N-1)^2+N^2$ for SLIP to N^2 for FIRM, resulting in improved delay characteristics at no extra implementation cost.

Randomization

Goudreau et al. [Goudreau00] proposed a randomization technique that can be used in conjunction with heuristic matching algorithms to improve their results. A randomization phase is inserted between each two consecutive iterations of the heuristic matching algorithm (such as PIM or SLIP). This randomization occurs as follows: Each unmatched input randomly selects an output it has at least one packet for. This input-output pair is added to the match, even if the selected output has already been matched, in which case the existing matched pair is removed from the matching. Ties are broken randomly in case multiple unmatched inputs select the same output. This approach is shown to lead to larger matchings and therefore improved performance in particular for non-uniform traffic patterns, which lead to low-degree graphs, i.e., graphs with only a few edges incident to each vertex. Such graphs generally have fewer optimal matchings than high-degree graphs; consequently, heuristic algorithms will more often result in a sub-optimal solution. The randomization procedure improves the quality of the solution by avoiding such local maxima in the solution space.

Neural-network matching

Memhet-Ali et al. [Mehmet89, Mehmet91, Chen94] took a neural network approach to the matching problem using Hopfield networks. Troudet and Walters [Troudet91] also proposed a Hopfield neural-network architecture. Fantacci et al. [Fantacci96] employed neural nets that belong to the class of *cellular neural networks* (CNNs).

The neural network that implements the matching consists of an array of $N \times N$ neurons n_{ij}, $1 \leq i, j \leq N$. The steady-state output V_{ij} of neuron n_{ij} can be either one (match) or zero (no match). The inputs I_{ij} to the neural network indicate whether or not input i has at least one packet for output j in which case $I_{ij} = 1$, otherwise $I_{ij} = 0$. In [Fantacci96] the following differential equation for the neuron state variables x_{ij} is proposed:

$$\dot{x_{ij}} = -x_{ij} + AV_{ij} - B \sum_{h=1,h\neq i}^{N} V_{ih} - B \sum_{k=1,k\neq j}^{N} V_{kj} + C. \tag{2.7}$$

The neuron output V_{ij} is derived from the neuron state x_{ij} through a piecewise-linear sigmoidal function $g(x_{ij})$:

$$V_{ij} = g(x_{ij}) = \begin{cases} 0, & x_{ij} < 0 \\ x_{ij}, & 0 \leq x_{ij} \leq 1 \\ 1, & x_{ij} > 1 \end{cases} \tag{2.8}$$

Note that Eq. 2.7 implies that each neuron only connects to neurons in the same row i and the same column j. This approach was already suggested in [Troudet91] in order to reduce the number of neural interconnections to $O(N^3)$ (compared to $O(N^4)$ for a fully interconnected net).

In [Fantacci96], the equilibrium point of the neural net is shown to be an optimum solution in the form of a permutation matrix if the constraints of Eq. 2.9 on the constant weights A, B and C in Eq. 2.7 are satisfied.

$$\begin{cases} A > 1 \\ B > C + A - 1 \\ C > 0 \end{cases} \tag{2.9}$$

Although the results are quite promising—delay–throughput performance is shown to be very close to ideal—the practical implementation of the proposed neural networks at the speeds envisioned does not seem feasible.

Stable Marriages

A discussion on bipartite graph matchings would not be complete without at least some mention of the *stable marriage problem*, first introduced by Gale and Shapley in 1962 [Gale62].[8]

The stable marriage problem is defined as follows: There are N men and N women. Each person lists all members of the opposite sex in their order of preference. A *matching* is any one-to-one pairing of men and women. A *stable* matching is a matching that cannot be upset by any pair not matched to each other but who prefer each other to their current partners. In that case, they would leave their current partners to become engaged, thus upsetting the (unstable) matching.

Gale and Shapley proved that there always exists at least one stable matching in any instance of the stable marriage problem by presenting an algorithm that always finds a stable matching. Furthermore, their algorithm exhibits the remarkable property that the matching found will always provide each man with the optimum partner he can obtain in any stable matching. The algorithm has a complexity of $O(N^2)$.

It has been suggested that the stable marriage problem may be applied in the domain of bipartite graph matching for VOQ switches [McKeown95, Stoica98], with the roles of men and women assumed by those of input and outputs, respectively. However, it is not intuitively clear

[8] A good reference book on the stable marriage problem is [Gusfield89].

how a stable marriage corresponds to a maximum size or maximum weight matching; the fact that each input c.q. output obtains an optimum partner does not mean that the overall result is optimal in terms of either size or weight. In the maximum size matching problem for VOQ switches, neither inputs nor outputs care about which 'partner' they are matched to. Furthermore, typically not every input has a packet for each output, implying that there may be holes in the inputs' preference lists. Outputs on the other hand are always open to receiving packets from all inputs, so that their preference lists have no holes. The preference lists can be arranged by, for instance, queue occupancies or service times. In their study of CIOQ switches that exactly mimick an OQ switch [Stoica98], Stoica and Zhang propose input preference lists based on arrival times (older packets have higher preference), and output preference lists based on the ideal reference OQ schedule.

```
assign each in- and output to be unmatched;
**while** some output $j$ that has not been rejected by all inputs is unmatched **do**
**begin**
    $i$ := first input on $j$'s preference list to which $j$ has not yet "proposed"
    **if** $i$ is unmatched **then**
        match $i$ and $j$ to each other
    **else**
        **if** $i$ prefers $j$ to its current match $j'$ **then**
            match $i$ and $j$ to each other and assign $j'$ to be unmatched
        **else**
            $i$ rejects $j$
**end**
```

Figure 2.19: The basic men-optimum Gale–Shapley algorithm, where outputs and inputs correspond to men and women respectively.

Note that the Gale–Shapley algorithm, which is of complexity $O(N^2)$, falls under the category of sequential matching algorithms because it matches one output at a time, and the outcome of subsequent matches depends on those already made. It will always converge on at least a maximal match.

2.6.5 An Implementation: Tiny Tera

The Tiny Tera [McKeown97a], developed at Stanford University, is a VOQ, crossbar-based packet switch, employing the i-SLIP algorithm presented in Section 2.6.4. It consists of a parallel sliced self-routing crossbar switch, a centralized scheduler to configure the crossbar, and 32 ports, each operating at 10 Gb/s, leading to an aggregate throughput (input+output bandwidth) of 320 Gb/s. It employs the i-SLIP algorithm to schedule unicast packets, whereas multicast scheduling algorithms based on the principles of fanout splitting and residue concentration (TATRA, WBA [Prabhakar95, Prabhakar96, Prabhakar97]) will be used to schedule multicast traffic (see also Chapter 4). A Tiny Tera chip-set, consisting of a scheduler chip, a number of crossbar chips, port processor chips, and data slicing chips, is also commercially available from PMC Sierra.

2.6.6 Variations

As a variation on VOQ it would be possible to implement the N queues at each input as a shared buffer, thus increasing buffer efficiency. Additionally, to reduce complexity at the input, the number of queues per input could be reduced by a factor G by grouping outputs together. This reduces complexity at the input at the expense of performance. To which degree the grouping factor G impacts performance remains to be studied.

2.7 Combined Input and Output Queuing

2.7.1 Introduction

So far, we have dealt with architectures that are either purely input- or output-queued. The third and final class of architectures combines the two concepts, and is commonly referred to as *combined input and output queuing* (CIOQ). Definition 4 formally defines the concept of CIOQ.

Definition 4 (Combined Input and Output Queuing) *A switch belongs to the class of* **combined input and output queuing** *switches if the queuing function is performed both* **before and after** *the routing function.*

Fig. 2.20 shows the logical structure of a CIOQ switch. We define the *input speed-up factor* S_i to be the number of packets that can be transmitted from a single input queue in one cycle, whereas the *output speed-up factor* S_o equals the number of packets that can be delivered to a single output queue in one cycle. S_i and S_o both can be between 1 and N. $S_i = 1$ and $S_o = 1$ is equivalent to classic input queuing (in which case the output queues are not needed), while $S_i = 1$ and $S_o = N$ is equivalent to classic output queuing. Given a particular speed-up S, the bandwidth of the input and/or output buffers must equal $S + 1$ times the link rate.

Several proposals for CIOQ architectures have been made over the years. Research has arrived at this hybrid solution from two opposite directions: either by realizing that adding output queues with a modest speed-up factor improves the performance of an input-queued switch, or by realizing that adding input queues to an output-queued switch improves its packet-loss characteristics.

As Fig. 2.20 illustrates, a CIOQ switch consists of three main components: the input queues, the interconnecting routing fabric, and the output queues. Accordingly, we classify CIOQ switch architectures based on the following criteria:

- Input queue organization: FIFO or non-FIFO, e.g. VOQ.
- Internal speed-up factor S: full speed-up, $S = N$, or partial speed-up, $1 < S < N$.
- Output buffer size: infinite or finite. In the former case, clearly not a practical configuration, $S < N$. In the latter case the output buffers can be dedicated or shared. Furthermore, the fabric may be internally lossy or lossless (using some form of fabric-internal flow control, e.g. back-pressure).

The remainder of this section reviews the various CIOQ architectures proposed in the literature.

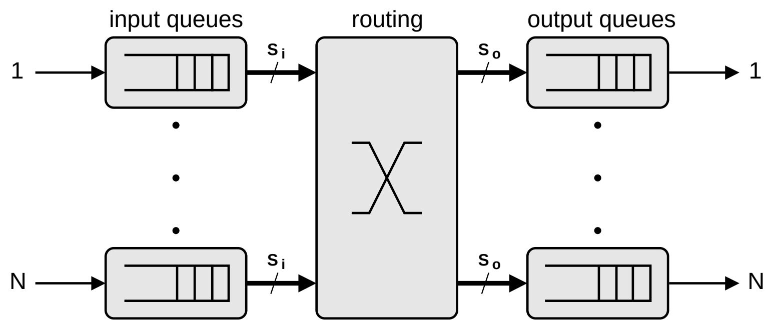


Figure 2.20: Combined input- and output-queuing, with input speed-up S_i and output speed-up S_o. The aggregate input and output bandwidth of the routing fabric equals $N(S_i + S_o)$ times the link rate.

2.7.2 CIOQ architectures with FIFO input queues

Oie et al. [Oie89] were among the first to study a CIOQ switch with limited speed-up ($S < N$; the speed-up S in this section refers to the output speed-up S_o, while $S_i = 1$), FIFO input queues and infinite output buffers. It is shown that for moderate speed-up large gains in maximum throughput can be achieved. Iliadis and Denzel analyzed the case with FIFO input buffers, finite dedicated output buffers using back-pressure to prevent packet loss at the output queues, and full speed-up in [Iliadis90a], and in [Iliadis91b] came to the conclusion that in such a configuration the best strategy is to share all available memory space among all output queues because this reduces the input queue HoL blocking probability (see also Section 2.7.3). Gupta and Georganas [Gupta91a] studied the case of finite dedicated output buffers with limited speed-up ($S < N$). They conclude that for a speed-up larger than three the performance-limiting factor is the HoL blocking at the input queues. Li [Li92] analyzed a CIOQ system with infinite output queues and a speed-up factor L under *correlated* traffic. Upper and lower bounds on the maximum throughput are derived for the cases of uncorrelated and strongly correlated traffic, respectively. The conclusions are that (a) traffic correlation influences maximum throughput negatively, but not to a large extent, and (b) the upper and lower bounds converge quickly to 1 for $S > 1$.

Pattavina and Bruzzi [Pattavina93] developed a general analytical model of a CIOQ system with limited input and output queues and a speed-up S. They distinguish between *queue loss* (QL) and *back-pressure* (BP) modes of operation, where the former may lead to packet loss at the output queues, whereas the latter prevents this by not allowing more packets to enter an output queue than the available space. Therefore, in the QL case, excess packets are dropped at the output queue, whereas in BP mode they have to wait in the input queues. Based on the analytical model and corresponding simulations, they arrive at the following conclusions:

- For both modes, a larger speed-up S leads to a higher maximum throughput. However, $S > 4$ brings little additional improvement.
- For a given output queue size, QL offers a higher maximum throughput than BP because less HoL blocking occurs in the input queues.
- For the BP mode, maximum throughput improves when the output buffer size is increased. Naturally, this also leads to a decrease in average delay.

- For the QL mode, on the other hand, increasing the output buffer does *not* improve maximum throughput, but leads to slightly higher average delay.
- A tradeoff between input and output buffer size exists when a total buffer budget is given to achieve the optimum packet-loss rate. For QL, more buffer space must be allocated to the output, whereas for BP more input buffer space is required. For utilization ≤ 0.8, BP requires the least amount of total buffer space.

Chang et al. [Chang94] also proposed a CIOQ switch with limited speed-up S based on a Batcher–Banyan network with r parallel distributing modules. The results of their analysis is that with speed-up $S = 4$ a performance close to output queuing can be achieved, assuming uniform, uncorrelated arrivals.

Lee and Ahn [Lee95] also analyzed a CIOQ system with limited in- and output queues and an arbitrary speed-up S, while adopting the queue loss scheme. Their main focus is to study the system under *non-uniform* traffic, which is known to cause performance degradation [Li90], [Chen91]. Packet-loss figures under uniform traffic are derived, indicating that for a given output queue size, an optimum speed-up S exists that yields the lowest loss rate. Increasing S further is not only more expensive implementation-wise, but will also increase the loss rate. As in [Chen91], an optimum distribution of a given buffer budget over in- and output queues is demonstrated. Regarding non-uniform traffic, it is concluded that the dominant non-uniformity is that of the output destination distribution. Based on these findings, an output buffer sharing strategy is proposed that improves loss performance by allocating more buffer space to queues experiencing higher load.

All these results have been obtained assuming $S_i = 1$ and $S_o = S$, i.e., the speed-up is implemented only at the output side.

This was noted by Guérin and Sivarajan in [Guerin97]. They collected evidence from previous publications that by lifting this rather arbitrary restriction and setting $S_i = S_o = S$, a speed-up of merely two is sufficient to obtain 100% throughput. They showed that for uniform traffic a speed-up factor of merely two results in a maximum throughput that approaches that of output queuing closely (no throughput limitation; up to 100%), for any burst size, provided the *saturation-stability property*[9] is satisfied.

Note also that using two parallel switching planes—although this might at first seem equivalent to a speed-up factor of two—*cannot* provide the same throughput because in this configuration it is not possible to switch two packets from the same input link. A way to overcome this problem was suggested in [Hui87], namely, by staggering the operation of one switch fabric by half a time slot with respect to the other.

Sunshine switch

The *Sunshine* switch [Giacopelli91], an effort undertaken at Bell Communications Research, is an example of a switch implementing CIOQ that also incorporates the concept of shared queuing. Basically, the Sunshine switch is an extension of the Starlite switch [Huang84], as

[9]From [Guerin97]: *"For specified distribution of packet destinations, we will say that an arrival process satisfies the Saturation-Stability Property if the input link queues are stable whenever the expected number of arrivals on each input link per switch slot, q, is less than the saturation throughput, γ, of the switch."* Note that $q = p/S$, with p being the average input load.

their architectures are quite similar, both being based on a Batcher–Banyan network with a trap network, a recirculating shared queue, and a concentrator. What sets it apart from the Starlite switch is the use of multiple, say K, Banyan networks in parallel, allowing it to switch K packets to a single output in one cycle. At the outputs the packets are gathered into output buffers. Thus, the Sunshine architecture achieves a degree of output queuing, thereby decreasing the rate at which packets are recirculated.

Increasing the number of parallel Banyan networks will greatly increase performance (both in terms of packet-loss rate and maximum throughput), up to a certain point where additional Banyans will no longer provide a significant additional performance improvement ($K = 8$ seems to be a good choice regardless of the switch size N).

2.7.3 CIOQ with finite output buffers and back-pressure

Iliadis and Iliadis and Denzel published a number of papers [Iliadis90a, Iliadis90b, Iliadis91a, Iliadis91b, Iliadis92a, Iliadis92b, Iliadis93] on the combined input- and output-queued switch architecture depicted in Fig. 2.21. We carefully review these results here because of their importance to the development of the CIOQ architecture in Chapter 3.

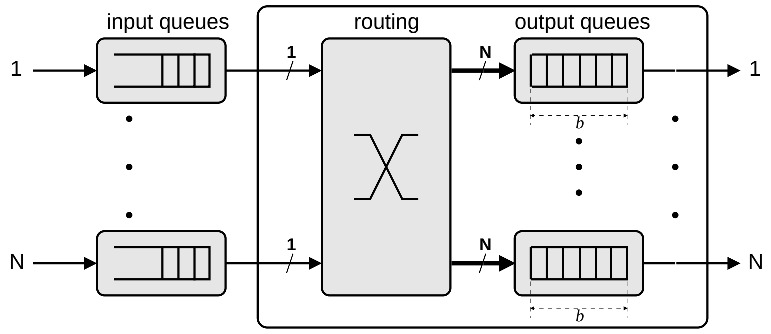


Figure 2.21: An $N \times N$ CIOQ switch with finite output buffers, input and output queuing.

The system consists of N output queues of size b each, N input queues and an interconnection structure between them to route the packets, with speed-up $S = \min(b, N)$. When an output queue becomes full, this is instantaneously flagged to the input queues (back-pressure mode, as previously described) so that no cells are lost owing to output-queue overflow. All queues, input and output, are always served in FIFO order. The impact on the performance of the HoL selection policy (FCFS,[10] LCFS,[11] Random) among the FIFO input queues is studied in [Iliadis90a, Iliadis91a]. It is found that maximum throughput does not depend on selection policy, but only on the output buffer size and packet size distribution. The FCFS policy is shown to give a lower bound on delay, whereas the LCFS policy leads to an upper bound. All results in this section have been derived under the assumption of uniform, independent arrivals (Bernoulli traffic).

In [Iliadis90b] the saturation throughput for given b is derived. Table 2.7.3 lists numerical values for a range of b. These results hold for large values of N. Note that the throughput increases

[10]FCFS: first come, first served.
[11]LCFS: last come, first served.

monotonously with b. For $b = 0$, that is, without any output queuing, the throughput equals 0.5857, while the limit for $b \to \infty$ equals 1, as had been established earlier in [Karol87].

The case where the available buffer space $N\alpha$ is fully shared among all output queues is analyzed in [Iliadis91b]. Eq. 2.10 expresses the maximum throughput $\lambda^\star_\alpha$ as a function of α, the number of buffers per output. $\lambda^\star_b$ is the maximum throughput for the dedicated output buffer (non-shared) case with b packets per output. Table 2.7.3 lists numerical results. Note that the limit $\alpha \to \infty$ again equals 1. Fig. 2.22 compares maximum throughput for the shared and non-shared cases as a function of α and b, respectively.

$$\lambda^\star_\alpha = \alpha + 2 - \sqrt{\alpha^2 + 2\alpha + 2}. \tag{2.10}$$

Table 2.1: Maximum throughput for various output buffer sizes with dedicated buffers, from [Iliadis93].

b	$\lambda^\star_b$	b	$\lambda^\star_b$	b	$\lambda^\star_b$	b	$\lambda^\star_b$
0	0.5857	5	0.8028	10	0.8601	15	0.8888
1	0.6713	6	0.8186	11	0.8672	16	0.8930
2	0.7228	7	0.8316	12	0.8735	·	·
3	0.7576	8	0.8426	13	0.8792	·	·
4	0.7831	9	0.8519	14	0.8842	∞	1.0000

Table 2.2: Maximum throughput for various output buffer sizes with shared memory, from [Iliadis91b].

α	$\lambda^\star_\alpha$	α	$\lambda^\star_\alpha$	α	$\lambda^\star_\alpha$
0	0.585	2	0.838	7	0.938
0.125	0.620	3	0.877	8	0.945
0.25	0.649	4	0.901	9	0.950
0.5	0.697	5	0.917	10	0.954
1	0.764	6	0.929	∞	1.000

Regarding the architecture of Fig. 2.21 we can conclude from these results that under the given traffic assumptions (a) more output buffer space improves throughput, and (b) sharing a given amount of total buffer space among all outputs also improves throughput.

The first conclusion is rather obvious, the second however not so. We will proceed by giving an intuitive explanation of both observations. First, we must realize why throughput is limited at all—this is clearly caused by the HoL blocking of the FIFO input buffers: When an output queue is full, the back-pressure mechanism blocks further packets from entering the output queue. Consequently, HoL blocking can occur on those inputs that have a HoL packet destined to a full output queue. This is essentially the same as the HoL blocking encountered in input-queued switches, but it only occurs when one or more output queues are full.

Throughput improves as the output buffer size increases because the HoL blocking probability described above decreases; with infinite output buffers, there is no HoL blocking, and therefore throughput equals 100%.

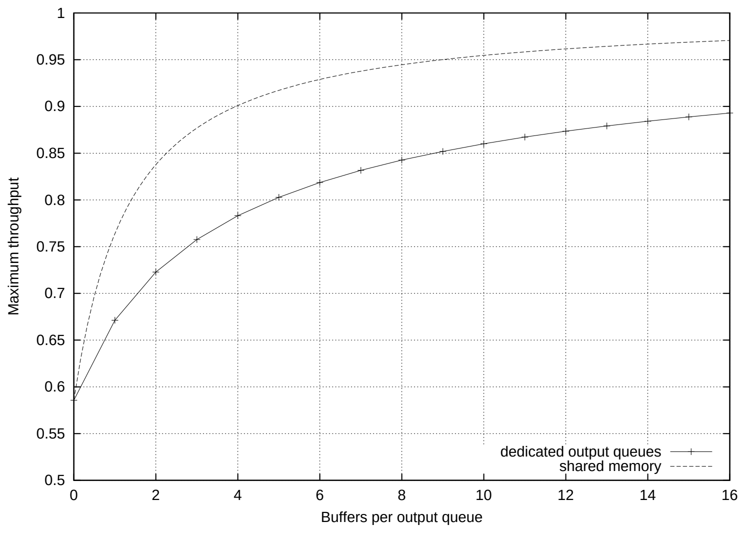


Figure 2.22: Maximum throughput vs. number of buffers per output for the shared and the non-shared case.

An argument why sharing the available buffer space also improves throughput can be constructed along the same lines: a packet will never be kept at the input FIFO HoL as long as there is at least one buffer available in the shared memory. Thus, HoL blocking due to output-queue-full conditions is eliminated, which improves throughput. Blocking only occurs when the entire shared memory is full. However, it must be mentioned explicitly that this result strongly depends on the traffic assumptions. In particular, for bursty or asymmetric arrivals, this result will no longer hold. In Chapter 3 we will investigate the behavior under bursty traffic, and look at the effects partial buffer sharing has on performance. Furthermore, a different arrangement of the input buffers will be proposed, and its effect on performance will be investigated.

For a purely output-buffered system (no input queues) with finite buffers, there is a tradeoff involved in sharing the memory: loss rate can be traded versus throughput. A system with dedicated output buffers will provide 100% throughput at a certain loss rate, whereas a system with a fully shared buffer will have a lower loss rate, but will simultaneously suffer throughput degradation owing to unfairness in case the shared memory is monopolized by only a few of the output queues (OQ lock-out). This is sometimes also referred to as resource *hogging*.

For a combined input- and output-buffered system with FIFO input queues, a similar trade-off exists. Full sharing eliminates HoL blocking at the FIFO input queues, leading to better throughput for a given total buffer size. However, as mentioned, full sharing also promotes OQ lock-out due to unfair monopolization of buffer space by overloaded outputs, which leads to throughput degradation. Because of the uniform Bernoulli traffic assumptions of the above analysis, however, this effect is not noticed. For highly correlated (bursty) or non-uniform (unbalanced) arrivals, the latter effect will indeed overshadow the former effect, so that dedicated buffers actually outperform the fully shared buffer. In cases like these, partial buffer sharing may turn out to be optimum.

2.7.4 Emulation of ideal output queuing

An interesting question is whether *any* input-queued or CIOQ switch with speed-up $S < N$ can provide the same performance as the ideal output-queued switch under *all* traffic patterns, assuming infinite queues. As the latter is work-conserving, any CIOQ switch that satisfies this requirement must also be work-conserving. In [McKeown97b] this question is partially answered: it is proven that a CIOQ switch having only a single FIFO queue per input (no VOQs) is *not* work-conserving *unless* it has a speed-up factor equal to N, which amounts to output queuing. In other words, such a switch with a speed-up factor $S < N$ cannot achieve the same performance as an ideal output-queued switch under all traffic patterns.

This answer triggered further research into CIOQ switches with VOQs instead of FIFOs that attempt to *exactly emulate* the behavior of the reference output-queued switch. Here, “exact emulation” means that, given the same input traffic patterns, the CIOQ switch exactly mimicks the packet departures at the outputs of the reference output-queued switch. This means that the departure times of all packets at the outputs of the CIOQ and the output-queued switch are identical. Note that this is actually a stronger requirement than merely having to be work-conserving: as the CIOQ switch is work-conserving this does not imply that exactly the same packets are departing from both switches, although packets depart from both switches at identical times. The significance in this research is that the speed-up S required to achieve this may be significantly smaller than N, given a smart enough centralized scheduling algorithm so that the traditional implementational bottleneck of output-queued switches can be overcome, while not losing any performance.

Recent publications [McKeown97b, Prabhakar99, Chuang99, Stoica98, Krishna99] have proposed CIOQ switches that achieve exact emulation of the ideal output-queued switch at a speed-up $S < N$ by employing VOQ and designing proper centralized scheduling algorithms. In [McKeown97b] it is proven (a) that a CIOQ with a single FIFO at each input can never be work-conserving *unless* it has a speed-up of N, and (b) that a speed-up of $N/2$ suffices if VOQ is applied and the Home Territory Algorithm (HTA) is used to perform the scheduling. In [Prabhakar99], an improved result is proven: using the MUCFA (Most Urgent Cell First Algorithm), a speed-up of four suffices to exactly mimick an output-queued switch, for any switch size N and any traffic pattern. Finally, in [Chuang99], it is shown that a speed-up of just $2 - \frac{1}{N}$ is both necessary and sufficient to precisely emulate an output-queued switch under the same conditions using the CCF (Critical Cells First) algorithm, which is based on the stable marriage problem, see Section 2.6.4. Independently, Stoica and Zhang obtained very similar results with their JPM (Joined Preferred Matching) algorithm [Stoica98], also based on stable matching. This is an important theoretical result; it implies that the fabric-internal bandwidth required to obtain the performance and delay control (QoS) advantages of an ideal output-queued switch is *not* proportional to the number of ports, as is the case with purely output-queued switches. However, the price to pay is a fairly complex scheduler, which so far does not seem feasible to implement. Krishna *et al.* [Krishna98, Krishna99] introduce a scheduling algorithm based on output occupancy, called LOOFA, which is also work-conserving at a speed-up of just two, yet is simpler to implement than CCF.

Charny et al. [Charny98a, Charny98b] have shown that a CIOQ switch using an arbitrary *maximal* matching algorithm can achieve 100% throughput if the speed-up $S > 4$, as long as no input or output is oversubscribed and the input traffic is leaky-bucket-constrained. Dai and Prabhakar [Dai00] obtained a significantly improved result using fluid model techniques, proving that a

CIOQ switch employing a *maximal* matching algorithm with a speed-up of 2 achieves 100% throughput, under the restrictions that no input or output is oversubscribed and that the input arrivals satisfy the strong law of large numbers.[12] This implies that, regardless of input traffic and switch size, the complexity of implementing maximum weight matching can be foregone by speeding the switch up by a factor of 2 and implementing a maximal matching algorithm (see Section 2.6.4).

2.8 Conclusions

Based on the overview given in this chapter, we summarize the potential of each of the three switch-architecture classes for application in future high-speed packet switches, considering performance as well as implementation aspects.

2.8.1 Input queuing

Looking at the volume of papers published on VOQ architectures and scheduling over the past few years, it becomes obvious that this approach has quickly gained widespread acceptance in the research community as the right approach for high-speed packet switches. The underlying reasoning is that line rates are so high that the shared-memory approach becomes impossible owing to a lack of sufficiently fast memories.

The VOQ approach remedies this situation, but poses other problems. In particular, the scheduling algorithms required to arbitrate the VOQs are quite complex. The ideal maximum weight matching algorithms are far too complex for implementation in fast hardware. Maximum size matching algorithms can lead to unfairness and starvation and are therefore unsuitable. Heuristic maximal matching algorithms overcome the complexity hurdle and lead to good, but not ideal, performance.

However, there are still a number of other drawbacks to purely input-queued architectures:

- Performance: Although the VOQ approach has greatly improved the performance achievable with an input-queued switch, there still is a fundamental performance disadvantage compared to an ideal output-queued switch owing to input contention.
- Arbitration latency: Even if a packet arrives for an uncontended destination, it must still wait for the completion of the entire request-grant-accept arbitration cycle before it can proceed.
- Support for multicast traffic: Although algorithms for multicast in a VOQ switch have been designed, the additional complexity involved is significant, and performance deteriorates under heavy multicast traffic. Chapter 4 will delve deeper into this issue.
- Support for traffic classes to provide QoS differentiation: Providing QoS guarantees is difficult in an input-queued switch because packets from all inputs must be scheduled

[12]With probability one, $\lim_{n\to\infty} \frac{A_{ij}(n)}{n} = \lambda_{ij}, \forall i, j$, where $A_{ij}(n)$ equals the cumulative arrivals at VOQ_i^j up to time n.

simultaneously, whereas in an output-queued switch only those for the same output need to be considered.

- Scalability: Although the VOQ architecture scales well in terms of buffer bandwidth, it does not scale well to large switch dimensions because of both the computational and the bandwidth burden on the centralized scheduler, which has to arbitrate among N^2 queues.

Implementation

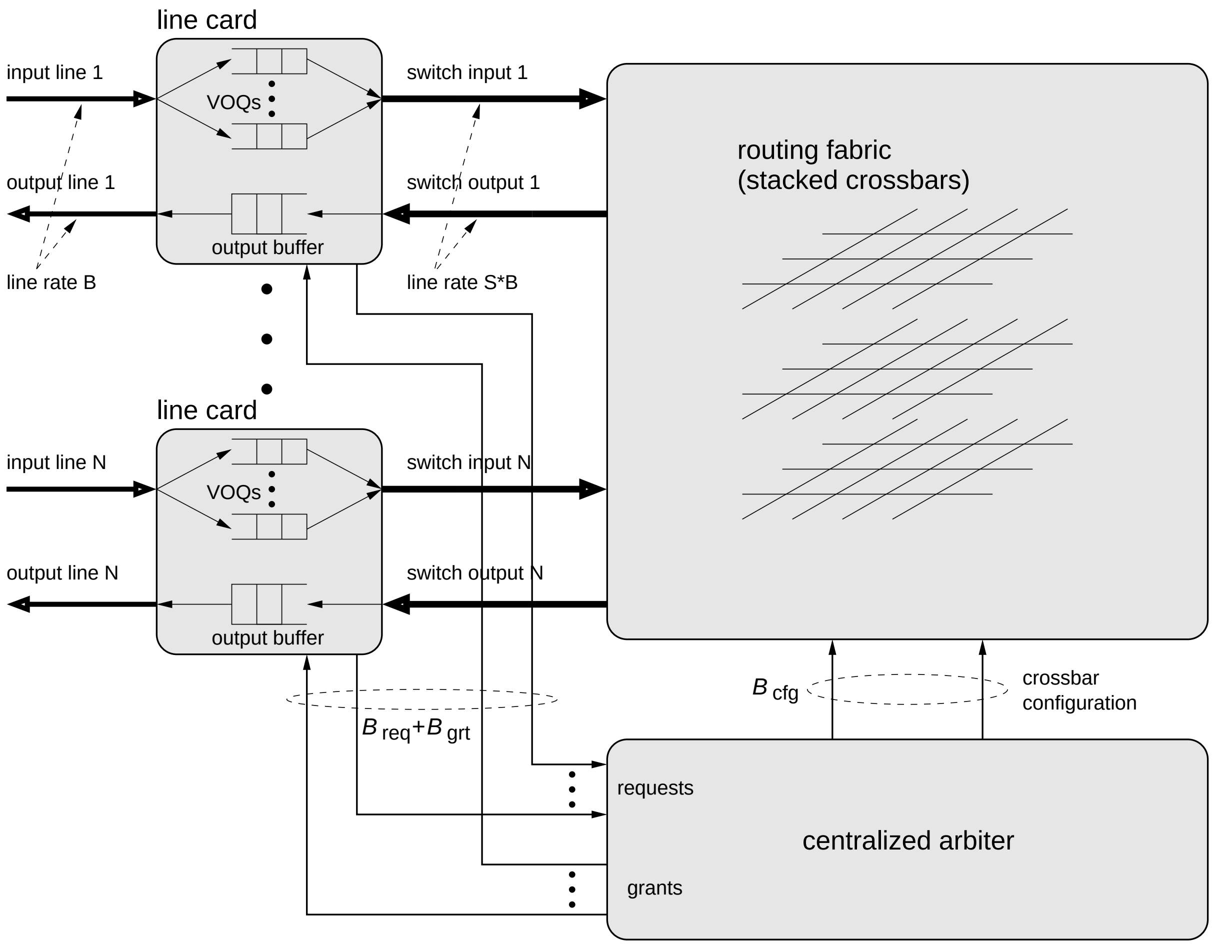


Figure 2.23: Implementation of a VOQ or CIOQ switch with limited speed-up.

Input-queued switches using VOQ are typically implemented as shown in Fig. 2.23. The VOQs and their associated buffer space reside on input line cards (adapters) that terminate the connecting lines. Input and output ports are usually paired together on the adapters, as shown. The arbitration unit implementing the scheduling algorithm is a separate chip located close to the switching fabric, which consists of one or more parallel crossbar chips. In each cycle, the arbitration unit must collect requests from N^2 VOQs, compute a matching, return N grants, and configure the crossbars. If the fabric runs at a speed-up S, the internal links, the routing fabric, and the arbiter must run S times faster than the external lines.

Two main disadvantages of this architecture are that although the bandwidth on any individual memory scales linearly with the port speed only, the bandwidth demand on the arbitration unit grows both linearly with the port speed B (in b/s) and quadratically with the number of ports N, because every input can submit up to N requests. If the switch has to support a minimum packet size of L bits, the aggregate request bandwidth equals $B_{\mathtt{req}} = (BN^2)/L$. The bandwidth required to return the grants equals $B_{\mathtt{grt}} = (NB \log N)/L$, and the bandwidth to configure the crossbar equals $B_{\mathtt{cfg}} = (NB \log N)/L$. Table 2.3 lists the resulting bandwidth values for a range of N, with $B = 10$ Gb/s, and $L = 64$ bytes $= 512$ bits.

Table 2.3: Comparison of aggregate request bandwidth to the arbitration unit for SLIP for a range of switch sizes and link rates B (10 and 40 Gb/s) for a minimum packet size of 64 bytes. TP = aggregate switch throughput. Bandwidths are given in Gb/s.

N	TP	$B_{\mathtt{req}}$	
		OC-192	OC-768
4	40	0.3125	1.25
8	80	1.25	5
16	160	5	20
32	320	20	80
64	640	80	320
128	1,280	320	1,280

As a remedy, the state (e.g., the occupancy count) of all VOQs can be kept in the scheduler, and only new arrivals are reported to the scheduler. For unicast traffic, this would require merely $N \log N$ bits per packet cycle because only a single packet can arrive. However, for multicast a full bitmap of N bits may have to be passed to the scheduler, so that in the worst case again N^2 bits of information must be passed unless the arbiter can do multicast lookups itself.

Furthermore, as N increases at constant B and L, the arbitration algorithm must choose from N^2 queues, in the same amount of time. For iterative algorithms such as i-SLIP, multiple iterations must be performed within a single packet cycle. As the port speed B increases, the packet cycle, and thus the time to arbitrate, become shorter. In the example mentioned, the packet cycle lasts 51.2 ns. With 4 to 5 iterations of the i-SLIP algorithm, this leaves on the order of 10 ns per iteration. To support the next generation of port speeds, e.g. OC-768 (40 Gb/s), the time in which one iteration completes must drop proportionally, to about 2.5 ns, which is a significant challenge in current CMOS technology.

Another problem is that as the system grows in terms of number of ports, it will also grow in physical size, and the links from input queues to arbitration unit will become long compared to the packet cycle: several request/packets may be in flight on these links. This implies that the scheduler is using outdated VOQ status information, which may impact performance.

2.8.2 Output queuing

Regarding purely output-queued switches, the verdict is clear: they are not suitable for future high-speed packet switches because of the memory bandwidth limitation. This holds equally for dedicated-output-queue and shared-memory architectures. Although very high memory throughputs can be achieved with highly parallelized implementations, such implementations are prohibitively expensive to implement and scale poorly. Such memories, which have to be implemented on the switch chip, are strictly limited in terms of size (in the range of 256 to 1 K packets), so that loss rates under bursty traffic conditions quickly become unacceptable.

2.8.3 Combined input and output queuing

Given the limitations of both purely input- and output-queued architectures, we must turn to combined architectures for a better solution.

There are two main alternatives: the crossbar-based architecture with a centralized arbiter and a speed-up of two, and the output-queued-switch-based architecture with full speed-up. Although the former architecture seems very promising because of its limited speed-up of two, regardless of switch size, the actual implementation of such a switch remains to be made practical. None of the ideal OQ-emulating algorithms surveyed in Section 2.7.4 has yet been shown to be amenable to implementation in VLSI at high data rates. The most practical result is that of Dai and Prabhakar [Dai00], who propose using a maximal matching algorithm and a speed-up of 2. Implementation-wise, this still compounds the problems of implementing a VOQ switch as described above by a factor of 2, i.e., for a given N, B, and L, the matching algorithm must operate twice as fast. In addition, both the bandwidth through the routing fabric and the input-buffer read bandwidth must also be doubled. Seeing that chip-IO and backplane bandwidth are the most costly resources, this approach is prohibitively expensive.

Therefore, we will explore the second alternative in this dissertation, taking as a starting point the architecture that combines FIFO input queues with a shared memory switch, as discussed in Section 2.7.3.

Chapter 3

A Combined Input/Output-Queued Packet Switch Architecture

In the preceding chapter we have argued that a CIOQ approach is the right one for a scalable high-speed packet switch, and we have identified significant drawbacks of existing CIOQ approaches. In this chapter we will develop an architecture that overcomes these drawbacks, evaluate its performance characteristics by means of simulation, and discuss its implementation.

3.1 Overview

The structure of this chapter is as follows: Section 3.2 introduces the architecture and explains its design rationale. Special attention is given to fabric-internal flow control, VOQ scheduling policies, and shared-memory size. In Section 3.3 we analyze system throughput under uniform traffic conditions. Section 3.4 presents performance simulation results of the proposed architecture under a wide range of traffic conditions and architecture parameters such as memory size, degree of memory sharing, input-queuing discipline, etc. Section 3.5 offers a way to improve performance by passing additional flow control information. Section 3.6 analyzes the implementational aspects. Finally, 3.7 concludes the chapter with a summary of the results.

Throughout this chapter, N refers to the packet switch dimension $N \times N$. This chapter deals with unicast traffic exclusively. Multicast traffic is the topic of Chapter 4.

3.2 Architecture

3.2.1 Design principles

The architecture proposed here is based on the following key observations:

1. Output buffers are expensive to implement because of their high bandwidth requirement. Therefore, their size should be kept to a minimum. Input buffers are significantly cheaper.
2. FIFO input queues are subject to HoL blocking. Therefore, VOQs should be used.

3. Centralized arbitration of VOQs does not scale because of its computational complexity. Therefore, distributed scheduling should be targeted.

4. Distributed scheduling implies that there is no coordination between VOQs on different inputs. Therefore, output contention must be resolved separately by providing some limited amount of output queuing.

5. Output buffer sharing has been shown to lead to improved buffer efficiency in existing switch architectures.

These observations naturally lead to the architecture depicted in Fig. 3.1, which combines input queues sorted per output (VOQs) with an output-buffered, shared-memory switch element. An arriving packet is stored in the VOQ corresponding to its destination port. The output-buffered switch enables distributed scheduling of the VOQs, so that each input arbiter can make a decision independent of all others. This selection process will be detailed below. The packet selected is then forwarded to the switch, which will route the packet to its destination(s).

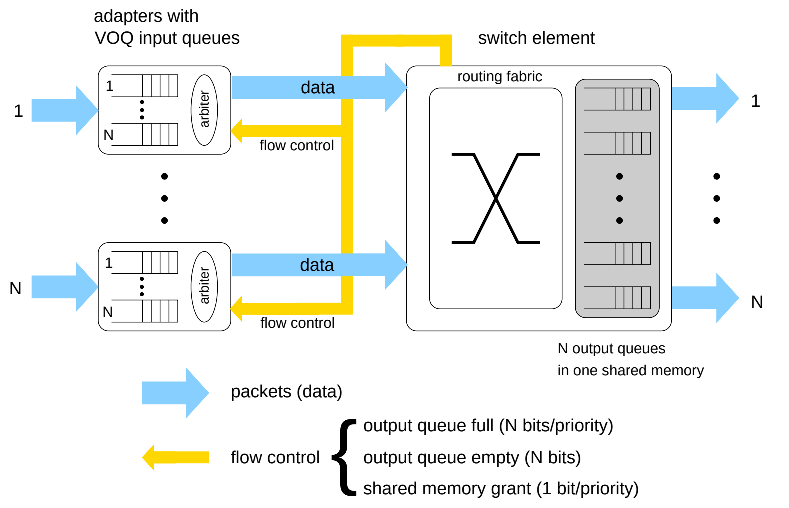


Figure 3.1: A CIOQ packet-switch architecture combining VOQ and shared-memory output queuing.

3.2.2 Grant-based flow control

Coordination between the input queues and the shared-memory switch is required to ensure that the switch architecture is lossless internally. Therefore, the switch should notify the input queues when its shared buffer is (close to being) full, so that they can stop forwarding packets before any have to be dropped because of buffer overflow. We opt for a *grant*-type flow control to achieve this,[1] and refer to this as the *memory grant*. The memory grant is derived by comparing a programmable shared-memory threshold to the current memory occupancy.

To enable optimum scheduling of the VOQs, the switch element communicates output-queue status information to the input queues. *Output-queue grant* is the permission to transmit to an

[1]A grant is a pro-active type of flow control that prevents packets from being lost in the first place. *Back pressure* on the other hand, is a reactive type of flow control that requests retransmission of packets that were lost.

output queue. This flow-control mechanism passes a vector of NP bits (one bit per output and per traffic class, assuming P traffic classes are supported) to all inputs. The output-queue grant is derived by comparing a programmable output-queue threshold to the current queue occupancy. Owing to the shared-memory architecture of the switch element, each shared-memory location can in principle be allocated to any output queue (usually implemented as linked lists of pointers to shared-memory locations), so any output queue can in principle hold all packets (locations). The threshold-based grant mechanism enables the degree of sharing to be tuned, so that the length of a single queue is limited to the threshold. We assume that the thresholds are programmed once and remain constant during switch operation. Adaptively changing thresholds is beyond the scope of this dissertation.

Note that the memory grant overrides the output-queue grant: if no memory grant has been received, an input will not forward any packets, even if for some packets output-queue grant has been received.

3.2.3 VOQ selection algorithm

Each input independently selects (using an $O(N)$ complexity arbiter) which VOQ may forward a packet, thus avoiding HoL blocking. To ensure fairness across VOQs, we must carefully choose a selection rule. Four alternatives policies have been considered: round-robin, least recently used, longest queue first, and oldest queue first. Fairness and HoL blocking elimination together constitute *input contention resolution*.

Round-robin (RR) selection

The selection process is performed in a round-robin fashion, i.e., the outputs are ordered by increasing output number. Each input i maintains a pointer to the VOQ j that was served most recently, say VOQ_i^j. Starting from $\mathrm{VOQ}_i^{(j+1) \bmod N}$, the arbiter selects the next VOQ for which (a) at least one packet is waiting and (b) output-queue grant has been received. Using a specially designed programmable priority encoder, this selection can actually be made in constant time [Gupta99]. Note that the RR policy predefines the order in which the VOQs are visited each cycle, namely, in ascending order of output number (wrapping around from $N-1$ to 0).

Least recently used (LRU) selection

Each input maintains a linked list that defines the order in which the VOQs are visited in this cycle. The list is initialized in a random order. The VOQ at the head of the list is the least recently used one, and should be served next. If it cannot be served because of an output-queue-full condition or because it is empty, the list is traversed until a VOQ is found that *can* be served. The selected VOQ is removed from its current position in the list and appended at the end. Like the RR policy, this implies that the most recently served VOQ now has lowest priority. The key difference is that the LRU policy implies that the *least* recently served VOQ has *highest* priority, which is not true for the RR policy, because of its predefined order. The LRU policy is clearly more expensive to implement, because a linked list must be maintained. Insertion into the list is easy, as entries are only added at the end of the list. However, to ensure constant-time deletion operations, a doubly linked list maintaining pointers to both previous and next entries must be implemented.

The list consists of an array of N entries, plus one *head* pointer and one *tail* pointer, indicating the current head and tail inputs of the list, both $\log N$ bits in size. Each entry consists of a *previous* and a *next* pointer. The input number of the current entry is implicit by its position in the array. Each pointer is $\log N$ bits in size, so that the total storage complexity of the linked list equals $(2N + 2) \log N$ bits.

Longest queue first (LQF) selection

Another possible selection policy is the *longest queue first* (LQF) selection, which, as its name suggest, selects the VOQ that currently has the most packets waiting. Although this policy has the desirable property that it attempts to keep the lengths of the VOQs equal, it has one significant drawback: it can lead to starvation of *short* queues. Suppose we have a 2×2 switch where there is one packet waiting in VOQ_1^2 and two packets in both VOQ_1^1 and VOQ_2^2. From this time on, each cycle a packet arrives at both VOQ_1^1 and VOQ_2^2. The LQF policy will always serve these two VOQs, because they are the longest, and as their packet departure rates equal their arrival rates they will always contain two packets, so that VOQ_1^2 will remain unserved indefinitely. For this reason we will not consider this policy further.

Oldest queue first (OQF) selection

The last selection policy we will present here is the *oldest queue first* (OQF) selection, which selects the packet that has waited longest at the head of its VOQ. This policy does not lead to starvation of any individual VOQ, because an unserved HoL packet will age until it eventually is the oldest, at which point it will be served.

The OQF policy operates as follows: Each input maintains a list of its VOQs, sorted in order of increasing time (oldest first). Initially, the list is empty. In each cycle, the list is traversed in order until a (non-empty) VOQ for which output-queue grant has been received is found, and this VOQ is served. When a packet arrives at the HoL of a previously empty VOQ, the VOQ is inserted into the list according to the time of arrival at HoL (local packet clock). When a VOQ is served, it is removed from its current position, and appended at the tail of the list, *except* if it is now empty, then it is removed from the list altogether. This last clause is in fact the only difference between OQF and LRU, i.e., empty VOQs do not age with OQF, whereas they do with LRU.

To avoid having to maintain arrival timestamps and perform time-consuming sorted-list insertions, we note that insertions into the sorted list *always* occur only at the end of the list:

- A packet arrives at an empty VOQ; this packet clearly has the youngest HoL arrival time and, correspondingly, is appended at the tail of the list.

- A packet is served from the VOQ selected, and the next packet (if present) takes its place at the HoL. The new HoL packet again is the youngest and is appended at the tail of the list. If there are no more packets in this VOQ, it is removed from the list.

Therefore, OQF is not more difficult to implement than LRU and, like LRU, at most one insertion and one deletion operation are required per packet cycle.

The OQF policy differs subtly from an *oldest cell first* (OCF) policy in that the latter implies that always the oldest packet is served, whereby the age is determined by the total time the packet has waited at the input, which requires aging all packets individually. OQF on the other hand only ages the HoL packet. A true OCF policy is expected to be optimum in terms of latency jitter performance because it most closely mimicks a FIFO policy, but unfortunately it is very expensive to implement, as timestamps must be kept for every packet at the input.

Note that all selection policies arrive at a decision after examining at most N different packets, and therefore have a time complexity of $O(N)$.

Distributed scheduling and buffering

The proposed CIOQ architecture implements a two-stage pipelined approach to scheduling: the arbiters at the input side perform input contention resolution, whereas the output-buffered switch element performs classical output contention resolution. Thus, the scheduling process is separated into its two essential components, and distributed accordingly over the architecture. With this distribution of functionality, the superlinear complexity of schedulers in traditional input-buffered VOQ systems is reduced to $O(N)$. By replacing the crossbar by an output-buffered fabric, the need for coordination between VOQ selection across inputs (which necessitates a centralized scheduler) is eliminated.

Note that in this architecture, the switching element no longer functions as a buffer in which to store bursts but rather as an output contention resolution device, just like the scheduler in Fig. 2.11. By means of the flow-control/VOQ interaction between switch element and input queues, the less expensive (because of its lower bandwidth) input-queue memory is used to cope with burstiness.

3.2.4 VOQ arbitration analysis

To analyze the behavior of the proposed system, we make the following abstractions:

- The shared memory is partitioned per output, so that each output queue can accept packets independently of all others. Each output queue can store exactly N packets.
- Flow control is instantaneous, i.e., not synchronized just on packet cycle boundaries. The inputs can immediately be stopped from sending to a particular output queue, as soon as it is full.
- To achieve output access fairness across inputs, inputs are served in a round-robin fashion in every packet cycle, starting at the input beyond the one served first in the preceding cycle.
- In every cycle, first the output queues are served, then the VOQs. VOQ_i^j refers to the VOQ of input i for output j.

In the following subsections we will analyze the maximum worst-case delay a packet can experience from the time it arrives at the HoL of its VOQ until the time it leaves the switch for the RR, LRU, and OQF policies.

RR selection policy

We will demonstrate that, using the RR policy, the maximum delay of a packet arriving at the HoL of a VOQ is unbounded. The proof is by example. Fig. 3.2 shows a 3×3 switch with VOQs and a switch element with partitioned memory, so that each output queue can store exactly three packets.

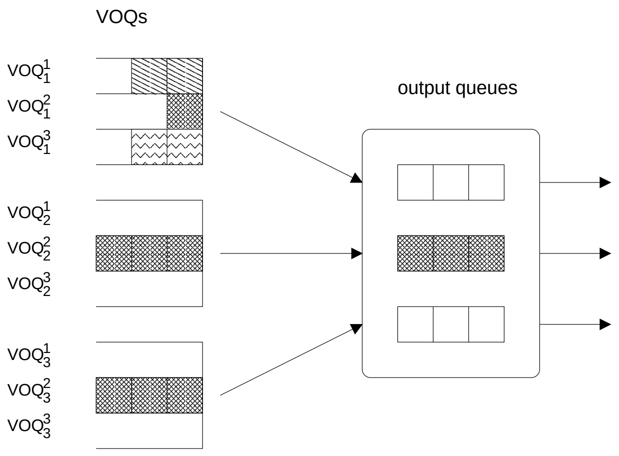


Figure 3.2: Pathological traffic condition under which RR selection leads to starvation.

Observe the depicted scenario. OQ_2 is currently full; it transmits one packet, freeing up one place. Suppose input 2 is currently first to be served: VOQ_2^2 will be served, so that OQ_2 is full again. Input 3 cannot transmit. Input 1 serves VOQ_1^1 and updates the RR pointer to VOQ_1^2. In the next cycle, VOQ_3^2 will be served first, so that OQ_2 is full again. Now, input 1 will *skip* VOQ_1^2, and serve VOQ_1^3 instead, to prevent HoL blocking. The RR pointer is updated to VOQ_1^1, so that the initial system state is reached again. This pattern can be repeated ad infinitum.

Now the problem becomes obvious: because of the RR VOQ pointer-update algorithm, VOQ_1^2 is always skipped, causing it to be starved indefinitely under this traffic scenario. Although the RR algorithm is simple and cheap to implement, it may lead to starvation, so alternatives must be considered.

Now compare the VOQ service list updates at input 1 according to both the RR and LRU algorithm:

RR	
initially:	$\mathrm{VOQ}_1^1 \to \mathrm{VOQ}_1^2 \to \mathrm{VOQ}_1^3$
VOQ_1^1 is served:	$\mathrm{VOQ}_1^2 \to \mathrm{VOQ}_1^3 \to \mathrm{VOQ}_1^1$
VOQ_1^3 is served:	$\mathrm{VOQ}_1^1 \to \mathrm{VOQ}_1^2 \to \mathrm{VOQ}_1^3$
VOQ_1^1 is served:	$\mathrm{VOQ}_1^2 \to \mathrm{VOQ}_1^3 \to \mathrm{VOQ}_1^1$

LRU	
initially:	$\mathrm{VOQ}_1^1 \to \mathrm{VOQ}_1^2 \to \mathrm{VOQ}_1^3$
VOQ_1^1 is served:	$\mathrm{VOQ}_1^2 \to \mathrm{VOQ}_1^3 \to \mathrm{VOQ}_1^1$
VOQ_1^3 is served:	$\mathrm{VOQ}_1^2 \to \mathrm{VOQ}_1^1 \to \mathrm{VOQ}_1^3$
VOQ_1^2 is served!	$\mathrm{VOQ}_1^1 \to \mathrm{VOQ}_1^3 \to \mathrm{VOQ}_1^2$

Although the model analyzed here is an abstraction of our real switch, it is possible that each time an input attempts to serve a particular VOQ, it finds that the output queue is full. Although the output queue will certainly become available again, and all input queues can then access it simultaneously, there is no guarantee that they will actually do so, in case there are other VOQs to be served.

LRU selection policy

We will demonstrate that, using the LRU policy, the maximum delay of a packet arriving at the HoL of a VOQ equals $N^2 - 1$ packet cycles.

To obtain the worst-case delay of a tagged packet that just arrived at the HoL of VOQ_i^j, we assume that output queue j is full, that VOQ_i^j has been served most recently on input i, and that input $i + 1$[2] is the first to be served in the current cycle. Furthermore, we assume that each input has at least N packets destined for output j, and none for any of the other outputs. We set the time of HoL arrival of the tagged packet at $t = 0$.

As output queue j is full, only one of its corresponding VOQs can be served per cycle. Given that all inputs currently only have packets for output j, VOQ_{i+1}^j through VOQ_{i-1}^j will be served during $t = 0$ through $t = N - 2$. Now input i is the first to be served, but just in time for $t = N - 1$ a packet for queue VOQ_i^{j+1} arrives, which is immediately served because it is ahead of VOQ_i^j in the LRU list. During $t = N - 1$ through $t = 2N - 3$, again VOQ_{i+1}^j through VOQ_{i-1}^j are served. At $t = 2N - 2$ input i is first to be served again, and a packet arrives at VOQ_i^{j+2}. Similarly, VOQ_i^{j+3} is served at $t = 3N - 3, \ldots, \mathrm{VOQ}_i^{j-1}$ at $t = (N-1)N - (N-1)$, until finally at $t = N^2 - N$, the packet at the HoL of VOQ_i^j is served. Thus, the maximum possible delay of any packet arriving at HoL of any VOQ equals $N^2 - N$ packet cycles. The maximum delay in the output queue, which is always served in FIFO order, equals $N - 1$ cycles, so that the maximum total delay is $N^2 - N + (N - 1) = N^2 - 1$ cycles.

Note that this result does not depend on the instantaneous flow control assumption. If flow control is updated only at packet cycle boundaries (but still has no transmission latency), we have to provide N extra packet locations in the shared memory to cope with this and prevent packet loss. In this case, the VOQs corresponding to output j will only be served once every $N - 1$ cycles, but all $N - 1$ VOQs will be served at once (the tagged input i is forced to serve

[2]All indices are to be interpreted modulo N. For the sake of brevity and readability we omit the "mod N" operation.

its other VOQs one by one at these instants, as in the scenario described above), so that also in this case, the total delay amounts to $N(N-1)+(N-1)=N^2-1$ cycles.

OQF selection policy

For the OQF policy, a scenario similar to, but not quite the same as the one used with LRU, can be constructed. We observe a tagged packet that advanced to the HoL of VOQ_i^j at $t=0$, so that VOQ_i^j is last in the OQF list of input i. Furthermore, output j is full, all $\text{VOQ}_{i'}^j$ are non-empty $\forall i'$, and all $\text{VOQ}_i^{j'}$ are non-empty $\forall j'$. Note that the main difference in the scenarios used with LRU and OQF is that in the latter case the packets on input i are already present (and older than the tagged packet) at $t=0$. The reason is that with OQF, no packet arriving after the tagged packet can preempt it, as in the LRU scenario. Using OQF with that scenario, the tagged packet would be served as soon as input i is first to be served (which would be at $t=N-1$), because it is the oldest.

Therefore, we attempt to delay serving the tagged packet by populating all VOQs on the same input and putting VOQ_i^j last in the OQF list. However, while VOQ_{i+1}^j through VOQ_{i-1}^j are served at $t=0$ through $t=N-2$, VOQ_i^{j+1} through VOQ_i^{j-1} are served at input i. Consequently, at $t=N-1$, input i is first to be served, while VOQ_i^j has advanced (aged) to the head of the OQF list, regardless of the arrivals at input i, so that the tagged packet is served. Repeating the scenario starting at the next cycle $t=N$, VOQ_i^j will be served again at $t=2N-1$, and again at $t=3N-1$, etc., so that the tagged VOQ is served at least once every N cycles.

As a result, the maximum delay with the OQF policy equals $N-1+N-1=2(N-1)$ packet cycles.

3.2.5 Memory sharing by means of OQ grant

In the proposed switch architecture the available switch-element memory space M is shared among all output queues, i.e., there is no static allocation of memory locations to output queues. This is traditionally seen as an advantage, because more memory can be allocated dynamically to outputs that are busier than others, leading to more efficient memory utilization. On the other hand, too much sharing may lead to unfairness.

By means of the programmable output-queue thresholds $T_Q(j)$, the degree of memory sharing can be tuned. As the thresholds are programmable but static, we typically program the thresholds to the same value for all outputs. In each packet cycle, the switch communicates the status of all output queues (output grant $G_Q(j) := \text{Occ}_Q(j) < T_Q(j)$) to all input queues, where $\text{Occ}_Q(j)$ equals the current occupancy of output queue j. The input queues do not transmit any packets destined to an output queue for which no grant has been received ($G_Q(j)=0$), but instead store them locally until the blocked destination(s) are available again. The shared-memory threshold T_{SM} prevents packet loss due to shared-memory oversubscription ($\sum_{j=1}^N T_Q(j) > M$). When shared-memory occupancy Occ_{SM} exceeds T_{SM}, the grant to send is removed for all inputs, regardless of output grant. Hence, the shared-memory grant acts as an emergency measure to prevent packet loss.

In general, we must take into account that the switch fabric and the adapters are some non-negligible distance removed from each other. The round-trip time R from adapter to switch and back, expressed in packet cycles, has an impact on both output-queue and shared-memory grants, namely that the grant information is slightly out-of-date when it reaches the input queues. Therefore, T_{SM} will generally have to be smaller than the actual shared-memory size M: when the round-trip time equals R packet cycles, the memory threshold must be set to $M - R \cdot N$ to ensure lossless operation. The simulation model used in Sec. 3.4 assumes that the round-trip time equals two packet cycles, so in all simulations $T_{SM} = M - 2N$, which guarantees that no packets will have to be dropped because of memory overflow. For the output queues, the delay implies that the output-queue thresholds may be violated. Arriving packets that violate the thresholds are still accepted, i.e., the thresholds are not “hard”. Thus, in the unlikely worst case in which all packets arriving in both the current and the following cycle are destined to the same output, its threshold may be exceeded by $2N - 1$ packets, with $R = 2$.

3.3 Throughput Analysis under Uniform Traffic

In this section we will show that the system throughput under uniform traffic equals 100%.

3.3.1 Assumptions and definitions

We assume that in each packet cycle, first packets arrive at the VOQs, then the output queues are served, and finally the VOQs are served.

To simplify the analysis, we assume that output grant is instantaneous and that the inputs do not perform parallel VOQ arbitration (as in the real system). Instead, the arbitration is modeled as a serial, N-step process. The inputs are served according to a RR schedule ($0 \to 1 \to \cdots \to N-2 \to N-1 \to 0$), where the starting point is shifted by one input every cycle. Each input randomly selects a VOQ corresponding to a *non-full* output queue.

We define: $L_i^j(t)$ is the occupancy of VOQ_i^j at time t (including packet arrivals at time t), $Q^j(t)$ is the occupancy of output queue j at the end of time slot t, $Q^j(t) \leq \hat{Q}, \forall t, j$, where $\hat{Q}$ is the output queue size.

$Q_k^j(t)$ equals the number of packets in output queue j after k input queues have been served. Hence, $Q_0^j(t)$ equals the number of packets in output queue j after the output queue has been served, but before the VOQs have been served, and $Q_N^j(t) = Q^j(t)$ equals the occupancy of output queue j at the end of the cycle. $S^j(t)$ equals the number of packets transferred from all VOQs to output queue j at time t; $S_i(t)$ is the number of packets served at input i in the current cycle. Note that

$$S^j(t) = Q_N^j(t) - Q_0^j(t), \tag{3.1}$$

$$Q_0^j(t) = \max(0, Q^j(t-1) - 1), \tag{3.2}$$

$$0 \leq S_i(t) \leq 1, \tag{3.3}$$

$$0 \leq S^j(t) \leq \min(\hat{Q} - Q_0^j(t), N). \tag{3.4}$$

Furthermore, the total number of packets served at the inputs must equal the total number of packets transferred to the outputs, call this number $C(t)$:

$$C(t) = \sum_{i=0}^{N-1} S_i(t) = \sum_{j=0}^{N-1} S^j(t). \tag{3.5}$$

Given a boolean expression dependent on k, $X(k)$, we define the notation $||X(k)||_k$ as follows:

$$||X(k)||_k = \left(\sum k : 0 \leq k < N \wedge X(k) \equiv \texttt{true} : 1\right). \tag{3.6}$$

In other words, $||X(k)||_k$ counts the number of k in the given range for which $X(k)$ holds.

Without loss of generality we can say $Q^j_k(t) = Q^j_i(t)$, where k is the index of input i in the current RR schedule, by simply reordering the inputs at the start of each cycle according to the RR schedule. Then $S_i(t)$ can be expressed as follows (we posit that the existence operator returns numeric value 1 iff it evaluates to true, 0 otherwise):

$$S_i(t) = \left(\exists j : 0 \leq j < N : L^j_i(t) > 0 \wedge Q^j_i(t) < \hat{Q}\right). \tag{3.7}$$

Consequently,

$$S_i(t) = 0 \Leftrightarrow \left(\forall j : 0 \leq j < N : L^j_i(t) = 0 \vee Q^j_i(t) = \hat{Q}\right). \tag{3.8}$$

From $Q^j_0(t) = \max(0, Q^j(t-1) - 1)$ and by definition $Q^j(t-1) \leq \hat{Q}, \forall t$, it follows that

$$Q^j_0(t) < \hat{Q}, \forall t, j. \tag{3.9}$$

A necessary and sufficient condition for work-conservingness can be expressed as follows, using the above definitions (ranges omitted for brevity):

$$\left(\forall t, j : Q^j_0 = 0 \wedge \left(\exists i :: L^j_i(t) > 0\right) : S^j(t) > 0\right). \tag{3.10}$$

Since non-empty output queues are served anyway, we only have to worry about empty output queues; the condition simply demands that for all empty output queues for which there is at least one packet at the input, at least one of those packets must be transferred. If this condition is satisfied for all t, the system is work conserving.

3.3.2 Saturation under uniform traffic

Using (3.10), we show that the system offers 100% throughput under uniform traffic. Assuming the system is in saturation for $t > t_0$, that is, every VOQ always has a packet available, $L^j_i(t) > 0, \forall t > t_0$, in which case (3.10) reduces to

$$\left(\forall j : Q^j_0 = 0 : S^j(t) > 0\right), t > t_0, \tag{3.11}$$

and (3.7) to

$$S_i(t) = \left(\exists j :: Q^j_i(t) < \hat{Q}\right). \tag{3.12}$$

Lemma 1 $C(t) = N, \forall t > t_0$.

Proof: It is easy to show that $S_i(t) = 1, \forall i, t > t_0$, and hence $C(t) = \sum_i S_i(t) = N$. Consider the RR scheduling process over all inputs. As there is at least one free position in each output queue (by (3.9)), at least N packets can be served. Suppose (3.12) does not hold true for all i, then there must be some i^* for which $\left(\forall j :: Q^j_{i^*}(t) = \hat{Q}\right)$, which implies that at least N packets have been served up to that point. However, as each input can only serve one packet, no more than i^* packets have been served so far. As there are only N inputs, this leads to a contradiction, which proves the lemma. □

The instantaneous throughput $T(t)$ is defined as follows (3.13):

$$T(t) = \frac{1}{N}\left(\sum j : 0 \leq j < N \wedge Q^j_N(t) > 0 : 1\right) = \frac{||Q^j(t) > 0||_j}{N}. \tag{3.13}$$

Lemma 2 *If for some t_0, $C(t) = N, \forall t > t_0$, then for some $t_1 > t_0$, $T(t) = 1, \forall t > t_1$.*

Proof: The quantity $Q(t) = \sum_j Q^j(t)$ equals the total number of packets at the output side at the end of time t. As no more than N packets can depart in one cycle and $C(t) = N$, this is a monotonously non-decreasing function for $t > t_0$. Assuming $Q(t_o) = Q_0 < N\hat{Q}$, either one of the following must hold: either there exists a $t_1 > t_0$ for which $Q_0 \leq Q(t_1) < NQ$ and $T(t) = 1, \forall t > t_1$, or there exists a $t_1 > t_0$ with $Q(t_1) = NQ$. The latter case implies that $Q^j(t) = \hat{Q}, \forall j, t > t_1$, and thus also $T(t) = 1, \forall t > t_1$. Note that t_1 is finite because NQ is finite. □

From Lemmas 1 and 2 we conclude that under uniform traffic throughput equals 100% after some limited time t_1.

3.3.3 Ideal OQ emulation

An interesting question is whether the proposed CIOQ system can emulate an ideal output queue system using some ideal VOQ scheduling algorithm and some limited output queue size $\hat{Q}$. As we shall see, the answer is no. Below, we will prove the following result: no CIOQ switch with input speed-up $1 \leq S_i < N$, output speed-up $S_o = N$, and limited output buffers can be work-conserving in the sense of Definition 2 under all traffic patterns, regardless of scheduling algorithm. The proof is by counter-example. This is an important result, because none of the schemes reviewed in Section 2.7.4 seem to have considered this.

Consider Fig. 3.3, with a switch of size $N \times N$, output buffers of size Q packets each, and an input speed-up S. The output speed-up equals N. Suppose the switch system has entered a state as depicted in Fig. 3.3, where at t_0 exactly S output queues, say queues 1 through S, are completely full, whereas all others are completely empty, and one particular input i has a number of packets for each of these full output queues; such a situation can occur either because of prolonged contention for the given outputs or because backpressure has been applied at these outputs. We assume that from t_0 on, none of the other inputs has any traffic, so we will just look at the given input.

Because the input speed-up equals S, we can transmit up to S packets from the input in one packet cycle, so in K cycles, a total of KS packets can be transferred. Assume that from t_0 on, in every cycle one packet arrives at this input destined for (empty!) output $S + 1$. To remain work-conserving, this packet must always be served immediately, so that up to $S - 1$ packets from the other VOQs can be served; we assume these are served in a RR order, but note that the actual service discipline does not matter for the final result. Thus, at $t_0 + 1$, we serve one packet for output $S + 1$, and one each for outputs 1 through $S - 1$. At the end of

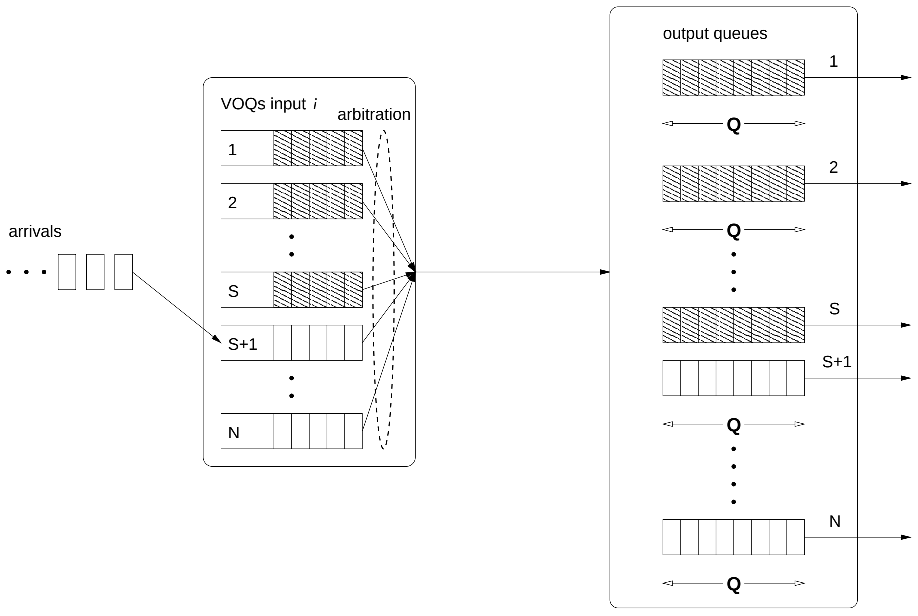


Figure 3.3: Pathological scenario to disprove ideal OQ emulation.

the cycle, the output queues now contain Q packets at outputs 1 through $S-1$ (one arrival, one departure), and $Q-1$ packets at output S (one departure). Continuing in this fashion, we find that after K cycles, with K an integer multiple of S, $K = nS, n \in \mathbb{N}$, we have served K packets for output $S+1$, and $KS - K = K(S-1)$ packets total for outputs 1 through S; because of the RR operation, $K(S-1)/S = nS(S-1)/S = n(S-1)$ packets were transmitted from each corresponding VOQ. At the same time, K packets have departed from each output queue 1 through S, so that after K cycles the occupancy of these output queues equals $Q + K(S-1)/S - K = Q + K((S-1)/S - 1) = Q - K/S = Q - n$ packets. Therefore, with $n = Q$, after $K = nS = QS$ cycles all output queues will be empty. If at this point, $t_0 + QS$, there are still packets left in all of the corresponding VOQs and another packet for output $S+1$ arrives, then in order to remain work-conserving $S+1$ packets destined for empty output queues must be served, which is not possible because the speed-up is just S. Hence, we have demonstrated a traffic scenario under which the given switch system is not work-conserving.

As a result, no CIOQ switch system with limited output buffers of any size Q and an input speed-up factor $1 \leq S < N$ can exactly emulate an ideal output-queued switch under all traffic patterns, regardless of scheduling algorithm.

This result implies that none of the algorithms reviewed in Section 2.7.4 can, strictly speaking, be practically implemented such that they are truly work-conserving. It also implies that more buffer space at the output is always better, so that a trade-off between the cost of additional output buffer space and the performance increase it entails must be evaluated with care.

3.3.4 Buffer sizing

A critical resource, in terms of both cost as well as performance, is the shared memory. A large shared memory is very costly because of its high bandwidth, but too little memory will have a negative impact on performance. Because the buffering and contention resolution functions are split between input and output queues in the proposed architecture, we will reduce the amount of memory in the switch to the minimum. We argue that, so that the contention resolution process of any one output does not interfere with that of any of the other output, we require a space of N packets per output for a total shared memory size of N^2.

As in the worst case N packets can arrive at an output queue simultaneously, each output queue should be able to store at least N packets to be fair to all inputs. Furthermore, to ensure that every output queue can resolve output contention independently of all other outputs, *each* output queue should be able to store at least N packets, so that N^2 packets in total are required.

The output queue can be seen as a pipeline of VOQ arbiter decisions: when there are x packets in an output queue, then the VOQ arbiters can in principle postpone scheduling packets for this queue until x packet cycles in the future at the latest.

In Section 3.4 we will verify by means of simulation that a memory size of N^2 is optimum in terms of a cost vs. performance trade-off.

Note that an increased round-trip time between switch and adapter increases the memory requirements proportionally.

3.4 Performance Simulation Results

3.4.1 Introduction

The proposed architecture is fairly complex, and analyzing its performance even under very simple traffic patterns is exceedingly difficult. Therefore, computer simulations using a software (C++) model of the architecture have been carried out, the results of which will be presented below. Three traffic models with different interarrival correlations are used in these simulations:

- An uncorrelated *Bernoulli* type: subsequent packet arrivals at an input are independent.
- A correlated, *bursty* type: packets arrive in bursts, a burst being defined as a sequence of consecutive packets from an input to the same output. The burst size is geometrically distributed around a configurable mean B. We refer to this type as Bursty/B.
- An IP-like type, which is also bursty, but has an irregular burst-size distribution, based on the Internet backbone traffic measurements of [Thompson97].

Unless otherwise noted, all traffic types are identically distributed across all inputs, and the destination distributions of the (burst) arrivals are uniform across all outputs.

Each simulation consists of a number independent runs to obtain a 95%-confidence interval on the throughput. The simulation will terminate when either the target confidence interval of 1% or the maximum number of runs (ten) is reached, whichever comes first.

Appendix C provides detailed information about the performance simulation environment in general, and the simulator, traffic models, and system model in particular.

In the following sections, we will investigate the impact of a variety of system parameters, both in isolation and combination, on the performance characteristics of the proposed architecture.

We will investigate the effect of varying the degree of memory sharing by varying the output-queue thresholds, and we will compare the influence of using the proposed VOQ input queues as opposed to conventional FIFO queues. We will study burst sensitivity, the impact various VOQ scheduling policies, the shared memory size, and the switch size have on performance. We will compare the proposed architecture with an input-queued VOQ architecture, we will study performance under non-uniform traffic patterns, and finally, will investigate input-queue size and delay distributions.

3.4.2 FIFO vs. VOQ input queues and memory sharing

In terms of a reference, we first compare the proposed CIOQ architecture to the conventional one employing FIFO input queues instead of VOQs in order to observe the performance improvement, and investigate the influence of the degree of memory sharing.

Figs. 3.4a through 3.4d present delay–throughput characteristics for a 16×16 switch system with a shared memory size M equal to 256 packets, using either FIFO or VOQ input queues with infinite queuing capability. Each figure contains six curves, corresponding to FIFO and VOQ input queues, with output-queue thresholds set to $\mathrm{OQT} = 256$, corresponding to full output buffer sharing, $\mathrm{OQT} = 64$, corresponding to partial output buffer sharing, and $\mathrm{OQT} = 16$, corresponding to no output buffer sharing (partitioning). Fig. 3.4a corresponds to the IP traffic model, whereas Figs. 3.4b, c, and d correspond to bursty traffic with an average burst size (BS) of 10, 30, and 100 packets, respectively. Table 3.1 presents the corresponding maximum throughput figures. These figures provide us with a number of interesting results, which we will discuss in detail below.

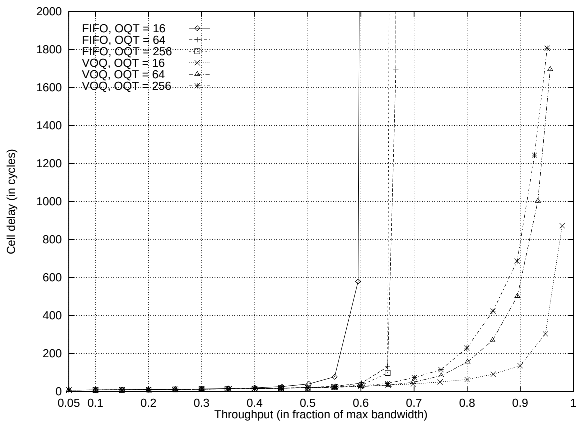
Table 3.1: Maximum throughput as a fraction of the available bandwidth.

Traffic type	OQT=256		OQT=64		OQT=16		4-SLIP
	FIFO	VOQ	FIFO	VOQ	FIFO	VOQ	
Bursty/10	0.710	0.961	0.712	0.961	0.620	0.982	0.973
Bursty/30	0.563	0.932	0.591	0.934	0.560	0.972	0.967
Bursty/100	0.450	0.875	0.515	0.891	0.535	0.955	0.951
IP	0.658	0.951	0.667	0.956	0.601	0.979	0.971

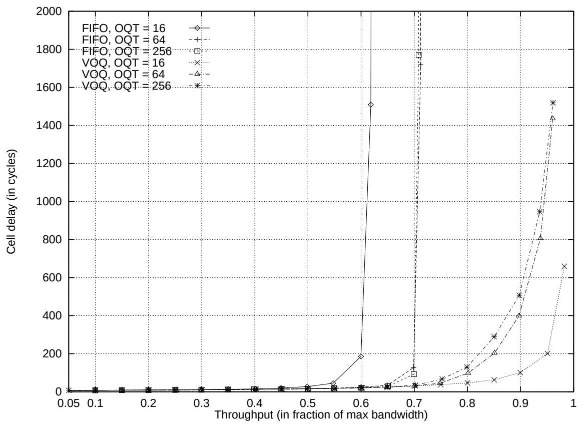
FIFO input queues

First, the system with FIFO input queues performs poorly under all conditions, which is due to HoL blocking, as argued in Section 2.7.3. However, increasing the degree of memory sharing also increases performance. The reason is that, as a result of increased sharing, packets are less likely to be blocked at the head of the input queues owing to output-queue-full conditions, which reduces HoL blocking. This corresponds to the theoretical result of Iliadis in [Iliadis91b], which shows that throughput improves in the given system if the available memory space M is fully shared among all output queues (under assumption of uniform i.i.d. traffic), see also Section 2.7.3.

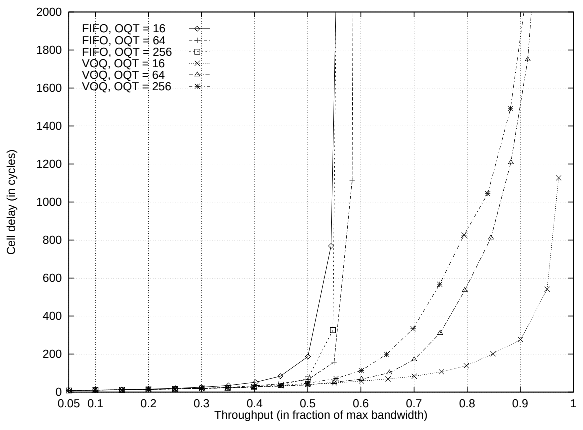
However, increased sharing also increases the probability that an output queue cannot access the shared memory because other output queues have appropriated an unfair portion. When there



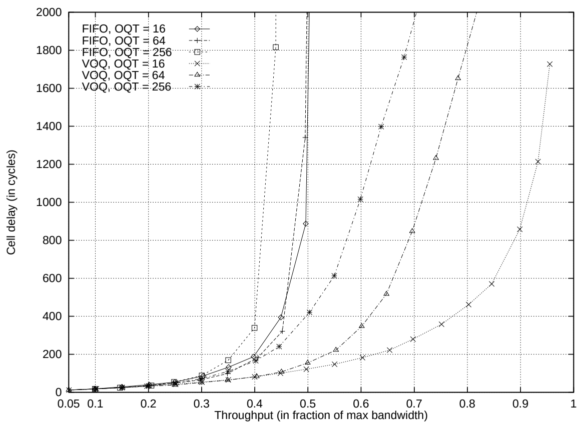
(a) IP traffic



(b) Bursty/10 traffic



(c) Bursty/30 traffic



(d) Bursty/100 traffic

Figure 3.4: Delay–throughput curves of the 16×16 switch system using either FIFO or VOQ input queues under IP traffic (a), and bursty traffic with average burst sizes 10 (b), 30 (c), and 100 (d).

is some degree of sharing, the entire shared memory may be full, while some output queues are not able to obtain access to the shared memory at all, resulting in throughput degradation. We refer to this effect as *output-queue lock-out* (hogging). Especially under very bursty traffic conditions, this effect cannot be neglected, as evidenced by Figs. 3.4c and d, which clearly demonstrate that full sharing is *not* optimum under these conditions. It is therefore important to realize that assumption of the uniform i.i.d. traffic is crucial in the derivation of Iliadis's result.

The reduced HoL blocking and OQ lock-out effects described above counteract each other as memory sharing increases, which implies that for a given memory size and traffic pattern, an optimum threshold setting exists somewhere in between full sharing and full partitioning.

Also note that maximum throughput decreases dramatically as traffic burstiness increases.

VOQ input queues

If the FIFO input queues are replaced by VOQ input queues, the picture changes: first and foremost, delay–throughput characteristics are greatly improved, and secondly, performance improves notably as the degree of memory sharing is *reduced* (i.e. as OQT approaches M/N), and the maximum is actually reached when the memory is partitioned completely ($\mathrm{OQT} = M/N$), a trend contrary to the one observed above.

This can be explained by realizing that the use of VOQ input queues prevents HoL blocking. This implies that the trade-off between reduced HoL blocking and increased output-queue lock-out probability observed with FIFO input queues does not apply here. In fact, as there is no HoL blocking due to output-queue-full conditions, increased sharing results in worse performance because of the increased output-queue lock-out probability. Performance is determined only by the queue space available for every output to resolve output contention and to ensure that packets can be sent to its output, i.e., to keep all outputs busy.

Furthermore, this trend is observed for all simulated traffic types, and becomes more pronounced as burstiness increases. Maximum throughput figures indicate that when the memory is partitioned ($\mathrm{OQT} = 16$), the maximum throughput value decreases only marginally with increased burstiness. This is a highly desirable feature, because it implies that throughput performance is robust regardless of traffic pattern.

Note that partitioning the shared memory may require larger input buffers to achieve the same packet-loss rate of the overall system because more of the queue is pushed back to the input side.

Average delay values are roughly proportional to average burst size, which is consistent with previously reported results.

One more interesting observation can be made from these simulation results, namely, that even when the memory is fully shared ($\mathrm{OQT} = 256$), the performance with VOQs is much better than with FIFOs, especially under highly bursty traffic. Note that the output grant does not convey any information in this case (grant will always be given as long as the memory is not completely full), so that the VOQ scheduling algorithm cannot gain any advantage by using this information. The reason for the improved performance is that the VOQ algorithm acts as a *decorrelator* because of its RR operation. Under heavy traffic, bursts are broken up before they enter the switch, so that output contention is reduced, and performance improves. This last realization is important because it implies that employing the VOQ scheme offers significant benefits even without using output-queue grant.

3.4.3 Burst sensitivity

Fig. 3.5 offers a different perspective: it plots the maximum throughput (actually, the measured throughput at an input load of 100%) as a function of traffic burstiness. The x-axis corresponds to the average burst size B of traffic type Bursty/B. The slope at which a curve declines is a measure for the burst sensitivity of the switch system. We have simulated a wide range of burst sizes, from 10 packets/burst to 500 packets/burst; $N = 16$, $M = 256$.

From these figures, we immediately see that using VOQ adapters offers much better (c.q. lower) burst sensitivity than FIFO adapters, as these curves slope off at a much lower rate. Furthermore,

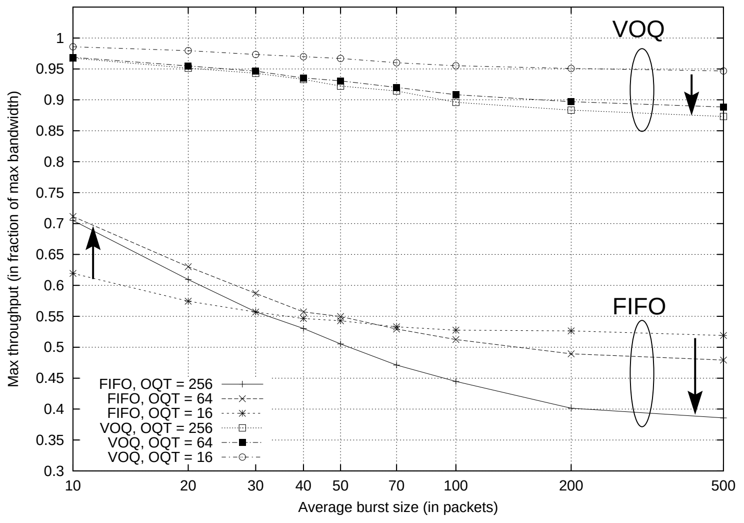


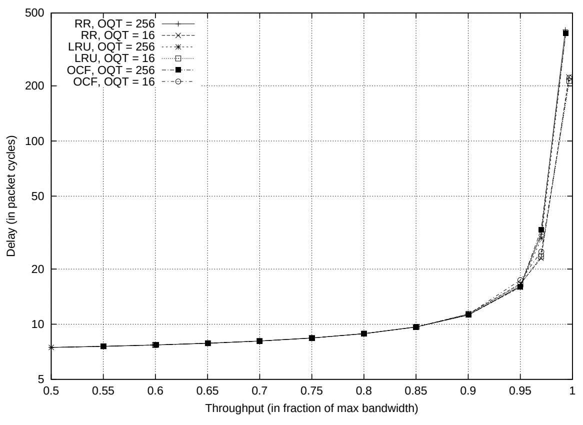
Figure 3.5: Burst sensitivity: impact of degree of memory sharing and input-queuing discipline on maximum throughput as a function of traffic burstiness, with $N = 16$ and $M = 256$. The arrows indicate the trend as the degree of sharing increases. Note how with FIFOs the trend reverses as the burst size increases.

full partitioning of the memory (OQT = 16) offers better burst sensitivity than full sharing (OQT = 256) does. In the fully partitioned VOQ configuration, a throughput of up to 95% can be achieved even for the largest burst sizes! However, even the fully shared VOQ configuration outperforms any FIFO configuration by far, which again illustrates that the decorrelating effect of the VOQs is very strong.

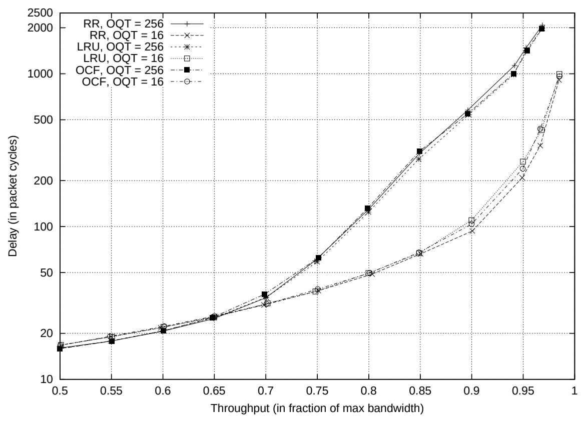
Using FIFO adapters, full sharing offers better throughput than full partitioning at smaller burst sizes, but at an average burst size of about 30 packets/burst a break-even point occurs, and for very large bursts partitioning of the memory performs better. This illustrates the two counter-acting effects described above: first, sharing significantly reduces HoL blocking at the adapter. Second, sharing allows one or a few output queues to monopolize the shared memory, thus causing output-queue lock-out. As the average burst size increases the second, negative, effect starts to offset the first, positive one, eventually leading to a net negative result. The reverse is true for full partitioning, as also illustrated by Fig. 3.5.

3.4.4 VOQ selection policy

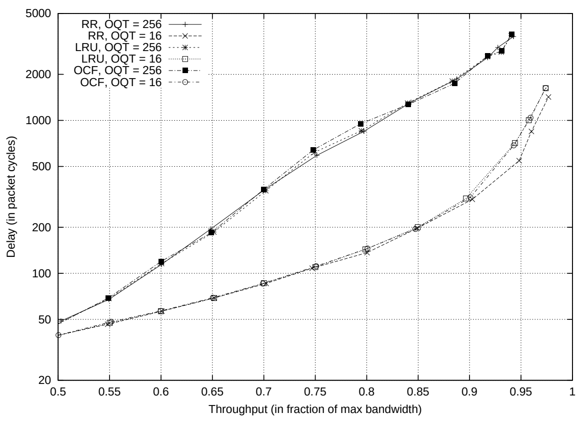
In this section we will study the impact of three different selection policies, as proposed in Section 3.2.3, namely RR, LRU, and OQF. Simulations results with Bernoulli, Bursty/10, 30, and 100 traffic are shown in Fig. 3.6. As the figures demonstrate, no significant difference in performance can be observed between the different policies, even though the analytical results suggest that OQF should have an advantage in terms of worst-case delay. However, under random traffic conditions, the VOQ selection policy is clearly not a significant factor because the pathological conditions described in Section 3.2.3 do not occur.



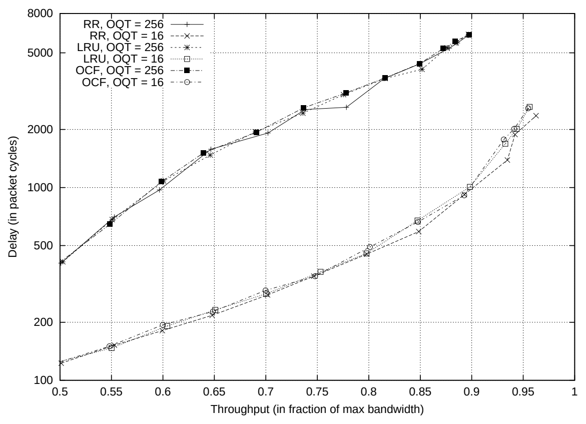
(a) Bernoulli



(b) Bursty/10



(c) Bursty/30



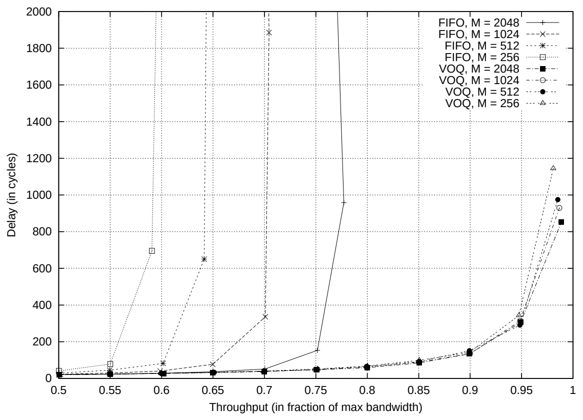
(d) Bursty/100

Figure 3.6: Delay–throughput results comparing RR, LRU, and OQF VOQ selection policies, for a 16 $\times$ 16 system, with $M = 256$, OQT = 16 and 256, and four different traffic types.

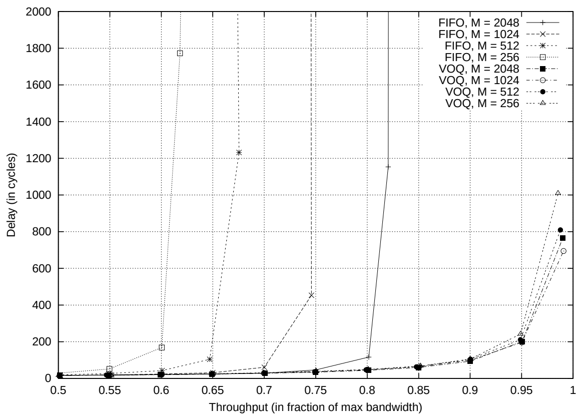
3.4.5 Memory sizing

To demonstrate that $M = N^2$ is the optimum trade-off between performance and cost, we now proceed to investigate the effect of varying the shared memory size M. We simulate a system with $N = 16$ and vary the memory sizes from 256 to 2048 packets with FIFO input queues, 64 to 2048 with VOQ input queues, with the output-queue thresholds set to $\frac{1}{16}$th of the shared memory size, implying full partitioning.

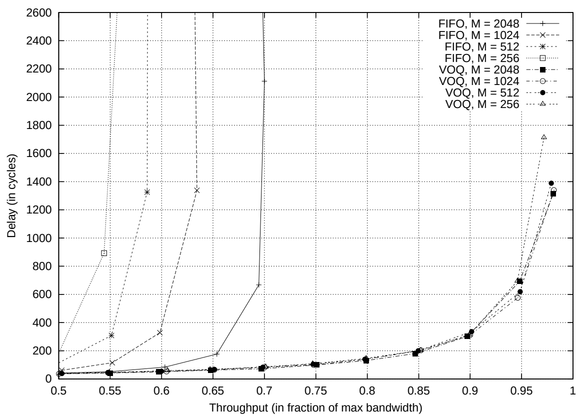
Fig. 3.7 shows delay–throughput graphs for different types of traffic, namely IP traffic and bursty traffic with average burst sizes of 10, 30, and 100 packets/burst. The input load is varied from 50% to 100% in steps of 5%. Fig. 3.8 shows results with just VOQs, with $M = 64, 128, 256, 512, 1024$, and 2048. Table 3.2 lists the maximum throughput figures corresponding to the graphs of Figs. 3.7 and 3.8.



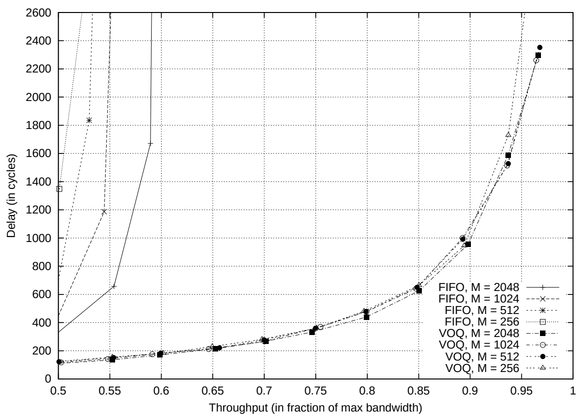
(a) IP traffic



(b) Bursty/10 traffic



(c) Bursty/30 traffic

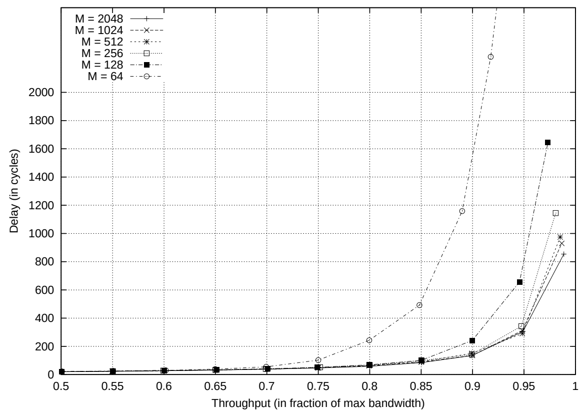


(d) Bursty/100 traffic

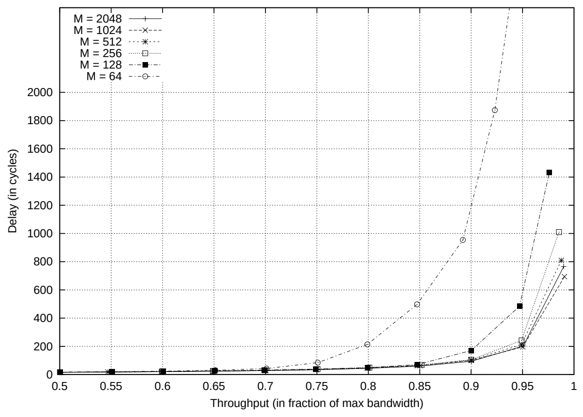
Figure 3.7: Impact of memory size M, input-queuing discipline, and traffic type on delay–throughput characteristics, with full memory partitioning ($\mathrm{OQT} = M/16$).

Table 3.2: Maximum throughput corresponding to Figs. 3.7a through 3.7d.

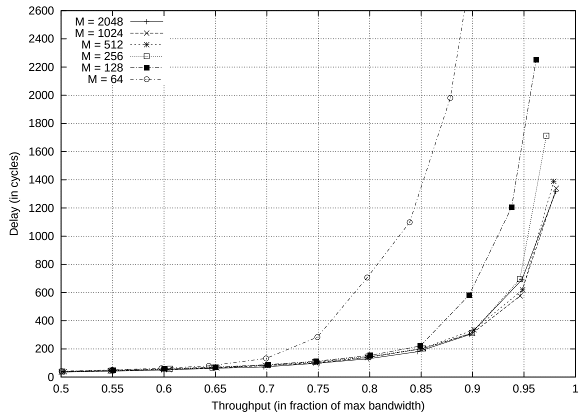
Traffic	FIFO input queues				VOQ input queues					
	M = 256	512	1024	2048	64	128	256	512	1024	2048
B/10	0.622	0.678	0.748	0.822	0.946	0.976	0.985	0.988	0.991	0.990
B/30	0.560	0.585	0.639	0.697	0.921	0.962	0.972	0.979	0.981	0.981
B/100	0.529	0.544	0.558	0.596	0.887	0.937	0.955	0.968	0.964	0.966
IP	0.598	0.645	0.702	0.774	0.938	0.973	0.981	0.985	0.987	0.989



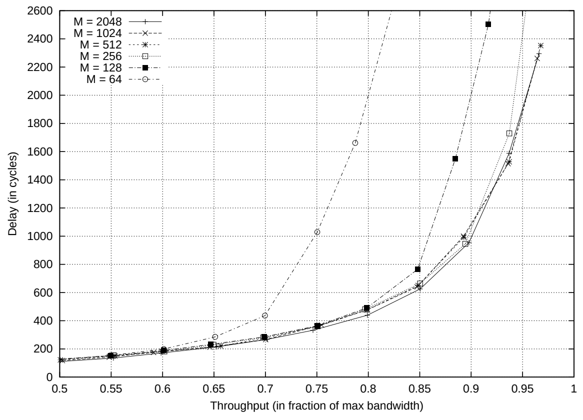
(a) IP traffic



(b) Bursty/10 traffic



(c) Bursty/30 traffic



(d) Bursty/100 traffic

Figure 3.8: Impact of memory size M and traffic type on delay-throughput characteristics, with VOQ input queues and full memory partitioning ($\mathrm{OQT} = M/16$).

Note how as average burst size increases, the system with FIFO input queues saturates at increasingly lower input loads (below 60% even), whereas the system with VOQ input queues can maintain throughput above 90% even for burst size 100; also, in the latter case, enlarging the shared memory beyond 256 improves throughput less than 1 percentage point in all cases, whereas in the former case having a larger memory improves performance by up to 20 percentage points. With FIFO input queues, the relative improvement achieved by increasing the memory size diminishes as the average burst size increases.

This effect is, as noted before, due to the HoL blocking phenomenon in the FIFO input queues, and again, we observe that using per-destination flow control in combination with VOQs completely alleviates the problem: the maximum achievable throughput declines only marginally with increasing burst size. The average delay (in the non-saturated regions of the curves) is roughly proportional to the burst size.

Unlike with FIFOs, the results with VOQs on the other hand are strongly subjected to the law of diminishing returns: the improvement gained from doubling the memory rapidly diminishes. The results confirm the rule that the optimum memory size is N^2, because larger memories bring very little improvement, whereas with smaller memories a drop in performance is observed.

3.4.6 Switch size

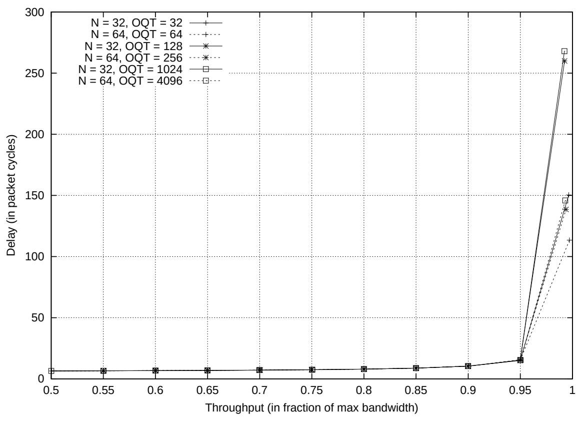
The results obtained for the 16 $\times$ 16 system with VOQs apply to any size $N \times N$. To verify this, we simulate 32 $\times$ 32 and 64 $\times$ 64 systems with shared memory sizes equal to $M = 32^2 = 1024$ and $M = 64^2 = 4096$, respectively. The degree of sharing is again varied from full partitioning to full sharing by programming the output-queue thresholds to $\mathrm{OQT} = N$, $4N$, and M. The usual range of traffic types is applied. Fig. 3.9 displays delay–throughput curves and Table 3.3 shows the corresponding maximum throughput numbers for all configurations.

The results confirm our previous findings. Throughput is maximized when the memory is fully partitioned. Maximum throughput figures decrease only marginally as the size of the switch system increases (see also Table 3.1), although the adverse impact of increased sharing is slightly higher.

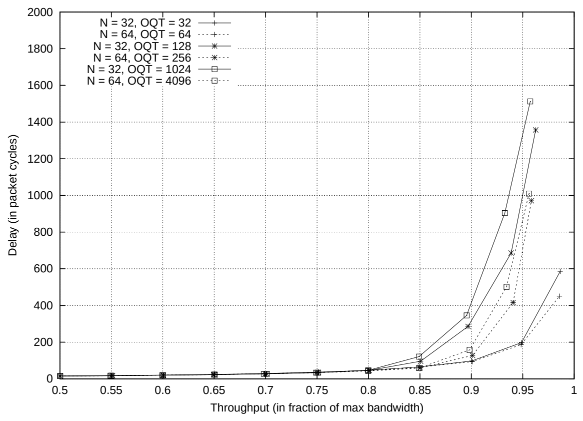
Note how the OQ lock-out effect is significantly smaller for the larger system; this is due to the overall larger memory, which reduces HoL blocking at the inputs due to memory-full conditions. However, when the memory is partitioned, there is little difference in performance between the two switch sizes, except for a slightly lower average delay at loads close to saturation for the larger system.

Table 3.3: Maximum throughput values for switch sizes 32×32 and 64×64.

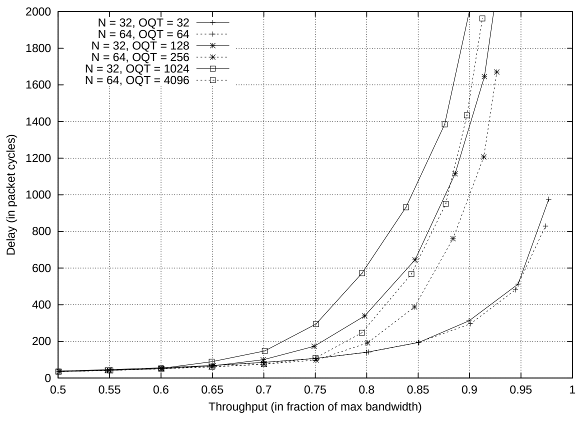
Traffic type	32 × 32				64 × 64		
	OQT = 1024	256	128	32	4096	256	64
Bernoulli	0.992	0.993	0.993	0.996	0.993	0.994	0.997
Bursty/10	0.957	0.959	0.963	0.986	0.956	0.958	0.986
Bursty/30	0.926	0.926	0.934	0.977	0.912	0.926	0.974
Bursty/100	0.865	0.879	0.889	0.960	0.858	0.882	0.957
IP	0.949	0.952	0.959	0.984	0.947	0.954	0.983



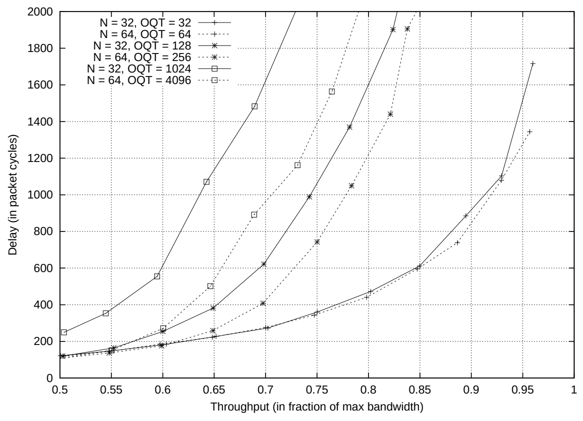
(a) Bernoulli traffic



(b) Bursty/10 traffic



(c) Bursty/30 traffic

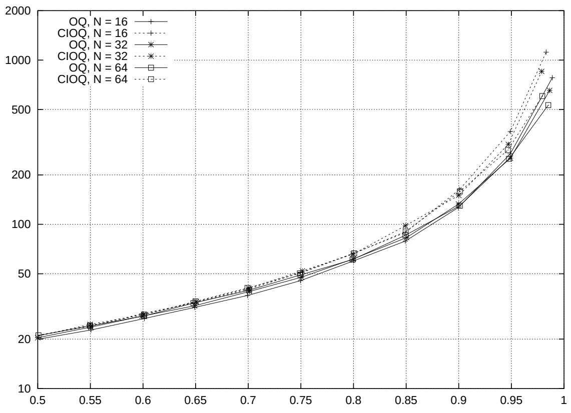


(d) Bursty/100 traffic

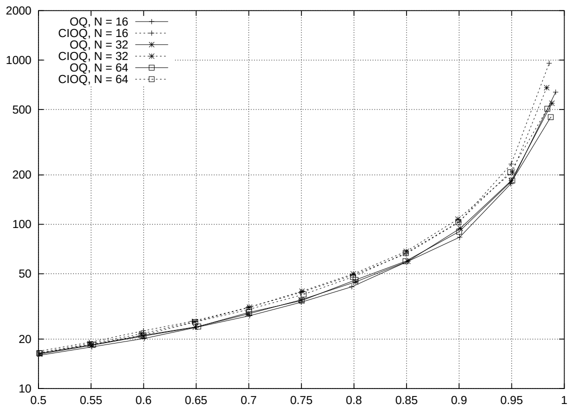
Figure 3.9: Delay–throughput curves for a 32×32 and a 64×64 switch for a range of traffic types and various degrees of sharing.

3.4.7 Comparison with ideal OQ switch

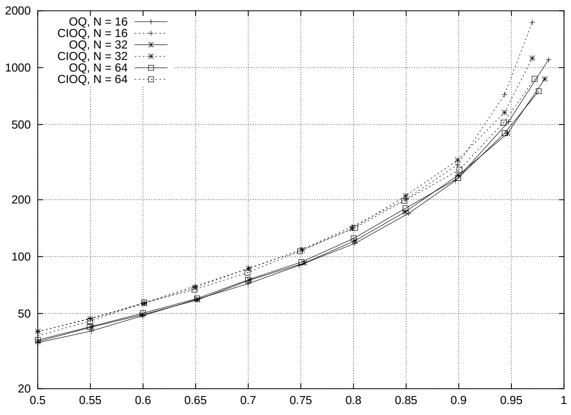
Fig. 3.10 compares the performance of the proposed CIOQ switch with an ideal output-queued switch with infinite output queues for three different switch sizes ($N = 16, 32$, and 64) and traffic types IP, Bursty/10, Bursty/30, and Bursty/100. Table 3.4 lists the corresponding maximum throughput figures. The memory size of the CIOQ switch equals $M = N^2$, with $\mathrm{OQT} = M/N$.



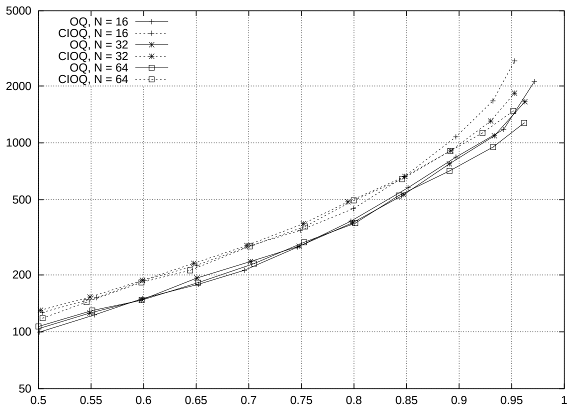
(a) IP traffic



(b) Bursty/10 traffic



(c) Bursty/30 traffic



(d) Bursty/100 traffic

Figure 3.10: Comparison of delay–throughput curves for CIOQ and ideal output-queued architectures with different switch sizes ($N = 16, 32, 64$) and traffic types.

The graphs and table show that delay of the CIOQ switch is slightly higher throughout the load range, and maximum throughput is slightly less. However, the differences are not large: in the 50%-90% load range, the average delay values of the CIOQ switch mostly are less than 25% higher. For larger switches ($N = 32, 64$) and smaller burst sizes (Bursty/10, Bursty/30), the differences are even smaller (10%-15% higher). For $N = 16$, the difference in maximum throughput is less than 1.8 percentage points and for $N = 32, 64$ less than 1.0 percentage points. Therefore, we can conclude that the CIOQ system closely approaches ideal output queuing performance.

Table 3.4: Maximum throughput values comparing CIOQ vs. ideal OQ

Traffic type	16×16		32×32		64×64	
	OQ	CIOQ	OQ	CIOQ	OQ	CIOQ
Bursty/10	0.992	0.986	0.988	0.983	0.987	0.984
Bursty/30	0.985	0.970	0.982	0.973	0.976	0.972
Bursty/100	0.971	0.953	0.963	0.953	0.962	0.952
IP	0.989	0.983	0.986	0.979	0.985	0.979

3.4.8 Comparison with input-queued architecture

As pointed out above, our approach is essentially VOQ with distributed scheduling. Therefore, we would like to compare its performance with approaches that use VOQ with centralized scheduling. As a representative of this class that achieves a very good tradeoff between complexity and performance, we select the i-SLIP algorithm with $i = \log_2 N$ iterations (i-SLIP, [McKeown95]).

Fig. 3.11 compares delay–throughput curves of the CIOQ architecture and the IQ architecture with i-SLIP for various traffic types. Figs. 3.11a and b correspond to a 16×16 system (4-SLIP; M = 256, OQT = 16), 3.11c and d to a 32×32 system (5-SLIP; M = 1024, OQT = 32). Table 3.5 shows maximum throughput values.

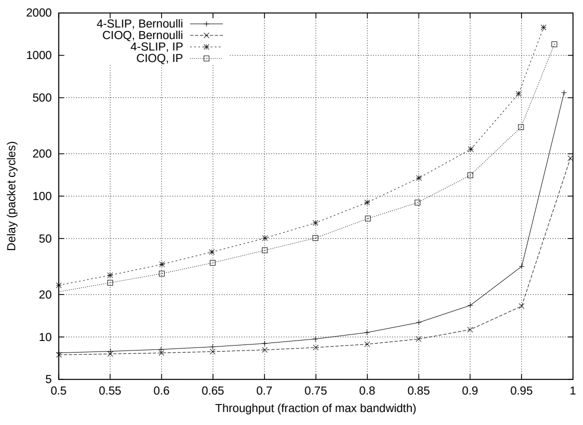
We observe that maximum throughput is marginally better for the CIOQ architecture, but delay is significantly better throughout the entire range of loads; i-SLIP delay is between 20% to 40% higher than CIOQ delay throughout the range. This is due to the reduced input contention in the CIOQ architecture compared to the purely input-queued approach. This applies to both system sizes.

Table 3.5: CIOQ vs. i-SLIP: Comparison of maximum throughput

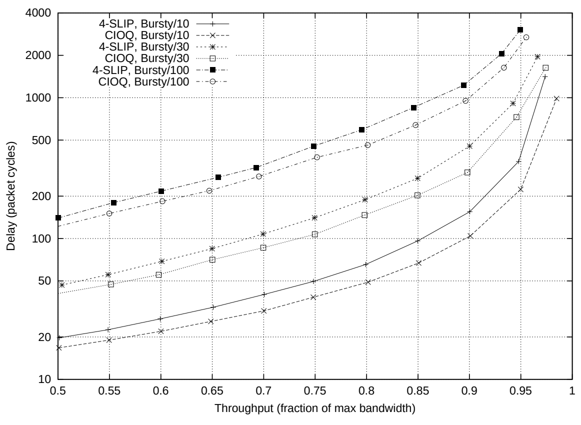
Traffic type	16×16		32×32	
	CIOQ	4-SLIP	CIOQ	5-SLIP
Bernoulli	0.997	0.991	0.997	0.990
IP	0.982	0.971	0.978	0.976
Bursty/10	0.985	0.974	0.984	0.979
Bursty/30	0.974	0.966	0.972	0.967
Bursty/100	0.955	0.950	0.953	0.945

3.4.9 Low-degree traffic

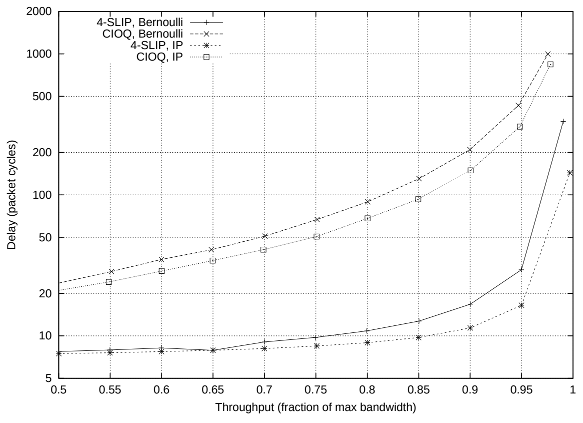
So far, we have used only uniformly distributed traffic models, in which all inputs have identical output probability distributions. If we represent the probability that a packet arrives at input i destined to output j by d_{ij}, then the uniform destination distribution model can be represented by $d_{ij} = \frac{1}{N}, \forall i, j$. This assumption is not necessarily always very realistic: in practice, it will be more likely that each source communicates with a limited set of destinations only. To reflect this, we will employ asymmetric destination distributions that are characterized by each input



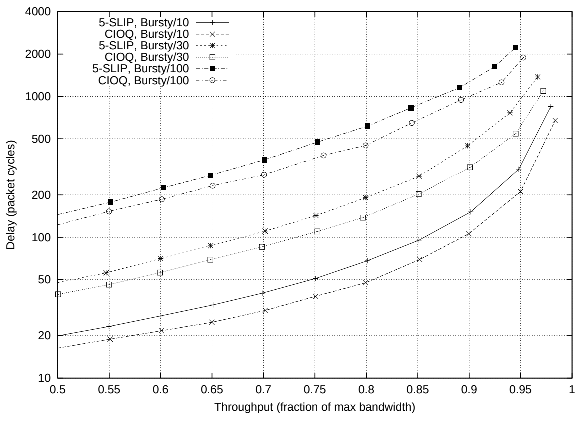
(a) 16×16, Bernoulli and IP traffic.



(b) 16×16, Bursty traffic.



(c) 32×32, Bernoulli and IP traffic.



(d) 32×32, Bursty traffic.

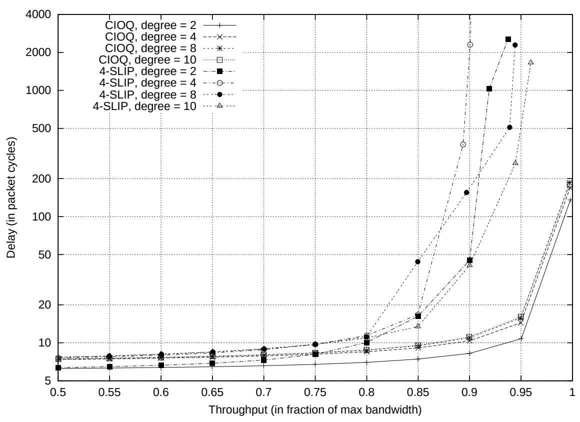
Figure 3.11: Delay vs. throughput comparing the CIOQ architecture with the input-queued architecture using i-SLIP, with two switch sizes and various traffic types.

having traffic only for a limited set of, say, k outputs, with $k \ll N$. These are also referred to as "low-degree" traffic patterns [Goudreau00]. We still require that the traffic is *admissible*, that is, no input or output is oversubscribed: $\sum_{i=1}^{N} d_{ij} \leq 1$ and $\sum_{j=1}^{N} d_{ij} \leq 1$, i.e., there are no overloaded outputs (hot-spots). See Appendix C.4.2 for more details on the destination distribution models.

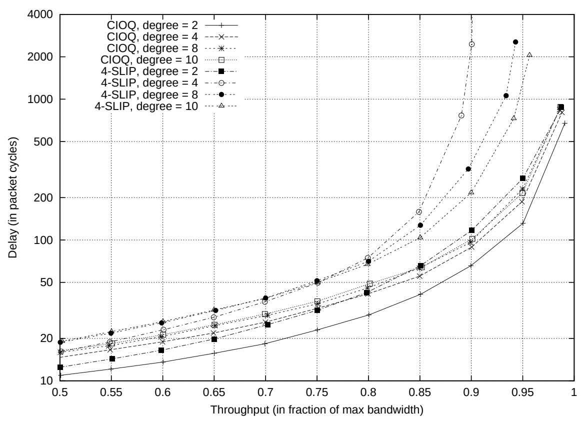
Fig. 3.12 shows the results for $N = 16$, comparing our CIOQ system with $M = 256$ to an input-queued system using 4-SLIP. Four different traffic types, each with degrees of 2, 4, 8, and 10, have been simulated. For each degree, the destination distribution matrix $[d_{ij}]$ has been chosen randomly once,[3] and then used for all simulations corresponding to that degree. k-degree traffic means that each input generates traffic for k outputs, and conversely, each output receives traffic from k inputs. Table 3.6 lists the maximum throughput values corresponding to Fig. 3.12.

The results show that the CIOQ system is largely insensitive to this kind of traffic imbalance:

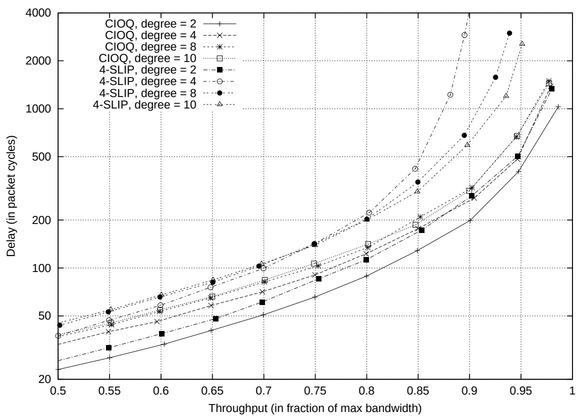
[3]We have attempted to generate the matrices such that each input has k *distinct* destinations.



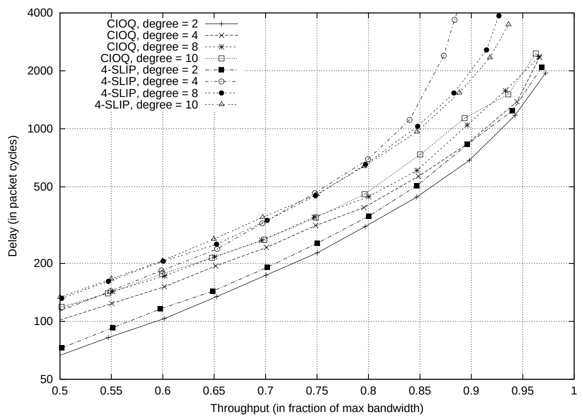
(a) Bernoulli



(b) Bursty/10



(c) Bursty/30



(d) Bursty/100

Figure 3.12: Delay–throughput graphs with low-degree traffic.

the impact on maximum throughput is less than one percentage point for all traffics, whereas delay increases by a factor over the entire load range, that depends on the degree. Traffic of medium degree (4, 8, 10) performs worst in terms of delay.

The impact on performance of the 4-SLIP system is much stronger: maximum throughput degrades significantly, in particular for the medium traffic degrees of 4 and 8. This effect has also been noted in [Goudreau00]; the reason is that the local maxima of the solution space to which heuristic maximum matching algorithms such as i-SLIP will converge are more likely to be sub-optimum solutions when the degree of the bipartite graph is low, i.e., when there are only a few edges incident to each vertex.

3.4.10 Input queue size distributions

So far, we have mainly studied delay–throughput results. Other performance metrics of interest are the delay variance, which is actually more important than absolute delay values for some applications (e.g., CBR traffic), and the average size of the input queues.

Table 3.6: Maximum throughput with low-degree traffic.

Traffic	CIOQ				4-SLIP			
	degree = 2	4	8	10	2	4	8	10
Bernoulli	0.998	0.998	0.997	0.997	0.937	0.901	0.944	0.960
Bursty/10	0.991	0.988	0.986	0.987	0.988	0.901	0.943	0.957
Bursty/30	0.986	0.979	0.977	0.977	0.980	0.901	0.939	0.951
Bursty/100	0.972	0.966	0.966	0.963	0.968	0.894	0.927	0.936

In addition to average delay, we also gather delay variance statistics during the simulation runs. Fig. 3.13a plots both the average delay and standard deviation of the delay as a function of throughput. Fig. 3.13b shows a *delay distribution* graph, which plots the probability $y = P(\mathrm{delay} > x)$ that a packet experiences a delay greater than x. This type of graph is derived from a 102-bucket histogram that records the delay values in the range 0–2048. These graphs give an impression of the spread in the delay values: a "narrow" curve indicates a small spread, and thus small variation, whereas a "wide" curve indicates the opposite.

As real packet buffers are limited, we have to assess packet-loss probabilities. Instead of fixing the input-queue size at given values and measuring packet loss rates, we have taken the approach to simulate with infinite input queues while measuring the input-queue size as a time average: in every packet cycle, the occupancy of all input queues is sampled and stored in a histogram as mentioned above. The occupancy of one input queue is defined as the sum of the occupancies of all VOQs associated with this input. From the resulting histogram a graph such as shown in Fig. 3.13c can be plotted. The y-axis plots the probability that the input-queue occupancy exceeds x at any time. These graphs can then be used to assess packet-loss probabilities given a certain input-queue size. Note the logarithmic y-scales on all of these graphs.[4]

The simulations in this section correspond to a 32×32 system with $M = 1024$. Traffic types are as before.

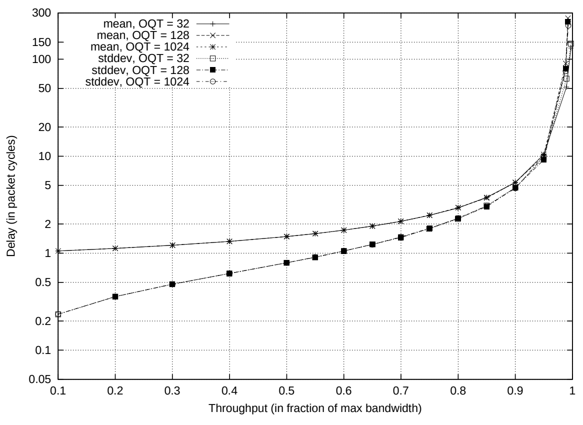
Bernoulli traffic

Fig. 3.13a shows delay–throughput (mean and stddev.) curves for Bernoulli traffic and OQT = 32, 128 and 1024. Up to about 99% throughput (close to saturation), there is no significant difference in performance between the threshold levels.

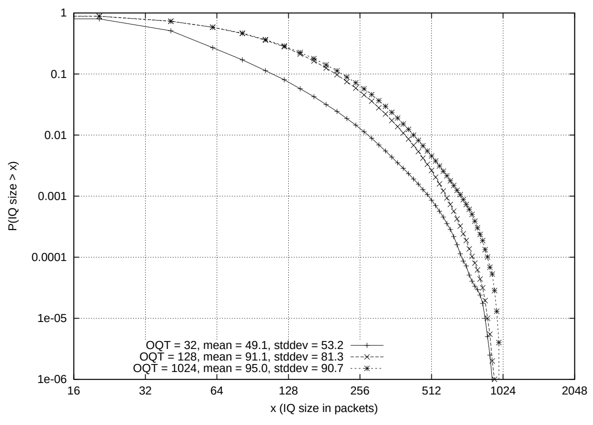
Figs. 3.13b and 3.13c show delay distribution (DD) and input queue size distribution (IQD) curves with the same parameters at an input load of 99%.

Partitioning the memory leads to slightly smaller delay variation than full sharing, and average delay is slightly lower. Even at 99% load, input queue size never exceeds 64 packets. The average IQ occupancy is also slightly smaller with partitioned memory (OQT = 32), than with full sharing (OQT = 1024)

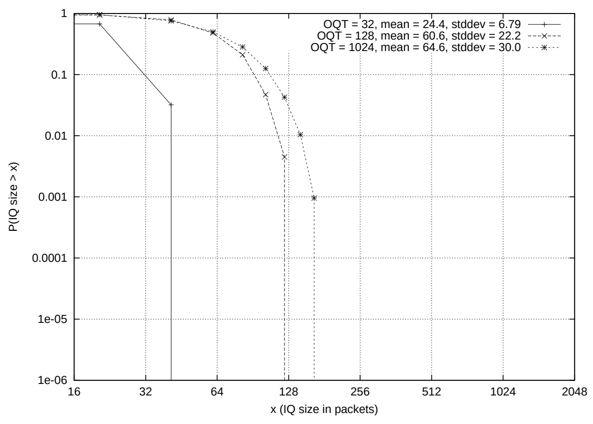
[4]The delay values plotted in these graphs have been adjusted by subtracting a fixed value of 6 cycles from each data point. This is a constant delay component due to the number of entities a packet traverses in the model.



(a) Delay–throughput (mean and stddev.)



(b) Delay distribution at 99% load.



(c) Input-queue size distribution at 99% load.

Figure 3.13: Delay–throughput, delay distribution, and input-queue size distribution curves, Bernoulli traffic, OQT = 32, 128, 1024.

IP traffic

Fig. 3.14a shows delay–throughput (mean and stddev.) curves, while Fig. 3.15 shows DD and IQD curves for IP traffic and OQT = 32, 128, and 1024 at input loads of 70%, 80%, 90%, and 95%. It is interesting to observe the trends in these distributions as the load increases because these trends are seen to reverse as the load increases; in particular, the OQT values that achieve the best DD (mean and stddev.) and IQD (mean and stddev.) are listed in Table 3.7.[5] From both this table and the graphs, we can observe the following trends: at lower loads, sharing the memory (OQT =1204) results in lower delay, lower delay variation and lower input-queue occupancy, compared to partitioning the memory (OQT = 32), see Figs. 3.15a and 3.15b. As the load increases, the curves pull closer together, which is especially obvious in the transition from Fig. 3.15b to 3.15d. As the load increases further, partitioning the memory achieves better

[5]Note that these "optimum" values are taken from the set of simulated values {32,128,1024}, i.e., no optimization to find the absolute optimum has been done.

average delay and input queue size values, see Figs. 3.15e through 3.15h (although variation is not optimal at 95%).

Table 3.7: Optimum OQT values

Ld	delay distr.		IQ distr.		Ld	delay distr.		IQ distr.	
(%)	mean	std	mean	std	(%)	mean	std	mean	std
IP					Bursty/10				
70%	1024	1024	1024	1024	70%	128	128	1024	1024
80%	32	128	128	32	80%	32	1024	1024	128
90%	32	32	32	32	90%	32	32	32	32
95%	32	32	32	32	95%	32	32	32	32
Bursty/30					Bursty/100				
40%	128	128	1024	1024	40%	128	128	1024	1024
60%	128	1024	1204	1024	50%	128	128	128	128
70%	32	32	128	32	60%	32	32	32	32
80%	32	32	32	32	70%	32	32	32	32

Bursty traffic

Figs. 3.14(b,c,d) show delay–throughput (mean and stddev.) curves for Bursty/10, Bursty/30, and Bursty/100 traffic, respectively. Fig. 3.16 shows DD and IQD curves for traffic type Bursty/30 at given loads, each for OQT = 32, 128, and 1024. To save space, we omit similar figures for Bursty/10 and Bursty/100, because they show the same trends. Table 3.7 lists the optimum OQT values that can be derived from these figures.

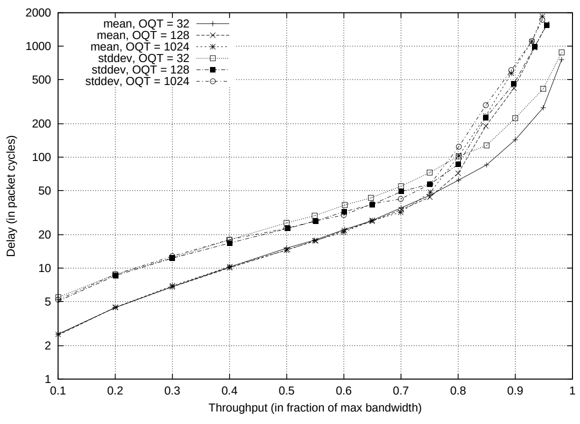
The trend already observed with IP traffic is even more pronounced with bursty traffic: the top two rows of Table 3.7, corresponding to lower loads, predominantly show OQT = 1024 as the optimum value, whereas in the lower two rows, corresponding to higher loads, OQT = 32 appears most frequently. This again indicates that at lower loads full sharing performs better, while at higher loads partitioning achieves superior performance.

Conclusions

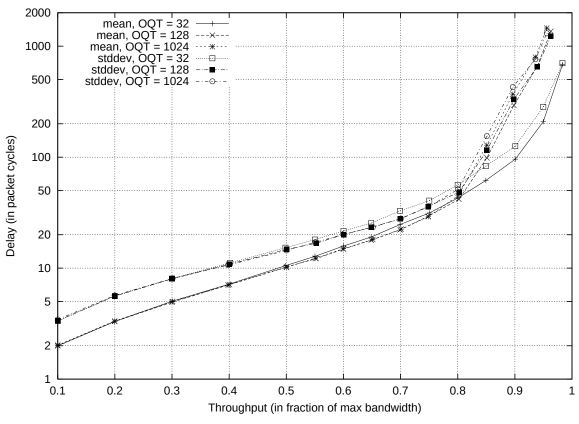
The results presented here clearly demonstrate that there is no single optimum OQT value that results in best performance in terms of throughput, delay, delay variance, and input-queue size under all traffic types and loads.

Low thresholds (M/N) lead to longer input queues at low loads, whereas high thresholds (M) lead to a lower maximum throughput, and consequently to higher delay and longer input queues at high loads.

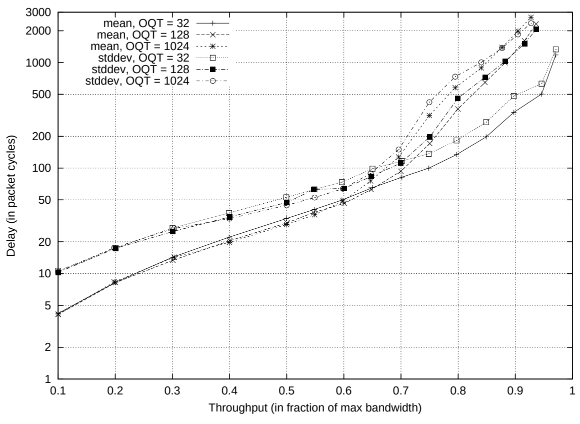
Non-uniformity can occur in the destination distribution (certain inputs are more/less likely to send to certain outputs), and in the loading of inputs and outputs (at a given overall load, certain inputs and outputs are loaded more/less). As the traffic patterns simulated here are uniform in both senses, OQT values are also equal for all outputs. Asymmetric (non-uniform) loading of the switch has not been treated here. With such non-uniform traffic patterns, it is expected



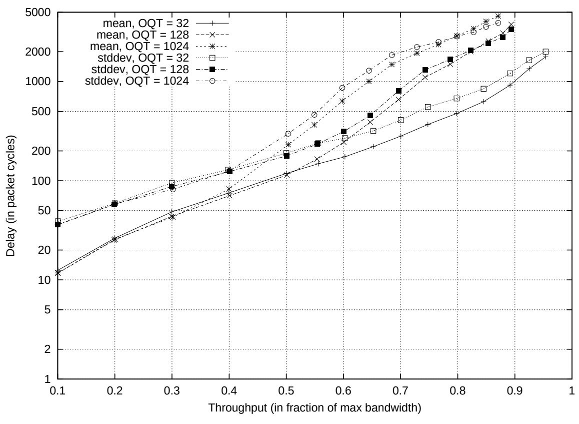
(a) IP traffic.



(b) Bursty/10 traffic.



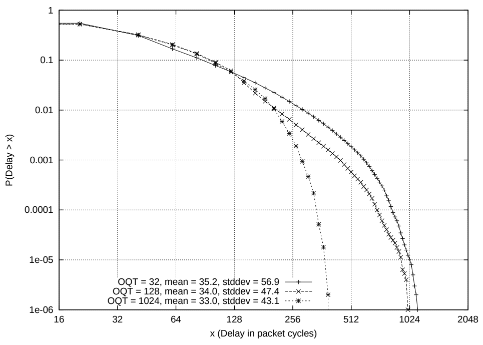
(c) Bursty/30 traffic.



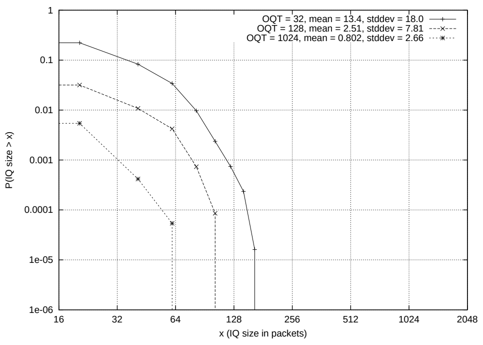
(d) Bursty/100 traffic.

Figure 3.14: Delay–throughput and delay variation (stddev.–throughput) curves, for IP, Bursty/10, Bursty/30, and Bursty/100 traffic; OQT = 32, 128, and 1024.

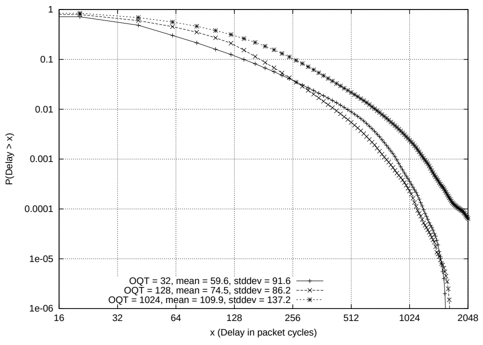
that individual (adaptive) threshold adjusting based on port load performs better than uniform threshold programming [Lee95]. This remains to be studied.



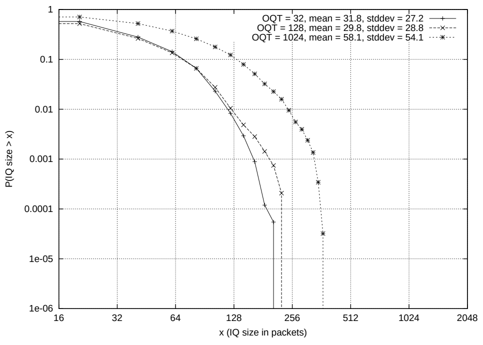
(a) Delay distribution at 70%.



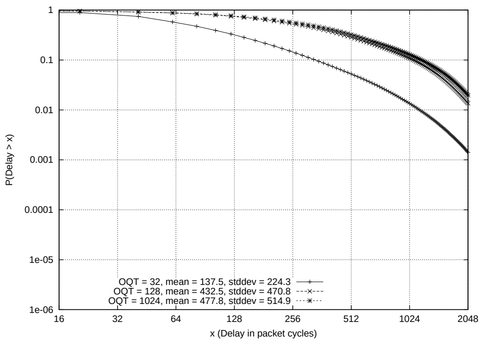
(b) IQ size distribution at 70%.



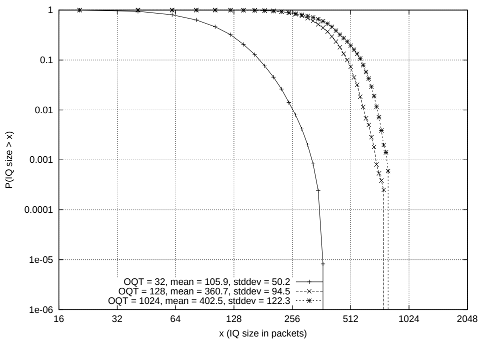
(c) Delay distribution at 80%.



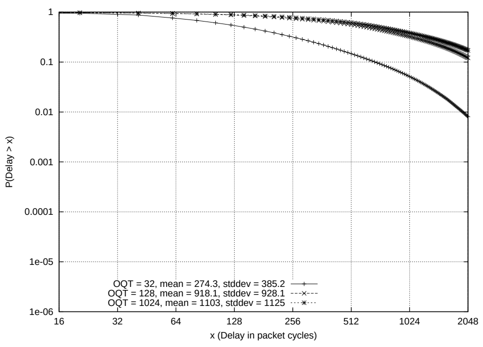
(d) IQ size distribution at 80%.



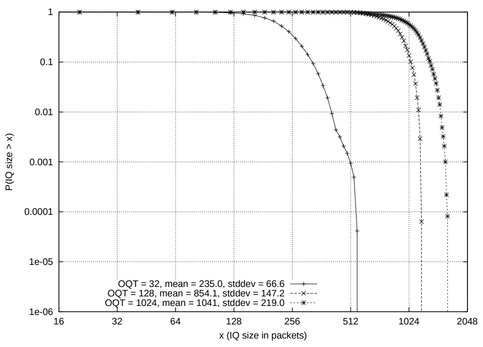
(e) Delay distribution at 90%.



(f) IQ size distribution at 90%.

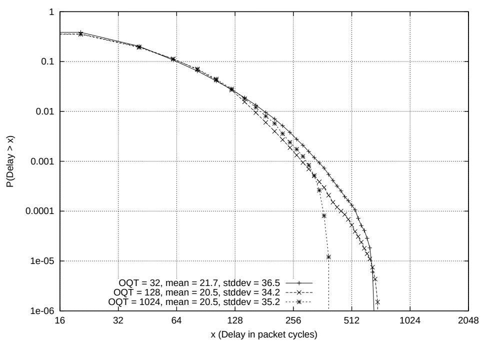


(g) Delay distribution at 95%.

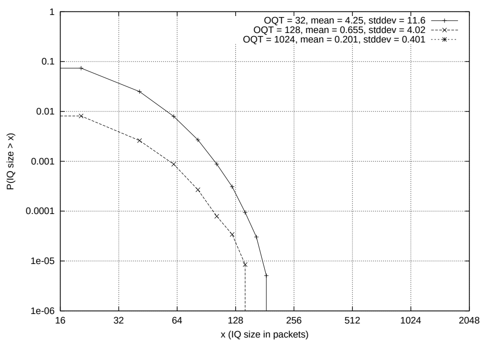


(h) IQ size distribution at 95%.

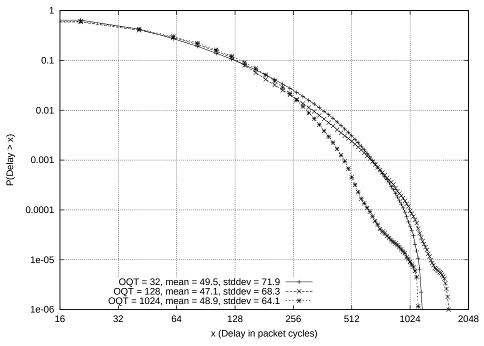
Figure 3.15: **IP traffic**: Delay distribution and input-queue-size distribution curves at loads of 70%, 80%, 90%, and 95%; OQT = 32, 128, and 1024.



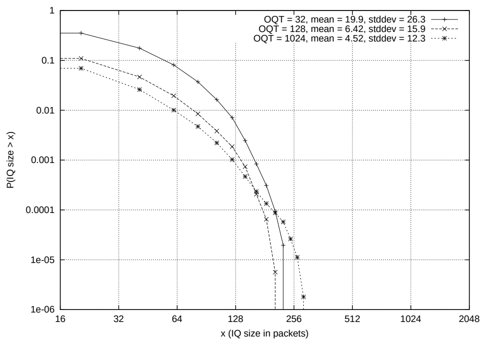
(a) Delay distribution at 40%.



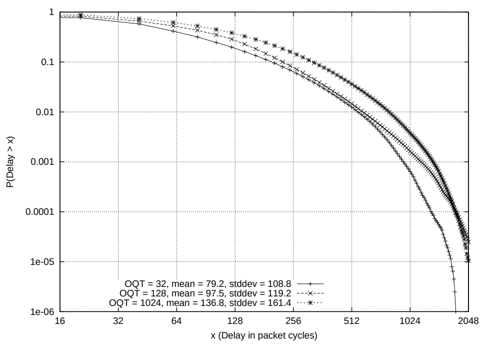
(b) IQ size distribution at 40%.



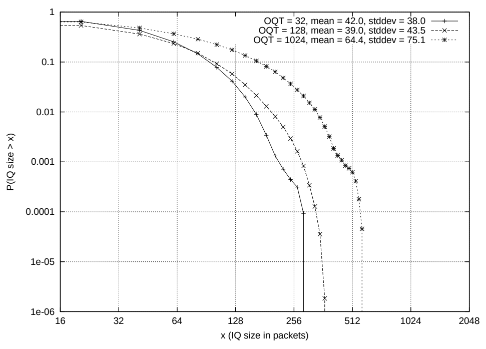
(c) Delay distribution at 60%.



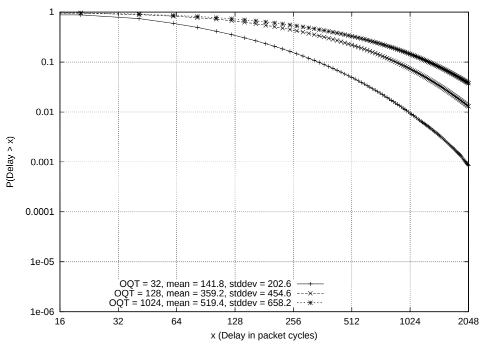
(d) IQ size distribution at 60%.



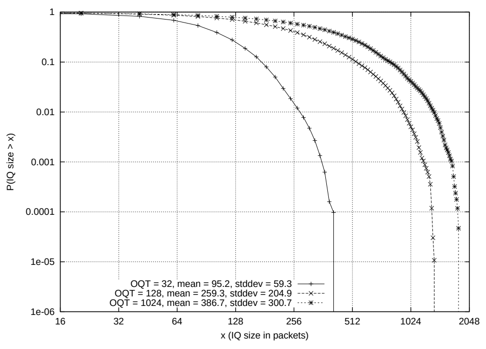
(e) Delay distribution at 70%.



(f) IQ size distribution at 70%.



(g) Delay distribution at 80%.



(h) IQ size distribution at 80%.

Figure 3.16: **Bursty/30 traffic**: Delay distribution and input-queue-size distribution curves at loads of 40%, 60%, 70%, and 80%; OQT = 32, 128, and 1024.

3.5 Queue-Empty Optimization

The flow-control mechanism employed in the proposed system is based on *full* status information, i.e., the signal passed from the switch to the input queues conveys information about thresholds being overrun. This method of flow control stems from the PRIZMA legacy, but there are very good reasons to use it: it allows the switch to operate internally lossless, prevents packets being forwarded to congested outputs, and enables performance enhancement as demonstrated in Section 3.4.2. This is a negative type of feedback: it tells the inputs where *not* to send to, instead of telling them where they *can* send. It seems useful to further guide the inputs in their decision taking by providing positive feedback that tells where packets should preferably be sent to. The principle of work-conservingness (see Definition 2) essentially dictates that output queues that are (almost) empty should be preferred to preserve throughput. This leads us to the addition of a *queue-empty* (QE) flow-control signal from the switch to the input queues that consists of a vector of N bits, one bit per output queue, flagging whether the corresponding queue has underrun a (programmable) queue-empty threshold (QET). This signal is used in addition to the queue-full information, although there is redundancy here: clearly, for all practical purposes, only one of the flags can be active at any given time.[6]

The threshold is programmable to account for the round-trip time between switch and input queues, so that the QE signal is triggered in a timely fashion; in this case, the QE signal is actually a queue-*almost*-empty signal. The VOQ arbiters will favor those outputs that are (almost) empty because these need to be served most urgently to keep the output lines busy. The impact of this optimization will be demonstrated below.

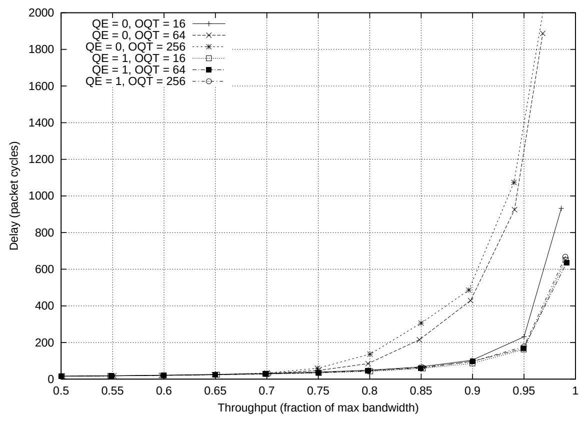
Note that although this optimization improves throughput, it may cause unfairness among input/output pairs. If an input has active flows for a congested and an uncongested output, the uncongested one will always be preferred to keep that output busy; thus, the flow from this input to the congested output is starved. This trade-off between throughput maximization and fairness has also been observed in input-queued VOQ switches: maximum matching algorithms suffer from the same kind of unfairness (see Section 2.6.2).

3.5.1 Performance simulation results

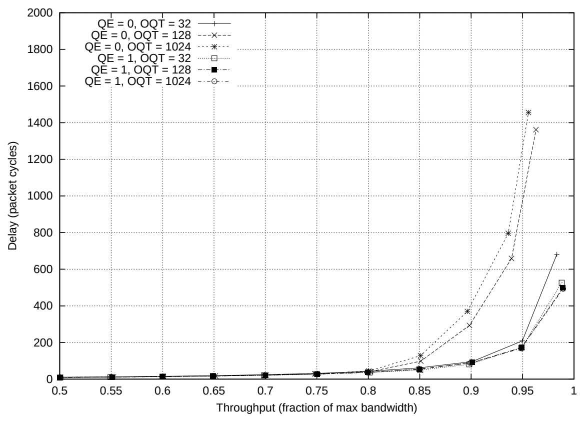
The impact of the proposed optimization on performance is studied in this section. Figs. 3.17a through 3.17f illustrate the improvement in delay–throughput characteristics obtained by using the QE information, for a 16×16 system with M = 256 (a,c,e) and a 32×32 system system with M = 1024 (b,d,f), under Bursty/10, 30, and 100 traffic. The output-queue thresholds have been also varied.

The parameters QE = 0 and QE = 1 in the figures indicate dis- and enabling of the QE feature, respectively. Table 3.8 compares the maximum throughput figures. The QET has been set to 2 in all cases, to take the round-trip time between switch and input adapter into account.

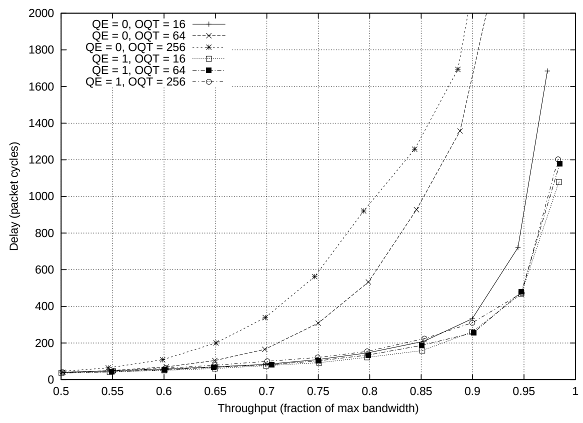
[6]This relates to the idea of using queue *level* information instead of full/empty flags, coded efficiently, so that L levels can be coded in $\log(L+1)$ bits.



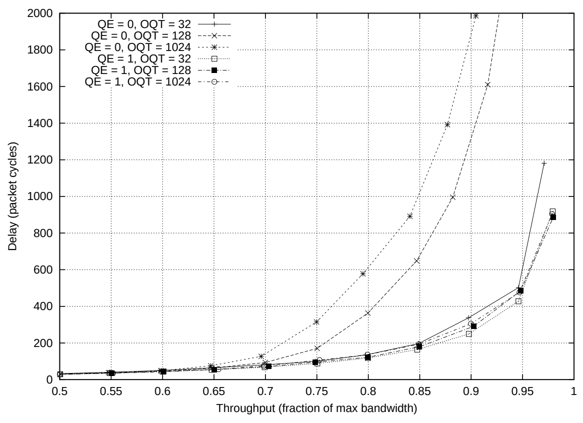
(a) 16×16, Bursty/10.



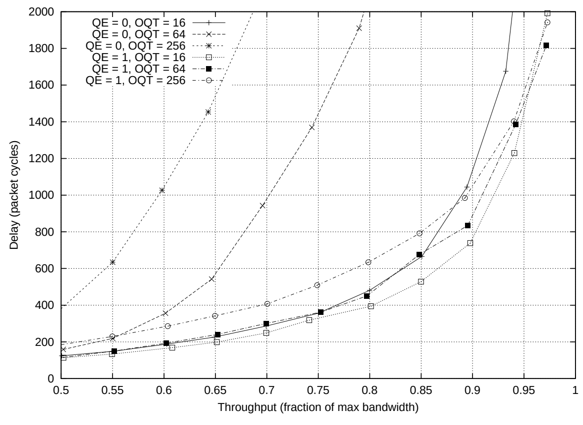
(b) 32×32, Bursty/10.



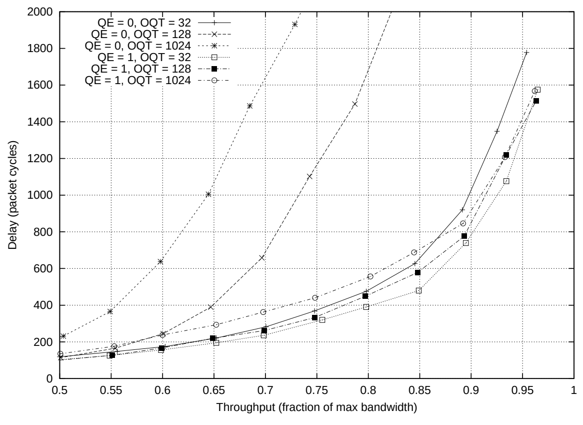
(c) 16×16, Bursty/30.



(d) 32×32, Bursty/30.



(e) 16×16, Bursty/100.



(f) 32×32, Bursty/100.

Figure 3.17: Delay–throughput curves of a 16×16 (a,c,e) and 32×32 (b,d,f) switch system with bursty traffic of three different burst sizes, (a,b) 10, (c,d) 30, and (e,f) 100. Each figure contains six curves, corresponding to output-queue threshold levels of M/N, $4M/N$ and M, with or without QE information.

The use of QE information by the input queues leads to improved performance in all configurations. In particular, when the OQT is high, performance improves drastically. When $\mathrm{OQT} = \mathrm{M}/\mathrm{N}$, the use of QE brings only a small improvement, which mainly manifests itself at high loads: maximum throughput is slightly higher, and latency in the high-load range is lower.

Performance with QE is largely independent of OQT setting because QE prevents, to a great extent, the detrimental output-queue lock-out effect.

Table 3.8: Maximum throughput with queue empty optimization.

Traffic	OQT = M		OQT = $4M/N$		OQT = M/N	
	w/o QE	w/ QE	w/o QE	w/ QE	w/o QE	w/ QE
Bursty/10	0.968564	0.990146	0.968204	0.991363	0.986201	0.990515
Bursty/30	0.941187	0.983153	0.946439	0.984726	0.972702	0.983942
Bursty/100	0.895972	0.972759	0.908165	0.971776	0.953049	0.972814
Bursty/10	0.955708	0.989441	0.963025	0.989269	0.983277	0.987978
Bursty/30	0.927589	0.978719	0.936442	0.979916	0.970919	0.979239
Bursty/100	0.870801	0.96182	0.893508	0.963271	0.953924	0.964607

3.5.2 Conclusion

The use of output-queue-empty information can further improve the delay–throughput characteristics of the proposed architecture, and can alleviate the negative effect of poorly programmed output-queue thresholds. However, the implications on fairness must be taken into account.

3.6 Implementation

In this section we will discuss implementation issues concerning the proposed architecture, in particular regarding the three main components: the switch element, the input queues, and the flow control.

3.6.1 Switch element

Several implementations of shared-memory switches have been presented in literature, such as PRIZMA and ATLAS I (see Section 2.5.2). From an architectural viewpoint, the two main bottlenecks in this switch architecture are first the shared-memory interconnection complexity (each input and each output must be able to reach each memory location, yielding an interconnection complexity on the order of $O(NM)$), and second the output-queue bandwidth (up to $N + 1$ accesses per queue per packet cycle). The split data/control path architecture and its advantages have been described in Section 2.5.2. Here, we will look in more detail at the implementation issues of control section and shared memory.

Control path: output queues and free queue

The control section consists mainly of the N output queues and the free queue. Each output queue must be able to store N addresses and release 1 in each packet cycle ($N + 1$ accesses/cycle). However, the bandwidth requirement on the free queue is even more stringent: it must be able to store N and release N addresses in each packet cycle ($2N$ accesses/cycle). Hence, if we can implement the free queue, the output queues pose no further problems.

Let us consider the bandwidth requirement on the free queue, and compute the required access time to achieve this. Given the number of ports N, minimum packet size L (in bits), and port speed B (in b/s), the free queue must be able to store and fetch one address every L/NB seconds. Taking PRIZMA-EP as an example, with $N = 32$, $L = 32$ bytes $= 512$ bits, and port speed 2 Gb/s, this yields an output queue access time of 4 ns, provided the memory can be written to and read from simultaneously. For EP, $M = 2048$, so that, to obtain the desired bandwidth, an 11-bit-wide dual-ported SRAM with an access time of 4 ns is required.

Currently, access times under 2 ns are not feasible, so that higher-bandwidth switches will require increased parallelism in the output queue implementation. To implement a 1 Tb/s switch in a 32×32 @ 32 Gb/s configuration with a targeted minimum packet size of 64 bytes, an output queue access time of 0.5 ns would be required, which is not feasible. With a more realistic access time of 2 ns, 4 addresses of 10 bits each (assuming $M = N^2 = 1024$) must be handled in parallel, using either multi-ported RAM or parallel RAM banks, which is feasible in current technologies.

Note that the control bandwidth required here equals $(N(N + 1)B(\log M))/L$, or $(2N(N + 1)B(\log N))/L$ with $M = N^2$. This is comparable to the control bandwidth required in an IQ VOQ switch or a CIOQ switch with limited speed-up (Section 2.8), which equals $(N(N + \log N)B)/L$, the difference being that the latter must be passed across chip boundaries, whereas the former is on-chip.

Shared memory

The required read and write bandwidths from and to the shared memory pose another great implementation challenge.

The original PRIZMA approach [Denzel95] is to implement the memory as an array of M independent memory locations, which can be accessed by all inputs and outputs in parallel. To achieve this, each input must be able to access each memory location, which leads to an interconnection structure of complexity $O(NM)$, at both the input and output side of the memory array. Returning to the targeted 1 Tb/s example, with $N = 32$ inputs, a shared memory of $M = N^2 = 1024$ packets, and a port speed of 32 Gb/s, again assuming a memory cycle time of 2 ns, the data path must be 64 bits wide (32 Gb/s * 2 ns). In this case, the total number of wires to interconnect the memory array to inputs and outputs exceeds 4 million, which is certainly not feasible. To reduce the width of the data path per chip, the speed expansion concept can be applied (see Section 6.3.1), which allows the data path to be sliced over a number of identical switch chips operating in parallel. A wiring study that is outside the scope of this thesis has shown that a 16-bit-wide data path is feasible in this configuration in the given technology, so that a speed expansion factor of 4 is required.

This approach clearly does not scale well to larger port numbers, because the number of wires grows as $O(N^3)$. It scales fairly well in port speed, because the speed of each individual memory scales linearly with port speed, and is not dependent on the number of ports.

The PRIZMA-E and EP implementations (16×16 @ 2 Gb/s and 32×32 @ 2 Gb/s, respectively), illustrated in Fig. 3.18, deviate from the original PRIZMA architecture, mainly for cost reasons. Here, the memory bandwidth is obtained by the traditional, yet unscalable, shared-bus approach. The memory is organized internally in two parallel banks, each 20 bytes wide, with 512 (2048 for EP) addresses per bank. These memories are accessed through two 20-byte-wide buses (the bus width equals the memory width), which are clocked at 8 ns (4 ns for EP). The input controllers receive access to these buses in a RR fashion; every time an input controller receives its turn, it writes 16 to 20 bytes at once (depending on the currently programmed packet size) to each bank, using the address provided by the address manager. The output side operates in a similar fashion, with the output queues providing the appropriate read addresses.

Clearly, this approach scales neither in number of ports nor in port speed because the bus bandwidth must equal their product.

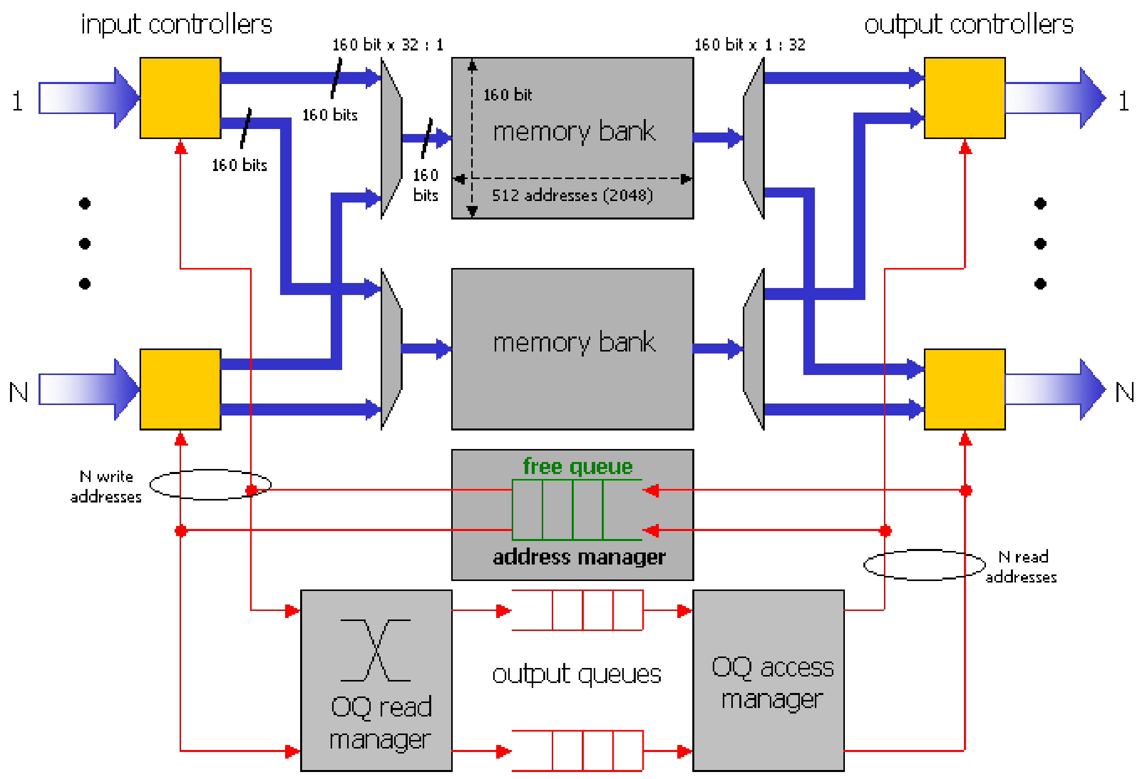


Figure 3.18: PRIZMA-E and EP implementation, based on a wide shared bus.

The insights obtained in this chapter suggest that to obtain best performance the available memory does not have to be shared among all inputs and/or outputs . This suggests that considerable savings in implementing the memory can be achieved by physically partitioning the memory on chip, thus significantly reducing the complexity of the input and output routing trees. Chapter

6 will elaborate on this subject.

Other issues: Cost, die size, IO links, power, and packaging.

Of course, there are other physical limits to the throughput achievable with a single-chip switch, which we will briefly review here. For one, the silicon die size is a very important cost factor. Beyond a certain die size (about 15×15 mm^2), the process yield quickly drops, resulting in a much higher per-chip production cost. Furthermore, the amount of bandwidth available through the chip's (electrical) pins is limited. Beyond a certain size, the chip package cost mounts very rapidly with increasing pin count, and packages with more than approximately 1000 pins are not economical for this application. Current high-density electrical IO technologies peak at data rates of about 2 to 2.5 Gb/s per differential wire pair. Assuming that half the pins are required for other functions, such as V_{dd}, ground, flow control, debugging, etc., a 1000-pin package can provide around 625 Gb/s of aggregate raw data bandwidth, which corresponds approximately to a 32×32 @ 10 Gb/s switch. Therefore, in order to scale to terabit/s or higher throughput, multi-chip solutions are required. Clearly, when faster high-density links become available higher per-chip throughput can be achieved.

Although die size and pin count are important, an even more crucial factor is power dissipation, both at the chip as well as the system level. The total power consumption of one switch chip should stay within the allocated power budget given by both system and technology constraints.

An important contributor to the power dissipation of a switch chip are the IO links (transmitters and receivers), the memories, and the control section. More and faster links, faster memories, and a faster control section all increase power dissipation. As link speeds and CMOS integration density increase, the power budget is sure to become the limiting factor in the throughput achievable with a single chip, so that VLSI chip design that optimizes for power will become increasingly important.

A final factor to be considered is the switch system implementation at the box level. For commercial products, a switch system, i.e., a redundant switch fabric plus N adapters (line cards), must often meet certain standard form factors determining card, backplane, shelf, and rack dimensions. These physical dimensions put hard limits on how many chips fit on a card, how many cards in a shelf, how many shelves in a rack, and how many racks in a room. The size of a card also determines, given the available connector density and link speed, how much bandwidth can be supported by a single switch card, regardless of the size or throughput of the switch chip itself. A system that does not fit in a single rack is drastically more complex and costly than one that does.

3.6.2 Flow control

Flow control plays an important role in the proposed architecture. It has a dual role: first, to prevent internal packet loss (memory grant), and second, to provide the VOQ arbiters with information to make better decisions (output-queue grant). However, this flow control consumes bandwidth between switch and input queues. Assuming P traffic classes, $P(N+1)$ bits of information should ideally be transported from the switch to *each* input queue in every packet cycle: PN bits of output-queue grant (one bit per output queue per traffic class), and P bits of shared-memory grant (one bit per traffic class). Theoretically, the flow-control bandwidth equals just $P(N+1)$ bits per packet cycle. However, this information must be distributed to all input adapters. Hence, practically, the total bandwidth required for flow control equals $NP(N+1)$ bits per packet cycle.

For PRIZMA-EP, with packets of 64 bytes, 32 ports, 4 traffic classes (strict priorities with guaranteed bandwidth provisioning), and a port speed of 2 Gb/s, the overhead bandwidth equals $\frac{NP(N+1)}{64 \cdot 8/2 \cdot 10^9} = \frac{32 \cdot 4 \cdot (32+1)}{256 \cdot 10^{-9}}$ = 16.5 Gb/s, which amounts to over 25% of the data bandwidth.

In the PRIZMA implementation, this added complexity is tackled as follows: first, the memory grant is put on dedicated chip pins (4 pins, one per priority) connected directly to the input queues. This is done because the latency of the memory grant is critical to guarantee lossless operation. Second, the output grant is transported *in-band*; it is inserted into the free header bitmap space of the *outgoing* packets, which offers just enough space to transport N times N bits of output grant. The information per priority is time-multiplexed in a RR fashion: in each cycle, the OQ grant for one priority is transmitted, in the following cycle for the next priority, etc. As a result, there is a certain, non-negligible latency in the output-queue grant transmission,[7] but the bandwidth required is reduced by a factor of P.

As the output-queue grant is inserted in the outgoing packets, it travels in the wrong direction. However, because the input adapters, which contain the input queues, and the output adapters are typically located on the same physical board, it is not a problem to relay the output-queue grant, which arrives at the output adapter, back to the input adapter.

3.6.3 Input queues

Typically, the input queues with their VOQ arbiter are located on the input adapters. Such an adapter can perform a wide variety of functions, but here we are interested only in these functions required to implement the proposed architecture. Section 4.6 describes in detail how the input queues can be implemented using a shared memory with split data/control path approach, similar to the internal PRIZMA architecture. The implementation of the various flavors of VOQ arbiters (RR, LRU, OQF) has been mentioned in Section 3.2.3.

To be able to sort packets by destination, the adapter must be able to determine the output port corresponding to the packet's destination, and prepend the correct routing tag to the packet. If source routing is used, only a swap of tags is required, otherwise a routing-table look-up must be performed.

Other functions such as segmentation, reassembly, packing, load balancing, traffic shaping, etc., are beyond the scope of this dissertation.

3.7 Conclusions

We have proposed a novel way to build robust and cost-effective switch systems by combining the concepts of VOQ at the input and output buffering at the output, thus simultaneously reducing the complexity of scheduling at the input to $O(N)$ and achieving high, largely traffic-insensitive performance. The basic premise is that the output-buffered switch element is not

[7]In PRIZMA-E, the output-queue grant is also available on chip pins, namely a 16-bit-wide bus that time-multiplexes the information per priority, with a 17th bit to synchronize on priority 0.

used as a buffer to store large bursts but rather that those bursts are stored in the input queues and that the switch element guides the VOQ arbitration at the input side by means of output-grant flow control to achieve maximum throughput. In contrast to purely input-buffered architectures, there is no need for a centralized scheduler, whereas performance is better. Extending the output-queue grant with output-queue empty information enables the VOQ arbiters to keep a maximum of switch outputs busy at all times, thus achieving very high throughput, at the expense of fairness. The system's performance has been demonstrated by means of simulations. An implementation of the switch element is available today in the form of the PRIZMA-E and PRIZMA-EP single-chip switches [Colmant98c, PRIZMA-E], which can be utilized to build switch systems in a modular, flexible fashion, owing to their expansion-mode capabilities.

From the results presented in this chapter, we draw the following conclusions:

- Combining VOQ at the input with an output-buffered switch element allows the classical central VOQ scheduler of superlinear complexity to be replaced by simple arbiters of complexity $O(N)$ located at each input.
- We recommend an OQF VOQ arbitration discipline to avoid starvation and minimize HoL waiting times. Practically, little difference in performance was observed between RR, LRU, and OQF policies.
- Maximum throughput under uniform traffic has been shown to be 100%. We have shown that a CIOQ switch with limited output buffers can never be strictly work-conserving.
- The proposed system achieves high throughput and low burst sensitivity by effectively separating the contention resolution and buffering functions such that the switch performs the former and the input queues cope with burstiness. A shared-memory size equal to N^2 packets is optimum in terms of cost and performance.
- A system using VOQ input queues and per-destination flow control far outperforms a similar system using FIFO input queues, and shows little sensitivity to bursty traffic conditions.
- Using FIFO input queues, *increasing* the degree of memory sharing improves delay–throughput characteristics. On the other hand, when using VOQ input queues, *decreasing* the degree of sharing improves performance, and full partitioning of the shared memory turns out to be optimum in this case.
- There exists no single optimum OQT level that results in best performance in terms of throughput, delay, delay variance, and average input-queue size for all types of traffic and input load levels, which suggests that adaptive threshold programming schemes may be employed to optimize performance. This requires further study.
- Compared to an input-buffered architecture using VOQ and an iterative SLIP scheduler, the proposed architecture offers better performance, especially under high load. Furthermore, it performs significantly better under more realistic, asymmetric (so-called "low-degree") load conditions.
- Output-queue-empty information has shown to be an efficient enhancement to queue-full-based flow control, by providing an indication to the input queues as to which outputs

should be served most urgently in order to keep all output lines busy. It offers good performance when used by itself, but optimum performance is obtained when used in conjunction with queue-full-based flow control. This throughput optimization is obtained at the expense of fairness.

- Practical implementation is limited by shared-memory bandwidth and chip IO bandwidth.

We argue that the system proposed here is a viable alternative to the purely input-buffered architectures that have received much attention recently, where the small output-buffered switch element effectively takes on the role of the scheduler in the input-buffered architecture, thus eliminating the need for complex scheduling algorithms, while offering high, robust performance, and enabling all traditional QoS mechanisms such as WFQ, GPS, etc., which require an output-buffered system.

A US patent application [Colmant98a] has been submitted on the proposed architecture. Some of the results in this chapter have previously been published in [Minkenberg00b, Minkenberg00d].

Chapter 4

Multicast

4.1 Introduction

As more and more point-to-multipoint and multipoint-to-multipoint applications are being deployed, the ability to handle these types of communication efficiently becomes increasingly important. Here clearly the duplication of data from one or more sender(s) to multiple receivers is involved; this is termed *multicast*, as opposed to *unicast*, which involves only one sender and one receiver. The elements that are instrumental in implementing efficient multicast are a network's switching nodes. Instead of setting up separate connections to each multicast destination and duplicating information at the source, the switching nodes can duplicate the information at points as close to the destinations as possible, thus reducing the overall network load [Phillips99], which is a main benefit of multicast switching. Therefore, also hardware support for multicast traffic is required at the packet-switch level. In this chapter we consider the problem of how to implement multicast scheduling in the combined input- and output-queued (CIOQ) architecture introduced in Chapter 3.

The remainder of this chapter is organized as follows: Section 4.2 gives an overview of previous work on multicast scheduling for both input- and output-queued switch architectures. Section 4.3 presents our switch architecture, and Section 4.5 introduces a suitable multicast scheduling algorithm. A possible implementation is discussed in Section 4.6, and performance simulation results are presented in Section 4.7. Section 4.8 discusses possible alternatives for input-queued architectures, and conclusions are given in Section 4.9.

4.2 Overview of Related Work

First, let us define the *fanout* F of a multicast packet as the number of output ports it is destined to.[1] We distinguish an arriving *input packet* from the *output packets* it generates; an input packet with a fanout of F generates F output packets. For the effective output load L_e on the switch relation (4.1) holds:

$$L_e = (p_m \overline{f_m} + (1 - p_m))L_i, \tag{4.1}$$

[1] This is also sometimes referred to as the *multiplicity* of a multicast packet.

where p_m is the probability that an arriving packet is a multicast packet, $\overline{f_m}$ is the average fanout of a multicast packet, and L_i is the offered input load. We consider a switch of size $N \times N$.

Early packet-switch designs based on Batcher-Banyan networks employed copy networks, prepended to the routing network, to obtain multicast functionality, e.g. [Turner88, Lee88a, Lee88b, Liu97]. A method to implement multicast in the classic Knockout switch (see Section 2.4.1) is described in [Eng88b]. In this section, we review more recent packet-switch architectures and scheduling algorithms for multicast traffic based on both input- and output-queued architectures.

4.2.1 Output-Queued Switches

In a purely output-queued switch, multicast can be performed almost trivially. Upon arrival of a multicast packet, it is simply duplicated to every output queue it is destined for. This scheme is also referred to as *replication at receiving* (RAR). However, there is a significant drawback to this approach as each incoming packet may have to be duplicated up to N times, which is a waste of internal memory bandwidth. This problem can be solved elegantly by adopting the *replication at sending* (RAS) approach, which can be achieved with a shared-memory switch architecture having a split data/control path architecture, where the output queues handle only pointers to the actual data stored in a memory shared by all output queues [Engbersen92, Denzel92, Denzel95] (see also Section 2.5.2, Fig. 2.10). Thus, the packet data need only be stored once, while the pointer[2] to the data is duplicated. With each shared memory location (pointer) an occurrence counter of size $\log_2 N$ bits is associated, which, upon receipt of a packet at the associated memory location, is initialized to the number of ports the packet is destined to, and is decremented each time a copy of the packet leaves the switch. When the counter reaches zero, that is, the last copy of the packet has been transmitted, the pointer is released.

The behavior of output-queued switches using either RAR or RAS with multicast traffic was studied extensively in [Chiussi97].

4.2.2 Input-Queued Switches

The bandwidth through the shared memory of an output-queued switch must equal N times the individual port speed, which poses significant implementation concerns at high line rates. Because of this, input-queued switches have gained popularity in recent years. The performance limitations of FIFO-queued crossbar-based switches have largely been overcome by applying techniques such as VOQ, combined with centralized scheduling to achieve good throughput, see for example [McKeown99a].

VOQ entails the sorting of incoming packets at the input side based on the packet's destination output. This arrangement is fine for unicast traffic, but does not fit well with multicast; for example, in which queue would one store an incoming multicast packet that has F different destinations? The generally accepted solution is to add an $(N + 1)$-th queue at each input that is dedicated to multicast traffic, see Fig. 4.1. This raises two new problems: (1) how to schedule packets from the N multicast queues, and (2) how to integrate multicast

[2]The size of the pointer equals $\log_2 M$ bits for a shared-memory size of M packets.

with unicast traffic in an efficient and fair way. Several publications study the former problem [Hayes89, Hui90, Hayes91, Chen92, Chen94, Schultz94, Haung96, Mehmet96, Prabhakar95, McKeown96b, Prabhakar96, Prabhakar97, Chen00a], but leave the latter untouched. A recent publication [Andrews99] attempts to shed some light also onto the latter problem.

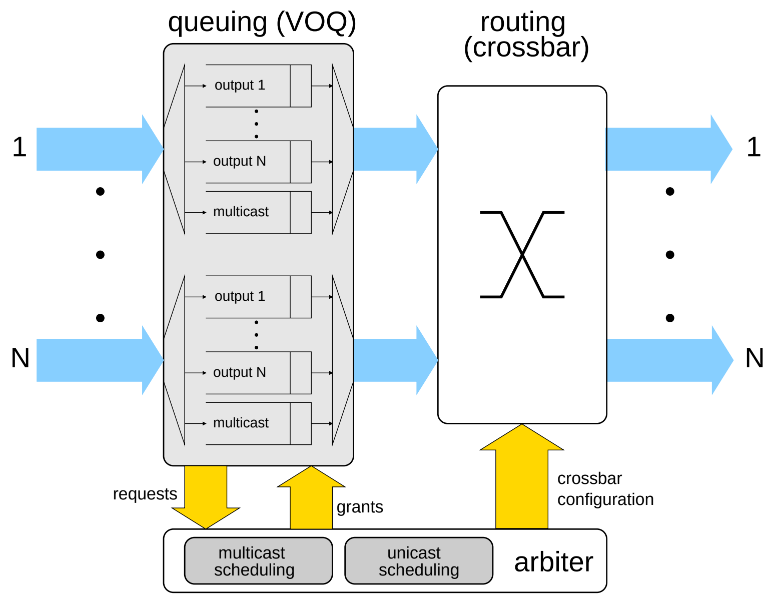


Figure 4.1: In VOQ switches, a dedicated multicast queue is used. In each packet cycle, the centralized arbiter collects requests from all inputs, performs its scheduling algorithm, informs the inputs by means of the grants, and configures the crossbar accordingly. The selected packets are removed from the input queues and transmitted, completing the cycle. Typically, separate scheduling algorithms for unicast and multicast traffic are used.

Hui and Renner [Hui90] provide an overview of multicast scheduling strategies for input-buffered switches with FIFO queues. They distinguish between *unicast* and *multicast service*, the former entailing sequential transmission to each of a multicast packet's destinations, whereas in the latter case multiple destinations can be served simultaneously. They also introduce the notion of *fanout splitting* for the multicast service case; this means that a multicast packet may be transmitted over the course of multiple timeslots, until all of its destinations have been served. Fanout splitting is also sometimes called *call splitting*. The opposite is *one-shot scheduling*, where all destinations have to be served simultaneously. Fanout splitting offers a clear advantage over one-shot scheduling because HoL blocking is reduced, which is confirmed in [Chen92] and [Prabhakar97]. Multicast service is clearly preferable to unicast service for a multitude of reasons, the main one being that the latter is wasteful of bandwidth towards the switch because a packet with a fanout of F must be transmitted F times across the input link, resulting in poor utilization and large delays. Hui and Renner come to the conclusion that an FCFS[3] service with fanout splitting is best in terms of throughput, delay, and fairness. The case of *random* selection with fanout splitting is studied extensively in [Hayes91], whereas in [Chen94] a cyclic priority reservation scheme without fanout splitting is employed.

In [Chen92], the hybrid *Revision* scheme is proposed, which combines one-shot scheduling and fanout splitting to obtain better delay–throughput performance than the individual policies.

[3]FCFS = First Come, First Served.

First, a round of one-shot scheduling is performed, selecting a set of HoL packets that can all be served completely in this cycle. Then, a fanout splitting scheme is applied to select some copies of other HoL packets that do not conflict with the packets already selected. This scheme clearly offers better throughput than one-shot scheduling because it allows more packets to be served, but it also outperforms a general fanout splitting policy, because the first round of one-shot scheduling ensures that as many new packets as possible appear at the HoL, thus potentially increasing throughput. This idea closely resembles the residue concentration principle introduced later by McKeown and Prabhakar [Prabhakar95] (see below).

Chen et al. [Chen00a] propose a window-based approach, similar to the approach described in Section 2.3.2 for unicast traffic. Although this approach offers clear performance improvements over approaches that only consider HoL packets, it still suffers from the drawback that the effectiveness of a given window depth diminishes rapidly as traffic becomes burstier, which is evidenced by their performance simulation results. Furthermore, it requires random access to the input queue, and the implementation complexity of the proposed scheduling algorithm is on the order of $O(N^2L)$, where L is the window depth. An earlier approach based on windowing using a content-addressable FIFO (CAFIFO) is presented in [Schultz94, Schultz96].

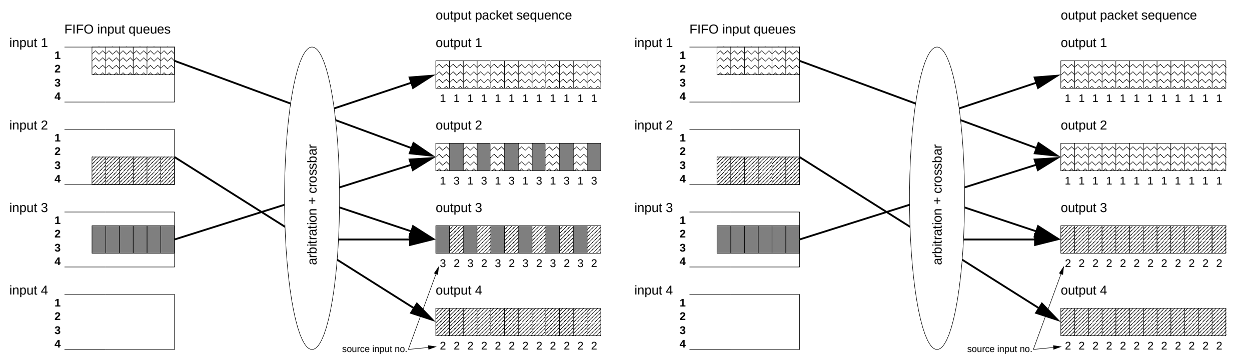
A different approach is introduced by Prabhakar and McKeown [Prabhakar95, McKeown96b, Prabhakar96, Prabhakar97]: the concept of *residue concentration*. At any given time only the HoL packets of all multicast queues are considered. Given their destination sets, one can compute the set of output packets that *cannot* be served. This set is called the *residue*. Two diametrically opposing scheduling policies are proposed: *Distribute* and *Concentrate*, illustrated in Fig. 4.2. A distributing policy attempts to place the residue on *as many inputs* as possible, whereas a concentrating policy attempts to place the residue on *as few inputs* as possible. An intuitive explanation is given why concentrating the residue is optimum; basically, it increases the likelihood that new packets are brought forward to the head of line, thus bringing forward more new work and potentially increasing throughput. Fig. 4.3 shows performance curves of the Concentrate algorithm. It performs well for uncorrelated arrivals, sustaining up to 90% throughput, but suffers a throughput loss for correlated arrivals owing to HoL blocking (down to 75% for Bursty/16, 72% for Bursty/64). However, it has been noted that Concentrate may lead to starvation of inputs, which prompted the development of alternative algorithms such as TATRA, based on a Tetris-like approach, and WBA, which allows a tradeoff between fairness and optimum throughput (residue concentration). However, Concentrate performs best in terms of delay–throughput, and therefore we will use this as a best-case representative of the class of scheduling algorithms for input-queued architectures with multicast FIFOs.

4.2.3 Open Issues

Despite the advances made with respect to multicast scheduling, the input-queued architectures presented above face several problems:

HoL Blocking

The FIFO organization of the multicast queue is prone to HoL blocking, in particular under heavy multicast load. Although concentrating algorithms try to minimize the impact by quickly serving entire HoL packets, they can never eliminate it. If it were possible to somehow apply



(a) Distribute (b) Concentrate

Figure 4.2: The residue here equals the set of outputs $\{2, 3\}$. The Concentrate policy concentrates the residue on as few inputs as possible, here on input 3. Note that under the traffic scenario shown, input 3 will be starved indefinitely under the Concentrate policy. The Distribute policy can alternate placing the residue on inputs $\{1, 3\}$ and $\{2, 3\}$. However, in general the Concentrate policy leads to better performance.

the VOQ arrangement used for unicast traffic also to multicast traffic, HoL blocking would be eliminated completely.

Multicast and Unicast Integration

Integration of multicast and unicast traffic and fairness between multicast and unicast traffic are issues that have gone largely untouched. Ideally, no artificial distinction should be made between the two types of traffic with respect to either the queuing discipline or the scheduling discipline in order to ensure fairness among all input/output pairs regardless of traffic type. In [Andrews99], an integration method is presented to “fill up holes” in the multicast schedule with unicast traffic. However, as the multicast load increases unicast receives less and less service; in fact, the aggregate throughput decreases, which is of course highly undesirable.

Memory Bandwidth

Packet switches that rely solely on output queuing do not scale well to high data rates because of the high memory bandwidth requirement. Implementations that use a high degree of parallelism can achieve the desired bandwidth, but limit the amount of memory that can be integrated on a single chip, thus potentially leading to high packet-loss rates and highly traffic-dependent performance.

Taking these aspects into account, Section 4.3 will present a new architecture and scheduling algorithm, based on a CIOQ switch system, that solves the issues indicated above, while offering superior performance compared to existing approaches.

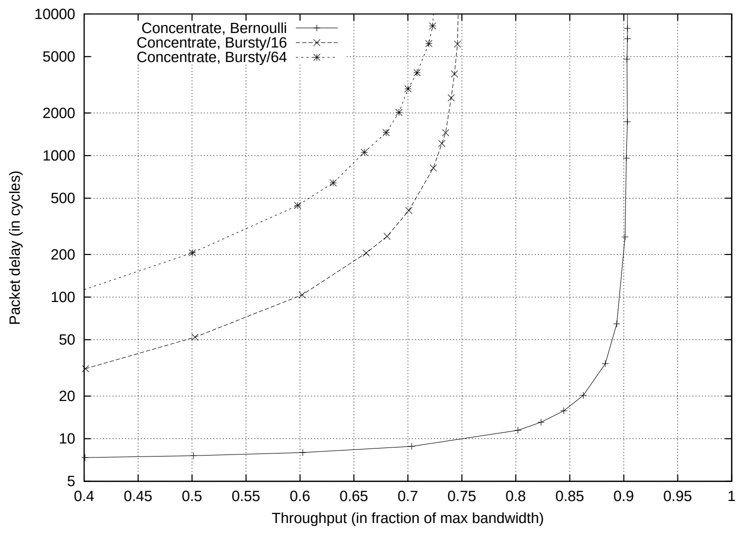


Figure 4.3: Delay–throughput characteristics of the Concentrate algorithm for a switch size of 8×8, uncorrelated arrivals (Bernoulli) and correlated arrivals with geometrically distributed bursts of 16 and 64 packets on average (Bursty/16, Bursty/64). Destination vectors are uniformly distributed over all $2^8 - 1$ possible ones.

4.3 Architecture

Fig. 4.4 depicts a CIOQ packet-switch system comprised of input queues sorted per output (VOQs) combined with an output-buffered switch element, as introduced in Chapter 3. Please refer to Section 3.2 for a precise description of all aspects of the architecture and its operation.

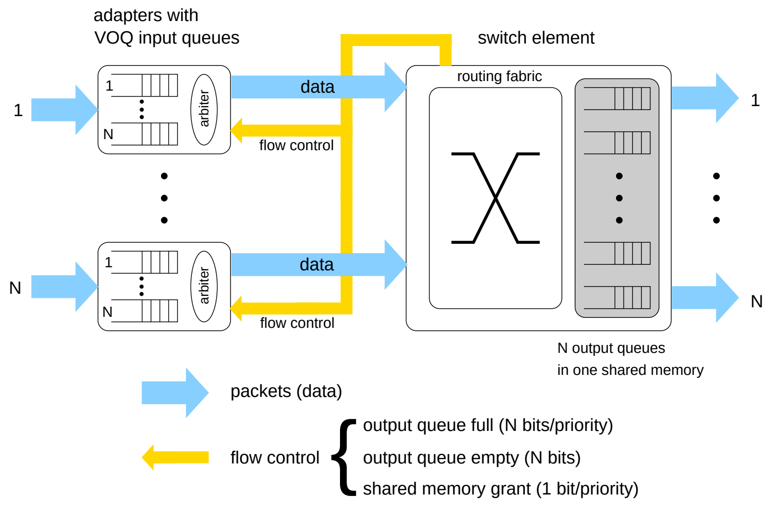


Figure 4.4: A switch system using combined input and output queuing.

An arriving packet is stored in the VOQ corresponding to its destination port. Recall that the output-buffered switch enables distributed scheduling of the VOQs, so that each input arbiter can make a decision independent of all others, and that each arbiter selects, according to a selection policy (e.g., OQF or RR, see Section 3.2.3), a non-empty VOQ for which an output-queue grant has been received thus simultaneously assuring fairness across all VOQs *and* preventing HoL blocking. The packet selected is then forwarded to the switch, which will route the packet to its destination(s).

The performance simulation results of Section 3.4 have shown that this arrangement offers high and robust delay–throughput performance under a wide range of unicast traffic patterns (see also [Minkenberg00b]). However, so far multicast service has not been satisfactorily integrated into this system.

For the method and algorithm presented here, it is not a prerequisite that the buffer of the switch element be realized as a shared memory; theoretically it will also work with an output-buffered switch having dedicated output buffers. We assume that the switch element imposes no restrictions on the number of multicast packets or their destination sets.

So far multicast-scheduling algorithms have only been considered with FIFO queuing—all arriving multicast packets are stored in a FIFO input queue, and served according to some algorithm, e.g. any of those referred to in Section 4.2.2. This arrangement is prone to HoL blocking, as is also FIFO queuing for unicast traffic. Therefore, the solution presented here is based on VOQ to (a) eliminate HoL blocking, and (b) integrate unicast and multicast traffic in a single algorithm. The main reason why VOQ has not been considered feasible for multicast is that the write bandwidth on the input buffer increases N-fold, which limits scalability. This issue is addressed in Section 4.6.

4.4 Existing Approaches

This section reviews some existing approaches to provide multicast service in the proposed CIOQ architecture, focusing on the organization of the input queues.

4.4.1 2^N Multicast Queues

One approach is to use a separate queue for each possible multicast destination. However, with N outputs, the amount of VOQs to maintain is 2^N (for 16 outputs, 64 K queues), which is clearly not feasible, let alone how arbitration amongst all these queues should be performed.

4.4.2 Duplication at Input Buffer (RAR)

An alternative solution is to perform replication-at-receiving (RAR), i.e., store a copy of the multicast packet in each of the VOQs it is destined to and transmit each copy separately. This is not desirable because it wastes bandwidth towards the switch as a packet with a fanout of F must be transmitted F times across the input link, and switch resources, as each copy must be stored separately in the switch, and because the multicast capabilities of the switch are not exploited at all.

To illustrate how unconditional RAR leads to poor performance, consider this scenario: only a single input is active and generates packets with a fanout equal to the number of switch outputs N. If we always duplicate at the input buffer, a throughput of only $1/N$ can be achieved, whereas fully using the switch's multicast capabilities allows 100% throughput in this scenario.

Hence, this approach can be expected to perform particularly poorly under low or asymmetric input load conditions. Clearly this solution, although very simple, is not satisfactory.

4.4.3 Single Dedicated Multicast Queue

Input-buffered Switches

Numerous schemes have been proposed to achieve good throughput with the configuration using a dedicated multicast queue as shown in Fig. 4.1 [Hayes89, Hui90, Hayes91, Chen92, Chen94, Schultz94, Haung96, Mehmet96, Prabhakar95, McKeown96b, Prabhakar96, Prabhakar97, Chen00a]). The main argument to not apply the VOQ approach to multicast packets seems to be that the input buffer access bandwidth would be increased N-fold, because a single multicast packet may have to be stored in all N VOQs, thus requiring input buffers that run N times faster than the link speed.

However, the basic architecture of having only one FIFO queue for all multicast traffic is highly susceptible to HoL blocking of multicast packets, just as is the case with a single FIFO queue for unicast traffic [Karol87]. In fact, the problem is exacerbated, because one multicast packet can experience blocking on many outputs.

Pathological Scenario

Fig. 4.5 demonstrates a pathological scenario involving HoL blocking of multicast packets in an input-buffered switch. In (a), two inputs are active with multicast traffic for outputs $\{1, 2\}$ and $\{3, 4\}$. Input 1 carries two active flows, input 2 only one. Here, all flows get their fair share of bandwidth; throughput for the flow to outputs $\{1, 2\}$ is 50%.

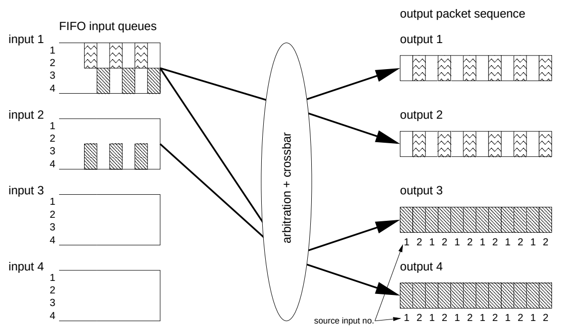
In (b), two more inputs have become active with flows for outputs $\{3, 4\}$ as well, causing these outputs to be overloaded. As the figure demonstrates, HoL blocking now occurs on input 1, dropping the throughput for the flow to outputs $\{1, 2\}$ to 25%, only half of its fair share. Alternatively, input 1 could be served every cycle to give the flow to $\{1, 2\}$ its fair share, but in that case the flow from input 1 to $\{3, 4\}$ gets twice its fair share, while the other inputs get less.

Note that this is essentially the same as HoL blocking for unicast traffic (just replace the destination sets $\{1, 2\} \rightarrow \{1\}$, $\{3, 4\} \rightarrow \{3\}$). However, because of multicast, multiple outputs are affected by HoL blocking on a single input, thus leveraging the impact on output utilization by a factor equal to the fanout F of the impacted flow. Also note that fanout splitting does not help here.

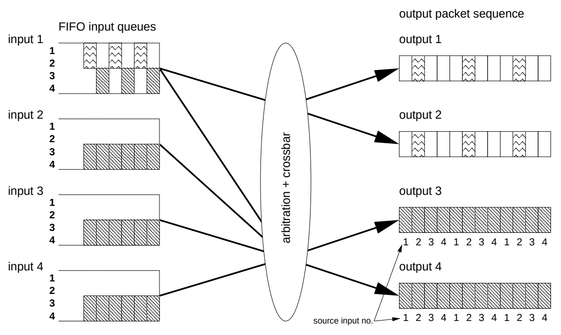
Fanout Splitting

The simplest scheduling approach with a dedicated multicast queue is *one-shot scheduling*, meaning a multicast packet must be transmitted to all its destinations at the same time.

Fanout splitting, on the other hand, means that a multicast packet may be transmitted in more than one part, at different times. It has been recognized that allowing fanout splitting, as opposed to one-shot scheduling, can improve throughput performance notably (see Fig. 2 in [Prabhakar97]), at the cost of slightly more complex implementation. Still, the HoL blocking



(a) Non-congested scenario: Each flow gets its fair share of bandwidth.



(b) Congested scenario: The flow to outputs $\{1, 2\}$ is unfairly impacted by congestion on outputs $\{3, 4\}$.

Figure 4.5: An example of throughput degradation due to multicast HoL blocking.

problem remains. Additionally, the prospect of having to implement entirely separate scheduling mechanisms for unicast and multicast is unattractive.

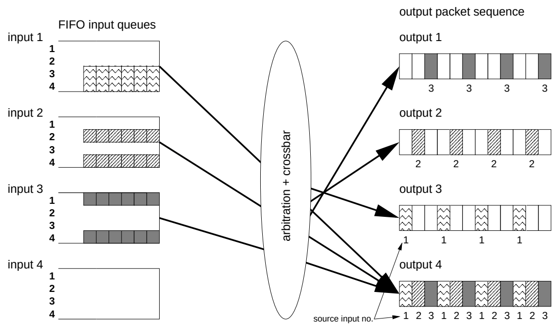
Fig. 4.6 demonstrates how fanout splitting can improve performance. Fig. 4.6a shows three flows with one common destination. Packets from each flow can only be transmitted if all of its destination ports are available. Implementing fanout splitting improves delay characteristics in this case, as is demonstrated by Fig. 4.6b: the non-common destination ports can be served as soon as a new packet appears at the HoL. Throughput is not improved in this case, because of HoL blocking—(part of) a new packet is only allowed to be served once the previous packet has been served completely. Fig. 4.6c demonstrates that organizing the queues in a VOQ manner allows 100% throughput in this scenario, a considerable improvement.

Combined Input/Output-Buffered Switches

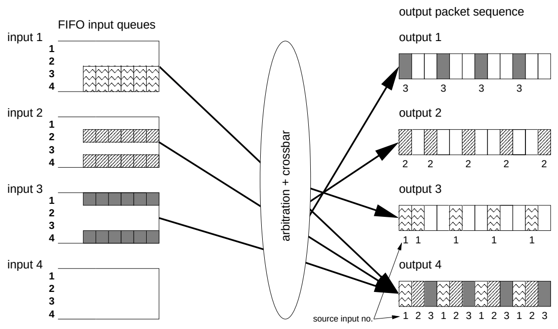
The situation in the proposed combined input/output-buffered switch is slightly different, because no centralized arbiter is required—each input buffer can transmit independently of all others, even for multicast traffic, and the output-buffered switch element resolves contention. Therefore, each input buffer has its own arbiter that decides which VOQ is allowed to transmit.

In the current PRIZMA-E input buffer implementation, all multicast packets are also stored in a dedicated multicast queue, which is served like the regular N VOQs, i.e., it contends with the other N VOQs in the OQF or RR scheme, with the exception that multicast packets *are not subjected to output-grant flow control*. This, however, implies that multicast traffic can easily monopolize output queues, leading to possible performance degradation on other ports. These output-queue grant violations can be allowed because each PRIZMA output queue can hold all shared memory addresses. Obviously, shared memory grant may *not* be violated.

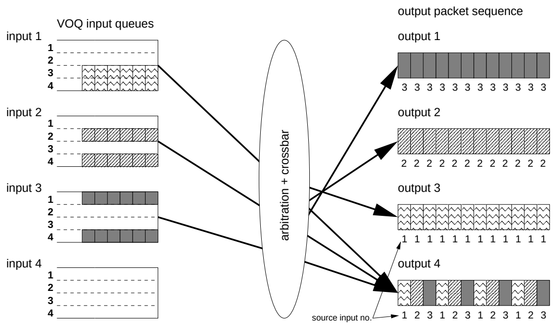
Alternatively, with the dedicated FIFO multicast queue, packets can be subjected to flow control, requiring that grant is present for all destinations, but this leads to serious HoL blocking, causing low throughput for *all* multicast connections. Allowing fanout splitting alleviates HoL blocking, but requires extra complexity.



(a) FIFO queues, no fanout splitting.



(b) FIFO queues, with fanout splitting.



(c) VOQs, with fanout splitting.

Figure 4.6: Multicast scheduling with fanout splitting.

Clearly, none of the solutions presented above are entirely satisfactory. The next section presents a new way to integrate multicast traffic into the VOQ mechanism, without requiring extra VOQs, while still complying with output-queue-grant flow control. Additionally, scheduler complexity remains $O(N)$.

4.5 Method and Algorithm

First we establish the following definitions: N denotes the number of input and output ports of the switch. For a particular input, the length of the VOQ associated with output j ($\mathrm{VOQ}(j)$) equals $L(j), 1 \leq j \leq N$. $\mathrm{QF}(j), 1 \leq j \leq N$, is the current output-queue-full status. $\mathrm{HOL}(j)$ represents the packet at the head of $\mathrm{VOQ}(j)$. $D(p)$ is the destination set of packet p, and the fanout $F(p) = ||D(p)||$. $\mathrm{Occ}(p)$ is the occupancy count of packet p, i.e. the number of copies of this packet still to be transmitted (see Section 4.6).

4.5.1 Design Principles

The design of our new multicast scheduling algorithm is based on the following principles, derived from previous approaches (Section 4.2):

- VOQ: As VOQ has been shown to be highly beneficial for unicast traffic, we seek to apply this concept also to multicast traffic, in order to overcome the performance limitation imposed by the FIFO queuing of existing approaches.

- RAS: packets should be duplicated as much as possible at the egress, i.e., in the switch element, to obtain better resource efficiency.

- Fanout splitting: There are two sides to fanout splitting: Needless fanout splitting causes higher delays, and should therefore be avoided, but prevents shared buffer monopolization (output-queue lock-out) by splitting up multicast packets into congested vs. non-congested destination sets. For the latter reason, which entails significant performance benefits, we want to incorporate this concept.

- Integration and fairness: In general we have noticed an artificial separation between unicast and multicast traffic in existing approaches, which not only leads to extra complexity, but also causes unfairness among unicast and multicast connections competing for the same outputs. By adopting a single, unified queuing *and* scheduling approach for both unicast and multicast traffic, we can remove these disadvantages.

- Complexity: In order to enable VLSI implementation at high line rates, algorithm complexity should be low, preferably linear in the number of switch ports.

4.5.2 Multicast Split/Merge (MSM) Algorithm

The system operates as follows: Upon reception of a packet, it is duplicated to each of its destination VOQs. This constitutes the *splitting* part of the algorithm. On the transmission side, each input-queue arbiter applies the following algorithm to select (a) which of the HoL packets to transmit and (b) the set of destinations to transmit the packet to.

The first part of the algorithm is identical to the selection algorithm proposed in Section 3.2.3 for unicast packets (the order in which the outputs are visited (the `for` loop over j) is determined by the selection policy). The second, novel part of the algorithm adds multicast support by performing a *multicast transmission merge* iteration. This technique stems from the observation that when a certain packet has been selected for transmission to a certain destination by the unicast selection algorithm, it can simultaneously be transmitted to all its destinations for which an output queue grant has been received. We call this algorithm *Multicast Split/Merge* (MSM) because of the way multicast packets are first split into their individual output packets at VOQ ingress, and then, if applicable, merged again at the VOQ egress. The proposed algorithm is shown in pseudo-code in Fig. 4.7. Its complexity is $O(N)$.

The MSM algorithm exhibits the following fairness properties:

- No VOQ can be starved because of the RR nature of the algorithm.

- Flows competing for the same output, regardless whether unicast or multicast, will receive a fair share of bandwidth because they share the same VOQ, unlike in the case of a dedicated multicast queue.

- Fairness among inputs is guaranteed by the switch element, by allowing either all or none of the inputs to send, depending on the shared-memory and output-queue state.

```
$\mathrm{voq} = -1$ /* initialization */
**for all** $j$ **begin** /* perform the unicast selection */
  **if** $\mathrm{QF}(j) == \mathrm{false}$ **begin** /* dest. OQ not full */
    **if** $L(j) > 0$ **begin** /* VOQ not empty */
      $\mathrm{voq} = j$ /* unicast selection done */
      **goto** done /* go check multicast */
    **end**
  **end**
**end**

done:
**if** $\mathrm{voq} \neq -1$ **begin** /* perform the multicast merge */
  $\mathrm{dst\_set} = \emptyset$ /* set of destinations, initially empty */
  **for all** $j$ **begin**
    **if** $\mathrm{QF}(j) == \mathrm{false}$ **and** $\mathrm{HOL}(j) == \mathrm{HOL}(\mathrm{voq})$
    **begin**
      $\mathrm{dst\_set} = \mathrm{dst\_set} \cup j$
      remove $\mathrm{HOL}(j)$ from $\mathrm{VOQ}(j)$
    **end**
  **end**
  get a copy of the packet at address $\mathrm{HOL}(\mathrm{voq})$
  $\mathrm{Occ}(\mathrm{HOL}(\mathrm{voq})) = \mathrm{Occ}(\mathrm{HOL}(\mathrm{voq})) - ||\mathrm{dst\_set}||$
  transmit packet to the destination set $\mathrm{dst\_set}$
**end**
```

Figure 4.7: The MSM transmission selection algorithm, which is executed by each input queue arbiter individually.

4.6 Implementation

Here we focus on the implementation of the input queue and scheduling algorithm. An implementation of the switch element can be found in [Colmant98c]. Fig. 4.8 shows a possible implementation of a single input queue. It consists of a shared memory bank of M packets, N logical queues (VOQs) of shared-memory addresses, an address manager, and an arbiter that implements the packet-selection algorithm described in Section 4.5. The shared memory organization with a split control/data path is very similar to the one that may be applied in the switch element [Engbersen92, Denzel95], with one significant difference: the aggregate read/write bandwidth through the shared memory equals only twice the port speed, instead of $2N$ times as for the switch memory. By duplicating only the addresses, the bandwidth on the shared memory to support multicast does not have to increase at all compared to unicast. The price we pay for supporting multicast, besides the extra logic required for MSM, is that in the worst case up to N addresses must be written to the VOQs, compared to just one for unicast.

With each shared-memory location x, an occupation counter $\mathrm{Occ}(x)$ is kept. These counters are initialized to zero. When a packet arrives, the address manager will provide a free address

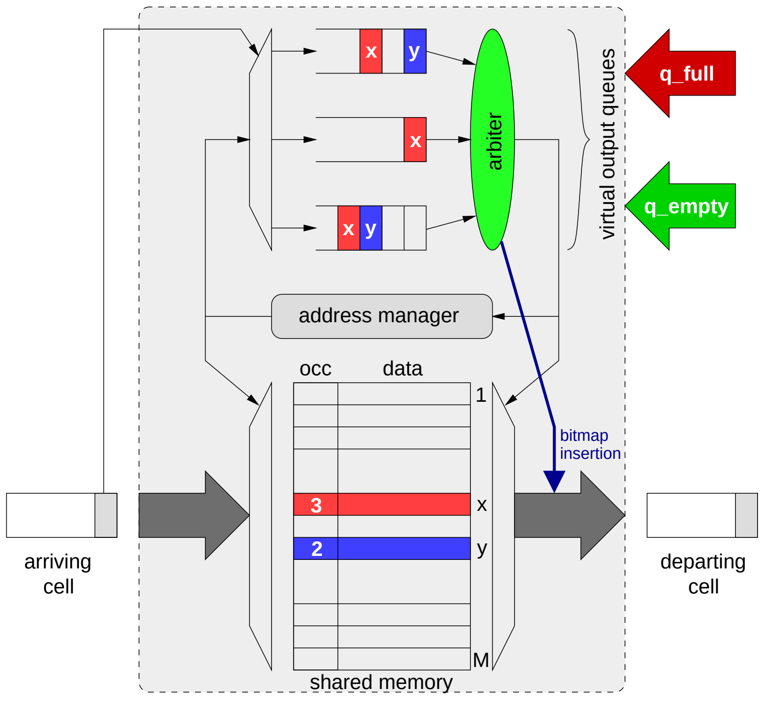


Figure 4.8: Input-queue implementation. The shared memory consists of M packet storage locations, each with an associated occupancy counter. The VOQs contain addresses that point to the shared memory. The arbiter implements the afore-mentioned selection algorithm.

y if one is available, and the packet will be stored at location y; otherwise, the packet will be dropped in its entirety. The corresponding occupation counter $\mathrm{Occ}(y)$ is set to the number of destinations requested by the packet header. The address y is appended to each VOQ to which the packet is destined.

When an address is selected by the arbiter, the corresponding packet is transmitted from the shared memory. The address itself is returned to the address manager, along with a count determined by the arbiter indicating the number of destinations the packet is being transmitted to in the current cycle. The address manager decreases the occupancy counter by this number. If the counter reaches zero, indicating that the packet has been delivered to all its destinations, the address is returned to the free pool.

Before the packet leaves the input queue, a new destination bitmap is inserted according to outcome of the arbiter's selection algorithm.

The incoming "q_full" and "q_empty" arrows indicate the queue-full and queue-empty status information from the switch element, which are used by the arbiter in its decision-making process.

Note that the second step in the selection algorithm (the multicast transmission merge) can actually be performed in parallel on all VOQs by having N address comparators, one at each VOQ, comparing the address at the head of the VOQ with the one selected. The outcome of the N comparators ANDed with the output-queue grant vector represents the destination vector to which the current packet will be transmitted.

Regarding the VOQ selection algorithm, the following complication due to multicast must be taken into account: because of multicast arrivals and departures, up to N VOQs can change from empty to full and back again in a single packet cycle. In case an LRU or OQF selection policy is to be implemented, this implies that up to N insertions and N deletions from the ordered VOQ list may have to done in one packet cycle; this can clearly become difficult to achieve at high data rates. A simple RR policy is not affected by multicast. However, care must be taken in updating the RR pointer: in order to be fair, the pointer should only be updated after the first selection in the MSM algorithm (initial packet selection).

4.7 Performance

The performance of the proposed system under unicast traffic has been studied extensively in Chapter 3. In this section its behavior under multicast traffic will be evaluated by means of simulation. To this end, we employ three different traffic models:

- Uncorrelated arrivals (Bernoulli),
- correlated, bursty arrivals with geometrically distributed burst sizes. A burst is a sequence of consecutive packets from an input to the same output (Bursty/B, B = average burst size), and
- "IP"-like traffic, with a burst-size distribution based on the Internet backbone traffic measurements of [Thompson97].

For all traffic types, the destination vectors are uniformly distributed over all $2^{16} - 1$ possible destinations, implying that heavy multicast traffic is generated (only 16 out of 64 K are unicast destinations).

Contrary to the simulations in Chapter 3, we use finite-size input buffers here, with a 2048-packet memory per input.

4.7.1 Delay–Throughput Characteristics

Fig. 4.9a compares delay–throughput curves for four different multicast schemes, three based on a CIOQ architecture, one on an input-queued architecture:

Scheme 1. CIOQ system with VOQ input buffer employing multicasting at the input side (replication at receiving—CIOQ/RAR): upon arrival, the multicast packet is duplicated to all its destination VOQs; this is what Hui and Renner termed *unicast service* [Hui90]. Implementing the input buffer as a shared memory with split data/control path architecture avoids having to provide N-fold memory bandwidth. Essentially, the multicasting function is performed at the VOQ ingress.

Scheme 2. CIOQ system with VOQ input buffer with an extra queue dedicated to multicast traffic, without output-queue-grant compliance. When the multicast queue is served, its HoL packet is transmitted if a shared-memory grant is present, regardless of output-queue grants. In this case, the multicasting function is entirely relegated to the switch element (replication at sending—CIOQ/RAS).

Scheme 3. CIOQ system with VOQ input buffer, using the MSM scheduling algorithm (mixed replication at receiving/sending—CIOQ/MSM).

Scheme 4. IQ system with FIFO multicast input buffer, using the Concentrate algorithm as a best-case representative of this class of multicast scheduling algorithms [Prabhakar97] (IQ/Concentrate).

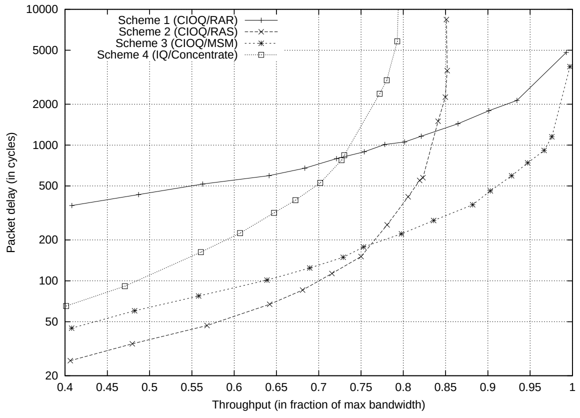
The system is of size 16×16; the shared memory of the switch has 256 packet locations; the output-queue-full thresholds are set to 16, implying complete output buffer partitioning. Note the logarithmic y-axis on the delay–throughput graphs.

At low loads, the CIOQ/RAS scheme performs best, offering the lowest average delay values of all schemes. The reason is that multicast packets will never be held up in the input queues because of output-queue-full conditions. However, this scheme performs worst of all CIOQ schemes at higher loads, having the lowest saturation load. The reason is that shared-memory unfairness (output-queue lock-out) occurs, allowing output queues to monopolize a large portion of the memory and thus blocking other outputs from receiving packets.

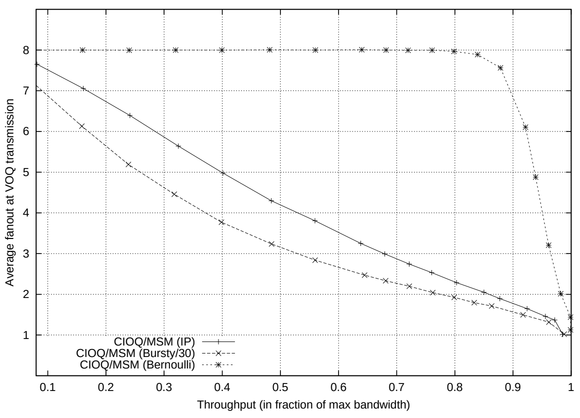
The CIOQ/RAR scheme performs worst at low loads, because each packet has to be transmitted separately for each destination (the switch's multicast capabilities go entirely unused), which naturally leads to much higher delay values. For high loads, on the other hand, it performs reasonably well, as it can support up to 100% load.

The IQ/Concentrate scheme saturates at a lower input load than any of the CIOQ schemes, which can be attributed to HoL blocking.

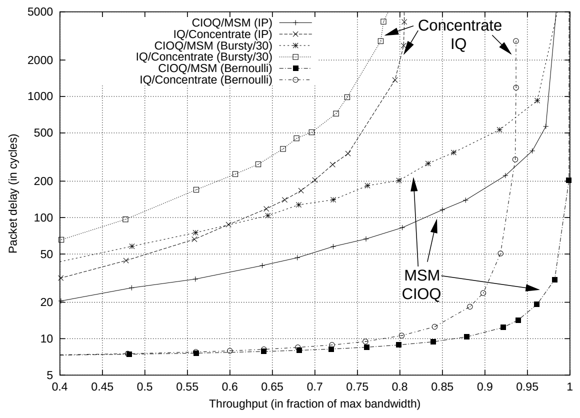
Overall, the MSM scheme proposed here performs best over the entire range—at low input loads the average delay is low, while a maximum throughput of 100% can be sustained. At low loads, MSM behaves like Scheme 2, whereas at high loads it behaves like Scheme 1, thus achieving a good compromise.



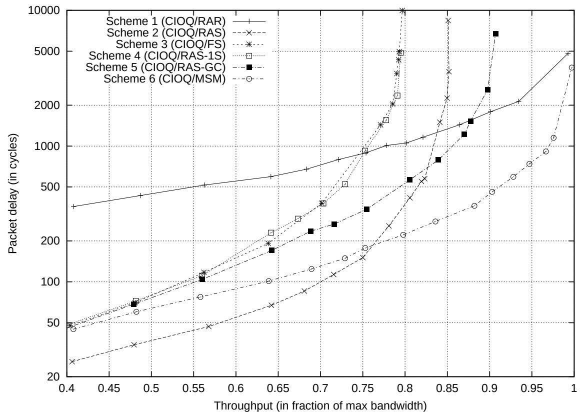
(a) Comparison of delay–throughput curves for four different multicasting schemes under Bursty/30 traffic.



(b) Average fanout F_{TX} at VOQ transmission point as a function of throughput of CIOQ+MSM system under Bernoulli, IP, and Bursty/30 traffic.



(c) Comparison of delay–throughput curves for CIOQ architecture with MSM scheduling algorithm versus IQ architecture with Concentrate scheduling algorithm under Bernoulli, IP, and Bursty/30 traffic.



(d) Comparison of delay–throughput curves for CIOQ architecture with various multicast scheduling algorithms (see Section 4.7.5).

Figure 4.9: Multicast performance simulation results.

4.7.2 Fanout at VOQ Transmission Point

Another interesting metric in evaluating multicast switching is the average fanout at the VOQ transmission point, denoted by F_{TX}. To distinguish, we denote the average fanout of incoming packets by F_{RX}. This metric measures how many output packets, on average, are merged together by a multicast-scheduling algorithm into a single packet transmitted towards the switch.

In general, a scheduling policy that achieves a higher F_{TX} value at a given load will result in smaller delay. Let us assume that at a given low input load an arriving input packet is served completely before the next packet arrives on the same input, which is a reasonable assumption. Thus, applying VOQ does not improve system performance. Under this condition, it takes on average F_{RX}/F_{TX} cycles to completely serve an input packet at the VOQ. Therefore, the average delay D experienced by an input packet is inversely proportional to F_{TX}. Intuitively, this explains why concentrating policies fare better than their distributing counterparts: the former strive towards achieving higher F_{TX} values.

Assuming $K = F_{RX}/F_{TX}$ is an integer, the average delay D under the above conditions equals

$$D = \sum_{k=0}^{K-1} k \frac{F_{TX}}{F_{RX}} = \frac{K-1}{2}. \tag{4.2}$$

If K is not integer, then

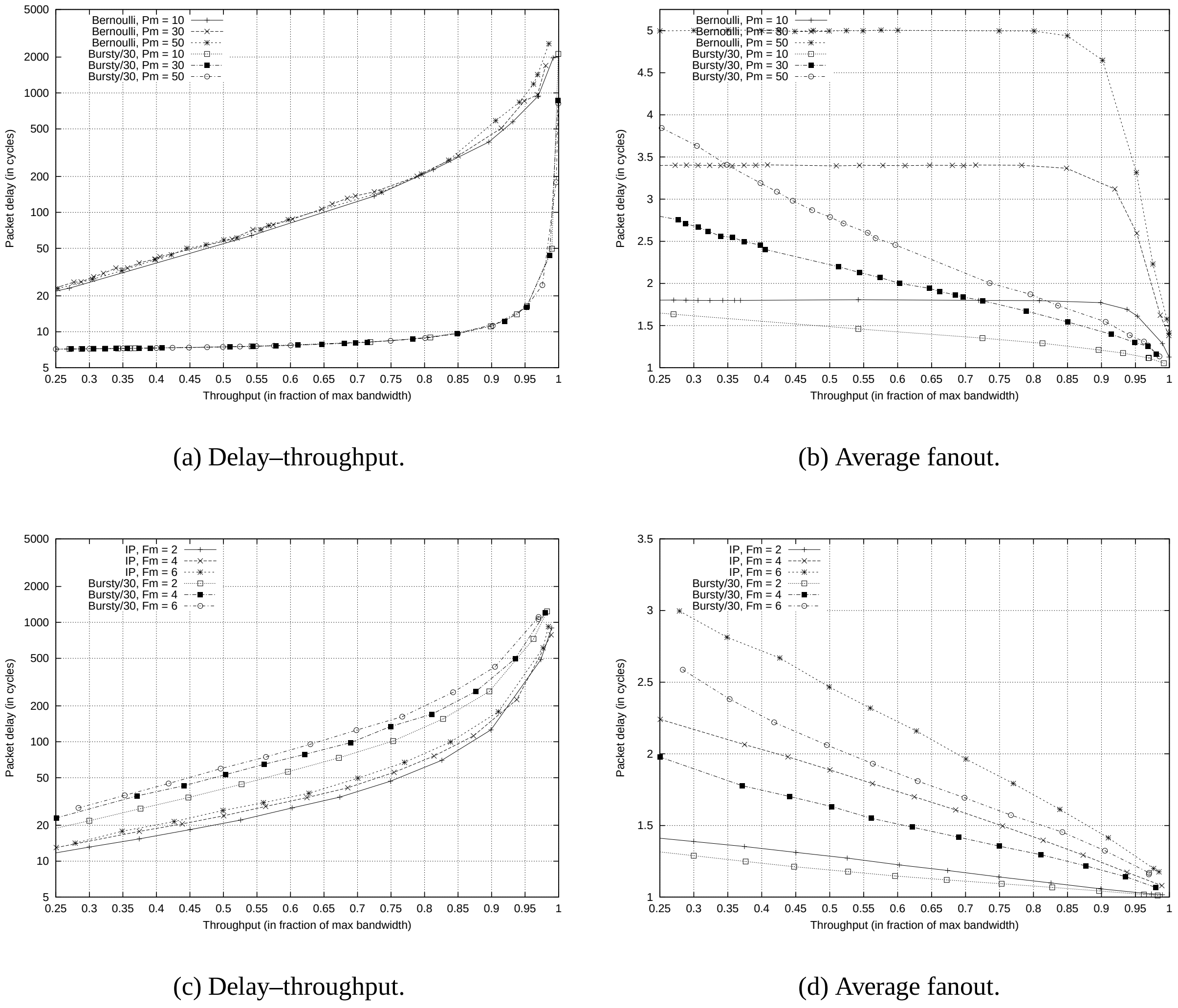
$$\begin{aligned} D &= \sum_{k=0}^{\lfloor K-1 \rfloor} k \frac{F_{TX}}{F_{RX}} + \lfloor K \rfloor \frac{F_{RX} - F_{TX}\lfloor K \rfloor}{F_{RX}} \\ &= \frac{\lfloor K-1 \rfloor}{2} + \lfloor K \rfloor \left(1 - \frac{\lfloor K \rfloor}{K}\right). \end{aligned} \tag{4.3}$$

Fig. 4.9b plots F_{TX} as a function of system throughput for the CIOQ system using MSM with three different traffic types. The scheme using unicast service at the VOQ ingress (Scheme 1) naturally has an average fanout of exactly 1. Scheme 2, on the other hand, which never splits the fanout at the VOQ side (dedicated multicast queue), has an average fanout of 8 under the traffic conditions given. The MSM algorithm is somewhere in between these extremes; at low loads, output-queue-full conditions occur only very rarely, so no fanout splitting is required, thus the average fanout will be close to 8. At high loads, on the other hand, the VOQs will frequently be flow-controlled as a result of output-queue-full conditions, leading to an increasing amount of fanout splitting, and thus a lower average degree of fanout. This behavior is clearly illustrated by Fig. 4.9b, and confirms the performance results of Fig. 4.9a. In essence, the multicast functionality is distributed over the input queues and the switch element, and will automatically adapt to the traffic characteristics to achieve maximum performance.

In Fig. 4.9c, the CIOQ architecture with MSM is compared to an input-queued architecture with Concentrate for three different types of traffic. CIOQ/MSM clearly outperforms IQ/Concentrate in all cases, offering both a lower average delay over the entire range of simulated workloads as well as a higher maximum throughput. The reasons for this are (a) the elimination of HoL blocking at the input queues, and (b) the incorporation of a small amount of output queuing in the switch element.

4.7.3 Mix of unicast and multicast traffic

A traffic mix consisting of a fraction $1 - p_m$ of unicast packets and a fraction p_m of multicast packets is applied to the CIOQ switch using MSM. Each multicast packet has a number of destinations randomly chosen in the range from 2 to $f_{\max}$, so $\overline{f_m} = (f_{\max} + 2)/2$, with $f_{\max}$ being the maximum fanout. For each multicast packet, first the amount of destinations f is drawn uniformly from the range $[2, f_{\max}]$, and then f unique destinations are drawn uniformly from the range $[1, N]$. The results with Bernoulli, IP, and Bursty/30 traffic are shown in Fig. 4.10. System parameters are the same as in the previous section.



(a) Delay–throughput. (b) Average fanout.

(c) Delay–throughput. (d) Average fanout.

Figure 4.10: Performance with a mix of uni- and multicast traffic. The parameters Pm and Fm are the multicast probability p_m and the average fanout $\overline{f_m}$, respectively; N = 16.

Figs. 4.10a and b show results with $\overline{f_m}$ kept constant at 9, whereas the amount of multicast packets p_m is varied from 10% and 30% to 50%. Figs. 4.10c and d on the other hand correspond to simulations where the amount of arriving multicast traffic is fixed at $p_m = 50\%$, while the average fanout $\overline{f_m}$ is varied from 2 and 4 to 6. Note that the effective load equals $((1 - p_m) + p_m\overline{f_m})L$, L being the input load.

Fig. 4.10a shows that performance changes very little as the amount of multicast traffic increases. Fig. 4.10c shows that, although there is no difference in maximum throughput, average delay throughout the load range increases as the average fanout of the generated multicast packets increases. This can be attributed to the concentration of traffic on fewer input ports for a given effective load in the presence of more multicast traffic, thus leading to increased input contention, and therefore larger delays. However, the results clearly indicate that the proposed architecture and algorithm are virtually insensitive to the traffic mix (recall that at a given throughput, the input loads to obtain this effective load are quite different, according to Eq. (4.1)).

The average fanout curves in Fig. 4.10b show that with Bernoulli traffic, regardless of the amount of multicast traffic, the maximum average fanout (1.8, 3.4, and 5 respectively) is achieved up to about 85%. This implies that, up to that load, almost every multicast packet is transmitted at once, i.e., no fanout splitting is necessary. Beyond 85% load, the average fanout drops rapidly, as multicast packets are increasingly split to obtain better throughput. With bursty traffic, average fanout is significantly lower throughout the load range, which is due to a tendency of bursty traffic to quickly clog certain outputs (transient hotspots), leading to grant removal, to which the MSM algorithm will react with increased fanout splitting to avoid blocking.

4.7.4 IQ size distribution

Figs. 4.11c and d show input queue size distributions for five types of traffic at throughput (a) 64% and (b) 72%. This distribution is obtained by sampling the occupancy of all input buffers at every packet cycle, and collecting these samples in a histogram. From this, we can plot the cumulative distributions shown in Figs. 4.11c and d, representing the fraction of time that the average input buffer size exceeds a given value (plotted along the x-axis). Figs. 4.11a and b show the corresponding delay–throughput and fanout curves, respectively.

Note that the curves for Bernoulli traffic fall outside the plot range because the input-queue size is almost always smaller than 16.

4.7.5 Alternative CIOQ Multicast Schemes

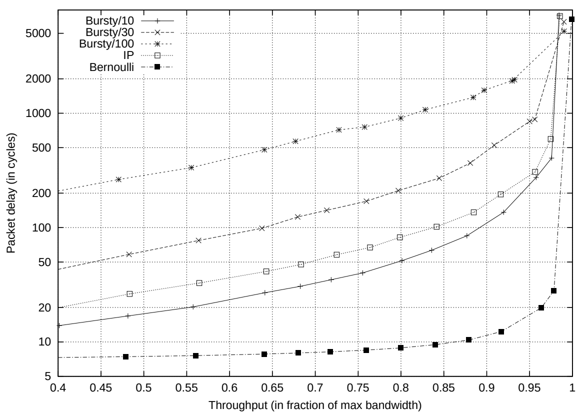
In this section we will study three additional multicast schemes based on the proposed CIOQ architecture. All three use a dedicated multicast FIFO, but employ different strategies to reduce HoL blocking. The performance results of these schemes will again confirm that multicast HoL blocking is significant, and that the MSM algorithm effectively eliminates it.

Fig. 4.9d compares delay–throughput curves for six different multicast schemes, including three alternatives to the schemes studied in the previous section. All are based on the proposed CIOQ architecture:

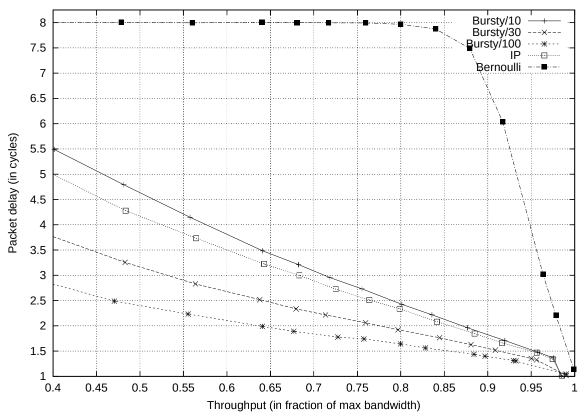
Scheme 1. CIOQ/RAR, as before.

Scheme 2. CIOQ/RAS, as before.

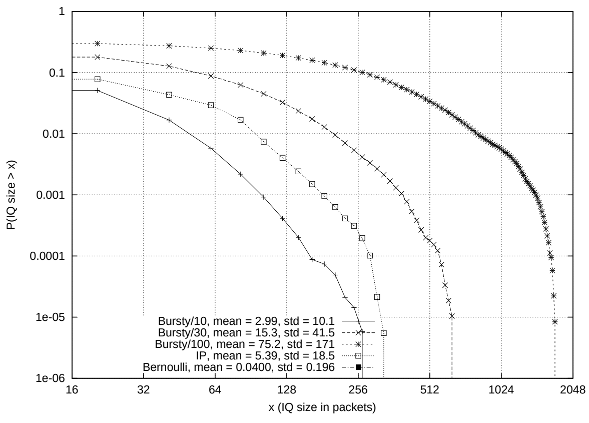
Scheme 3. CIOQ system with VOQ input buffer with a dedicated multicast queue, employing fanout splitting to achieve output-queue-grant compliance. When the multicast queue is



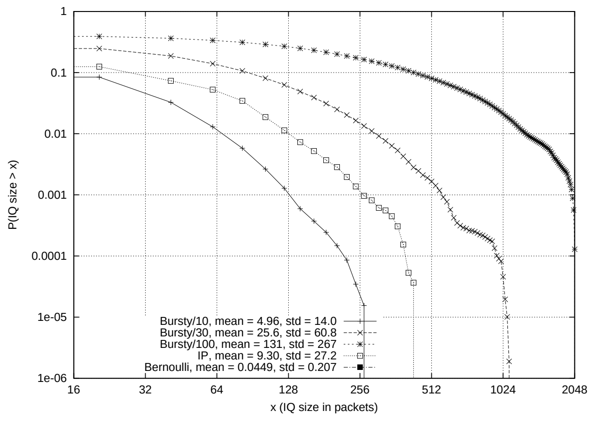
(a) Delay–throughput.



(b) Average fanout.



(c) IQ size distribution, throughput = 64%



(d) IQ size distribution, throughput = 72%

Figure 4.11: Input-queue size distributions.

served, the HoL packet is transmitted only to those outputs for which it has output-queue grant. The residue (that is, the outputs to which the packet still has to be transmitted) is left at the head of the multicast queue (fanout splitting—CIOQ/FS).

Scheme 4. CIOQ system with VOQ input buffer with a dedicated multicast queue, with one-shot scheduling to achieve output-queue-grant compliance. When the multicast queue is served, the HoL packet is transmitted only if it has output-queue grant for all outputs it is destined to, otherwise it is not served and remains at the head of the multicast queue (replication at sending, one-shot—CIOQ/RAS-1S).

Scheme 5. CIOQ system with VOQ input buffer with a dedicated multicast queue, with "output queue grant collection" at the transmission side: when a packet at the head of its VOQ is selected to be transmitted, it is removed from the queue, but *only* transmitted when it is the *last* copy of the packet, otherwise the selection algorithm proceeds. If it is indeed the last copy, it is transmitted to all its destinations at once, regardless of output queue grant. A shared memory implementation with separate data/control path greatly simplifies keeping track of the number of copies of each packet and reduces memory bandwidth requirements (replication at sending with grant collection—CIOQ/RAS-GC).

Scheme 6. CIOQ/MSM, as before.

The simulated system is of size 16×16; the shared memory of the switch has 256 packet locations; the output-queue-full thresholds are set to 16. The traffic type is bursty with geometrically distributed burst sizes with an average of 30 packets per burst (Bursty/30). The destination vectors are uniformly distributed over all $2^{16} - 1$ possible ones. Fig. 4.9d shows delay–throughput graphs for all six schemes.

Note that the schemes that use a dedicated multicast queue and obey flow control (Schemes 3 and 4) perform worst in terms of maximum throughput. Both saturate around 80%, and very little improvement is gained by employing fanout splitting (Scheme 4). The limiting factor here is clearly the HoL blocking due to the FIFO queuing.

Compared to Scheme 2 (RAS), Scheme 5 performs better at higher loads, because the grant collection provides some protection against output-queue lock-out due to shared-memory monopolization, but on the other hand it performs worse at lower loads, because multicast packets may be held up longer at the input until all requested grants are collected.

Overall, MSM performs best over the entire range—at low input loads average delay is low, while a maximum throughput of 100% can be sustained. The results from this section demonstrate that employing a dedicated multicast FIFO performance is significantly worse than the proposed MSM system, even when fanout-splitting or other improvements are applied.

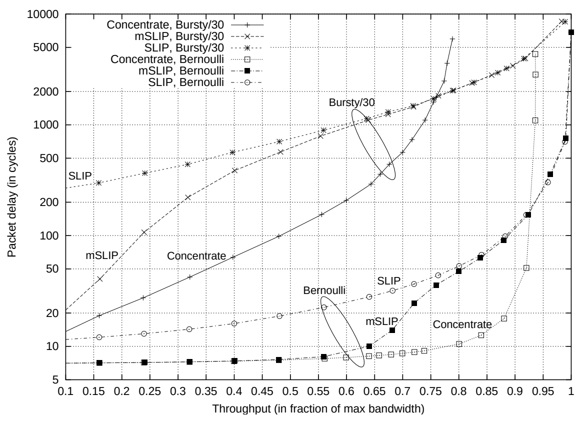
4.8 MSM in Input-Queued Switches

So far, we have only considered applying the MSM algorithm to a CIOQ system. However, the method can also be easily applied to switches that rely on input queuing alone. In particular, the SLIP algorithm [McKeown99a] could take advantage of MSM. By the nature of the SLIP algorithm, an input may receive grants from multiple inputs. In the unicast case, all but one of these are wasted. In the multicast case, however, we can accept multiple grants by applying MSM. In order to do this, a slight modification to SLIP is required:[4] the request, accept, grant, and pointer-update steps are performed as usual, but each input subsequently checks whether any additional grants have been received for output packets that correspond to the one accepted, and will also forward them to their outputs. In this way, integrated unicast and multicast scheduling is achieved in an input-queued architecture, and HoL blocking is eliminated.

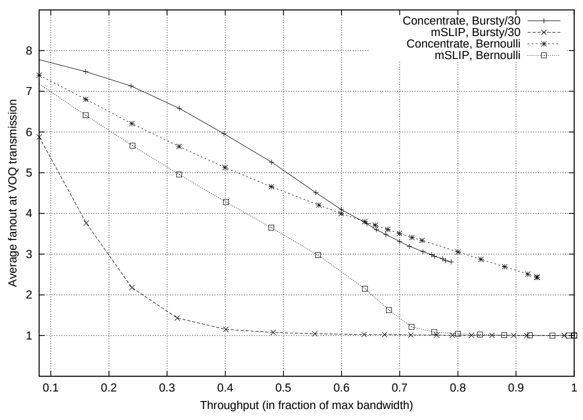
We compare this approach, *mSLIP*, to the Concentrate algorithm and to the regular SLIP using unicast service (RAR) for a 16×16 switch system loaded by Bernoulli and Bursty/30 traffic. Destinations are again uniformly distributed over all possible multicast vectors. Fig. 4.12a shows delay–throughput curves, and Fig. 4.12b the fanout F_{TX} at the VOQ transmission point as a function of throughput.

Owing to HoL blocking, Concentrate saturates at about 80%, while mSLIP can sustain up to 100% throughput. However, at medium loads, Concentrate achieves significantly better delay

[4]In [McKeown99a] a version of SLIP called *ESLIP* is described that also supports multicast, although it uses an additional FIFO queue at each input for multicast traffic.



(a) Delay–throughput characteristics of mSLIP and SLIP vs. Concentrate.



(b) Fanout at VOQ transmission point F_{TX} of mSLIP and Concentrate.

Figure 4.12: Performance comparison of mSLIP (VOQ) vs. Concentrate (FIFO) for a switch size of 16×16, with Bernoulli and Bursty/30 traffic.

figures. To explain this, we have to look at the F_{TX} curves. These clearly show that Concentrate achieves much better values than mSLIP, thus leading to significantly lower delays at low to medium throughput values. These results are not surprising, as SLIP essentially tends to distribute the grants to achieve maximum throughput because it was designed for unicast traffic. In fact, the results show that at medium to high throughput, mSLIP behaves identically to regular SLIP with RAR, while fanout merging is only effective at low loads.

These results indicate that integrating unicast and multicast traffic in a purely input-queued system is not easy; a compromise between distributing for unicast and concentrating for multicast must be found. The core problem is that the central scheduler is faced with the compounded complexity of having to (a) resolve both input and output contention and (b) schedule both unicast and multicast traffic simultaneously, whereas the CIOQ approaches distributes these functions over input and output, leading to a simpler solution.

4.9 Conclusions

A CIOQ packet-switch system has been proposed to integrate the switching of unicast and multicast traffic. The shared-output-buffer switch with a relatively small memory and per-output flow control enables VOQ scheduling algorithms to be distributed across all inputs instead of requiring a centralized scheduler. This applies to unicast as well as multicast scheduling. Additionally, we have shown how applying the concept of fanout splitting and merging allows the integration of both unicast and multicast traffic into a homogeneous, unified queuing and scheduling structure. Just as the buffering and scheduling is distributed over input queues and switch element, also the multicast functionality becomes distributed. The performance advantages compared to the two other CIOQ approaches (RAS and RAR) and the IQ approach (Concentrate) have been established under a range of traffic conditions, and an implementation, amenable to high-speed hardware, has been outlined.

A modification to SLIP has been proposed to achieve integrated unicast and multicast scheduling in an input-queued architecture, but the results indicate that more research into this problem is warranted.

The results from this chapter have partly been published in [Minkenberg00a, Minkenberg00c].

4.10 Discussion

Note that the MSM algorithm is based on the observation that *payload-equal* packets can be combined into a single multicast packet transmitted to the switch. Copies of the same multicast packet happen to be payload equal, but two different packets with exactly the same payload could be combined as well. As a variation, the search depth to find payload-equal packets (or equal pointers, as in the proposed implementation) can be increased from one, as proposed here, to some fixed depth d, which incurs extra implementation complexity but may improve performance. Out-of-order delivery can occur in this case.

Chapter 5

Frame Mode

In this chapter we will often use the following quantified expression notation:

$$(\underline{\mathrm{OP}}\, i : \mathtt{range}(i) : \mathtt{expr}(i))\,,$$

where OP can be any symmetric and transitive binary operator, i is a list of one or more dummy variables, $\mathtt{range}(i)$ is a boolean expression in i, and $\mathtt{expr}(i)$ can be any expression, usually also a function of i. This notation is shorthand for

$$\mathtt{expr}(i_0) \text{ op } \mathtt{expr}(i_1) \text{ op } \cdots \text{ op } \mathtt{expr}(i_{n-1}) \text{ op } \mathtt{expr}(i_n),$$

where i_0 through i_n are all the values of i that satisfy the $\mathtt{range}(i)$ condition. See [Cohen90, Chapter 3] for more details.

5.1 Introduction

Many data transmission protocols and technologies, such as for instance TCP/IP and Ethernet, use variable-length packets as their transmission units, whereas high-speed cell switches typically only support short, fixed-size units. It would be desirable to support long variable-length data units in a fixed-size cell switch by offering non-interleaving switching and transmission of such data units, i.e., to support long packets by offering non-interleaving switching using a fixed-size packet granularity. This would remove the need for cell reassembly at the receive side, considerably reduce header overhead, and lead to better average delay characteristics. Additionally, many such networks employ multicast, i.e. the duplication of one incoming data unit to multiple outputs, and QoS provisioning, i.e. service differentiation among classes of traffic, as important functions. Consequently, multicast and QoS support must today be offered by any switching fabric. A common implementation of high-speed cell switches adopts an output-queued, shared-memory approach. Integrating the functions of multicast, QoS support, and variable-length data units in such an architecture entails a number of possible deadlock scenarios, which must be addressed in order to prevent switch operation from being stalled indefinitely

Here, we aim to tackle three problems:

- offering deadlock-free support for variable-length packet mode (*frame mode*),

- extending this frame mode with multicast support in a deadlock-free fashion, and
- enhancing this frame mode with support for multiple traffic priorities, also in a deadlock-free fashion.

We present solutions to all three problems, as well as suggestions towards a practical hardware implementation of these solutions.

We demonstrate how, at a modest cost in terms of additional hardware, the concepts of both multicast and frame mode can be unified in an elegant manner in a single-chip switch fabric, based on an output-queued, shared-memory architecture.

5.2 System Description

Fig. 5.1 displays the system level architecture. We assume a system dimension of $N \times N$, although the approach is not limited to systems with equal numbers of inputs and outputs. At the input side, long variable-length data units arrive and are segmented into fixed-length cells by the segmentation units depicted in Fig. 5.1. In general, an arriving data unit may not equal an integral number of the fixed-size packets, in which case a certain inefficiency occurs, which may be close to 50% in case the arriving units are just one byte larger than the fixed-size packets supported by the switch. The long-packet support described here does not aim specifically to reduce this problem, because it does not operate at such a fine granularity.

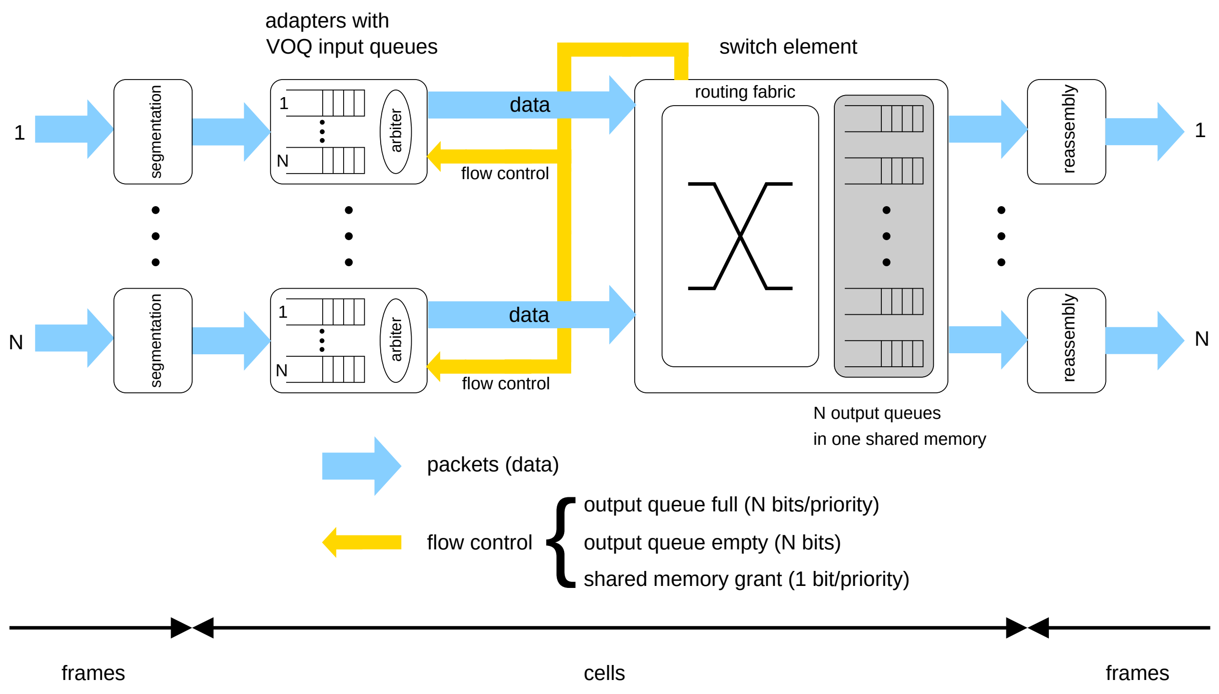


Figure 5.1: System architecture.

The segmented cells are forwarded to the input queues, where they are stored according to their destination (VOQ). The input-queue arbiter makes a decision on which start-of-frame cell to forward, in case multiple are present, taking into account output-queue grant and memory-grant information. Strict FIFO order of cells destined to the same output and of the same priority must be maintained to guarantee frame integrity. Although not mandatory for correct system operation, it is recommended that frames be transmitted from the input queue without interleaving cells of frames to other outputs, that is, once a start-of-frame cell has been selected, the input queue forwards the corresponding continuation and end-of-frame cells in subsequent time slots, again always taking grant information into account. Other start-of-frame cells are only considered for transmission once the current frame has been completed, i.e., just after an end-of-frame cell has been transmitted. This allows only SoF cells to carry a header; CoF and EoF cells carry only a continuation/end-of-frame flag and their priority. As the switch knows that all subsequent cells from the same input and priority belong to the same frame, the header can be omitted from all but the SoF cell. Naturally, this leads to a considerable reduction in header overhead that is proportional to the average frame size.

Cells are forwarded to the switch, which routes them to their proper destination(s). The cells emerging at the egress side of the switch are then reassembled into frames by the reassembly units. If frame mode is enforced, this block will have almost no functionality (except perhaps stripping off the switch's internal cell headers). The internal organization of the switch is such that the control section only handles memory addresses, while the data is passed through the shared memory, as described in [Denzel95].

The shared memory configuration is known to be well suited for implementation of multicast transmission—the data is stored only once, whereas the corresponding address is duplicated to all destination queues. A counter associated with the memory address, initially set to the number of copies to be transmitted, is decremented for every copy that leaves the switch, and the address is only released when the counter reaches zero, i.e. the last copy has left the switch.

The switch employs a flow-control mechanism consisting of two signals:

- a *shared-memory grant* $\mathbf{G}_{i,SM}$, based on memory occupancy O_{SM}. When the memory fill exceeds a programmable threshold T_{SM}, the shared-memory grant is removed. The input queues are no longer allowed to send more cells as soon as they detect this condition. Note that, for the time being, the shared-memory grant is not actually a function of the input i:
$$\mathbf{G}_{i,SM} := O_{SM} < T_{SM}, \tag{5.1}$$
where $0 \leq i < N$.

- an *output-queue grant* (OQ grant) $\mathbf{G}_i(j)$, based on output-queue occupancy $O_Q(j)$. This is a vector of size N, of which bit j corresponds to the status of output queue j. When the output-queue fill exceeds a programmable threshold $T_Q(j)$, the corresponding output-queue grant is removed by resetting the corresponding bit in the grant vector. When the input queues detect this condition, they will not send any more cells destined to this output.
$$\mathbf{G}_i(j) := O_Q(j) < T_Q(j), \tag{5.2}$$
where $0 \leq i, j < N$.

Additionally, the output queues are assumed to be of such size that each queue can fit all memory addresses. Cells are always accepted as long as there are shared-memory addresses available, regardless of output-queue status. The unavailability of a shared-memory address is flagged as a fatal error condition, as this would imply the loss of a cell.

5.3 Frame Mode

A *frame* is defined to be a variable-length data unit, equivalent in size to a multiple of the fixed cell size of the switch. In order to transmit a frame, it is segmented into smaller, fixed-length cells, which are subsequently routed through the switch. The first cell in a frame is termed the *start-of-frame* cell, any subsequent cells before the last cell *continuation* cells, and the last cell *end-of-frame* cell. If no additional precautions are taken, cells from different frames (arriving from different inputs) may be interleaved on one output, so that frame reassembly is required. Therefore, it is desirable to transmit frames in a non-interleaved fashion. This mode of operation is henceforth referred to as *frame mode*.

Unfortunately, introducing the frame mode leads to a number of possible deadlock conditions, namely

- output-queue deadlock and shared-memory deadlock due to unicast traffic,
- shared-memory deadlock due to multicast traffic, and
- shared-memory deadlock due to multiple traffic priorities.

In general, these deadlocks occur because an output is waiting for a continuation cell of the frame it is currently transmitting, but this cell cannot enter the switch because of an output-queue-full or a shared-memory-full condition.

5.4 Unicast Deadlock Prevention

First, we will explain how to prevent deadlock conditions from occuring when only unicast traffic (traffic with only a single destination) is considered.

The output-queue-full deadlock condition occurs when the output-queue threshold is crossed while there are no more continuation cells of the current frame in the queue. As the queue is marked full, these will also not be allowed to enter the switch, and because the queue cannot transmit any cells, the occupancy cannot drop below the threshold again, so the queue is deadlocked.

The shared-memory deadlock is similar. When the shared-memory threshold is crossed (the memory is "full") all inputs are blocked from sending any more cells, possibly preventing the required continuation cells from entering the switch, in turn preventing the switch from freeing up the memory addresses needed in the first place.

Fig. 5.2 illustrates the deadlock. Two very long frames 1 and 2 have the same destination, of which frame 1 is being transmitted. Before frame 1 finishes, the output queue (and the shared memory) fills up with cells of frame 2, until the queue-full threshold (or the shared memory threshold) is crossed, preventing further cells from entering the switch.

We have developed the concept of *active inputs* to circumvent both of these deadlock conditions.

Definition 5 (Active Concept) *An input i is defined as being* active with respect to output j *when on output j a frame that* arrived on input i *is being transmitted. The active states are*

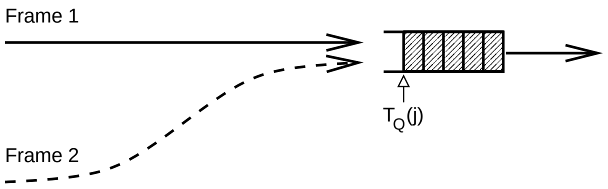


Figure 5.2: Frame Mode output-queue deadlock.

represented by the boolean vectors $\mathbf{A}_i(\cdot)$. *An input is said to be* active, *represented by* $\mathbf{A}_i^\star$, *when it is active with respect to at least one output:*

$$\mathbf{A}_i^\star := (\exists j : 0 \le j < N : \mathbf{A}_i(j) \equiv \mathtt{true}). \tag{5.3}$$

“Being transmitted” in this sense means that the start-of-frame cell of said frame has been transmitted on the output, whereas the end-of-frame cell has not. A frame being transmitted is also called an *active frame* and the output it is being transmitted on is an *active output*.

Additionally, an *almost-full condition* is defined for the shared memory. A programmable almost-full threshold[1] T_{AF} is compared to the current memory occupancy O_{SM}. When the occupancy is above this threshold, the shared memory is said to be *almost full*. This threshold is programmed to some value *below* the actual memory-full threshold.

Now, compared to the conventional flow control of Eqs. (5.1) and (5.2), the outgoing flow control to the adapters is modified as follows for the shared-memory grant

$$\mathbf{G}_{i,SM} := (\neg(O_{SM} \ge T_{AF}) \vee \mathbf{A}_i^\star) \wedge \neg(O_{SM} \ge T_{SM}) \ , \tag{5.4}$$

whereas the output-queue grants are now determined by

$$\mathbf{G}_i(j) := \neg(O_Q(j) \ge T_Q(j)) \vee \mathbf{A}_i(j) \ , \tag{5.5}$$

with $0 \le i < N$ and $0 \le j < N$.

The above equations can be read as follows:

- An input i receives shared-memory grant if and only if ((the shared memory is not almost full) OR (input i is marked active)) AND (the shared memory is not full).
- An input i receives output-queue grant for output j if and only if (output queue j has not exceeded its threshold) OR (input i is marked active for output j).

Note that the switch resorts to a crossbar-like mode of operation when the almost-full threshold is exceeded, because then only one input is allowed to send to an active output, i.e., there is a one-to-one matching between (active) inputs and outputs, just as in a crossbar. Note also that both shared-memory grant as well as output-queue grant are determined on a per-input basis now.

[1]Assuming a round-trip time equivalent to R cell slots between switch and adapter, this threshold should be programmed smaller or equal to the memory-full threshold minus R times the number of ports N: $T_{AF} \le T_{SM} - R * N$.

This scheme solves the output-queue deadlock, because it ensures that the output-queue grant for the input that may still have to send the rest of the frame is positive as long as the frame has not been completed, thus allowing it to bypass the output-queue full condition. Note that *only this particular* input gets a grant for this output! The active status of an input must be marked on a per-output basis because an input can be active for *multiple* outputs simultaneously. If the input queue does not interleave frames, it is guaranteed that only one of these frames is not entirely in the switch yet.

Furthermore, this scheme also solves the shared-memory deadlock. Recall that this deadlock occurs because the inputs that have the required continuation cells cannot send because the switch is full, and the switch cannot free up memory addresses because it needs those continuation cells. What we achieve by the active scheme is that when the switch is almost full (close to the full threshold, but there are still some addresses available), we make sure those addresses go to the inputs that need them.

Another way of looking at it is that, when almost full, we guarantee that at least as many cells exit the switch as enter, thereby ensuring that the memory occupancy does not grow further, and thus preventing the memory from reaching the full threshold. That the previous statement holds true can easily be seen when realizing that there are never more active inputs than there are frames being transmitted (no output can be busy transmitting frames from more than one input), and hence, in each cycle at least as many cells leave the switch on the active outputs as enter on the active inputs.

5.5 Multicast Deadlock Prevention

Now that the deadlock conditions have been solved for the unicast case, another deadlock scenario presents itself when multicast traffic is taken into consideration. The essence of the problem is that cell transmission and address recycling are no longer directly coupled, because a cell being transmitted will only free up a memory address when it is in fact the last copy of that cell. With unicast this always holds true, because there is only one copy to be transmitted, but for multicast cells the active concept is broken.

Consider Fig. 5.3, where two very long frames, one multicast, frame 1, and one unicast, frame 2, go through the switch. Frame 1 is destined for outputs 1 and 2, frame 2 only for output 2. Frame 1 is being transmitted on output 1, frame 2 on output 2. According to the active concept, both inputs 1 and 2 are active, allowing continuation cells from both frames to enter the switch. However, because the cells from frame 1 still need to be transmitted on output 2 as well, *their memory addresses are not freed*. This causes the queue and the memory to fill up with cells for frame 1, until the memory-full threshold is crossed, causing the switch to enter a deadlocked state.

The solution is quite straightforward. Definition 6 introduces the *extended active concept*, which replaces the active concept of Definition 5:

Definition 6 (Extended Active Concept) *An input i is defined as being* active with respect to output j *when on output j a frame that* arrived on input i *is being transmitted* and it is the last copy of the frame. *The active states are represented by the boolean vectors $\mathbf{A}_i(j)$. An input is*

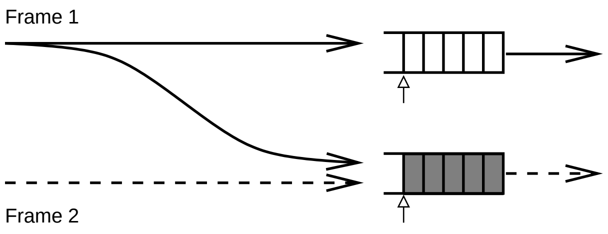


Figure 5.3: Active concept is invalid for multicast.

said to be active*, represented by* $\mathbf{A}_i^\star$*, when it is active with respect to at least one output:*

$$\mathbf{A}_i^\star := (\exists j : 0 \leq j < N : \mathbf{A}_i(j) \equiv \mathrm{true}). \tag{5.3}$$

The rules for determining the shared-memory and output-queue grants remain unchanged. Note the addition of the qualification that an input is only marked active if the *last copy* of the frame is being transmitted. This guarantees that the addresses are indeed being freed as the cells leave the switch, so that cell transmission and address recycling are once again coupled for active inputs. Also note that this extended definition of the active concept still holds for unicast traffic.

5.6 Cyclic Waiting Deadlock

Unfortunately, the shared-memory deadlock is not the only added complication. To maintain good switch throughput also under heavy multicast traffic, we want to be able to transmit multiple multicast frames in parallel on different outputs. As every multicast frame's set of destinations can be any subset of all destinations, this leads to output conflicts. The shared-memory, output-queued architecture of our switch allows us to partially transmit frames, so that not all outputs need to be available before transmission, which leads to higher throughput. However, this ability leads to possible deadlock conditions, as we shall demonstrate below.

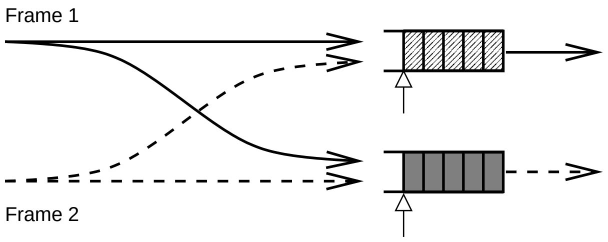


Figure 5.4: Cyclic waiting multicast drame deadlock.

Consider the following case (see Fig. 5.4): Frame 1 from input 1 has destinations 1 and 2. Frame 2 from input 2 also has destinations 1 and 2. Both frames are very long. Frame 1 is active on output 1, frame 2 on output 2. Suppose that these are the only two frames currently in the switch. Because according to Definition 6 neither frame is currently active, the memory will fill up and both frames will be blocked when the almost-full threshold is crossed. At that point, frames 1 and 2 will be mutually waiting for the other's completion, which will never happen, because neither is active, disallowing the required continuation cells from entering the switch. Hence, we have a deadlock that is fundamentally different from the shared memory deadlock—it cannot be prevented by the extended active concept alone.

Note that it is not required that frames have at least two destinations in common for the deadlock to occur. Consider a three-frame scenario as depicted in Fig. 5.5, where frame 1 is destined to outputs 1 and 2, frame 2 goes to outputs 2 and 3, and frame 3 to outputs 1 and 3. Any two frames always have only one destination in common, but when frame 1 is active on output 1, frame 2 on output 2 and frame 3 on output 3, then frame 3 will be waiting for frame 1 to end, frame 1 for frame 2 to end and frame 2 for frame 3 to end, closing the vicious circle.

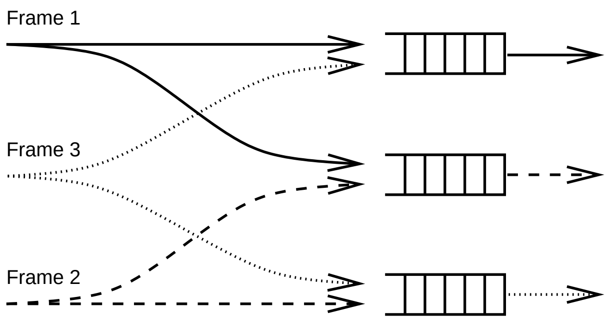


Figure 5.5: Cyclic multicast frame deadlock.

The decision of which frame to send is governed by an arbiter on each output independently (typically using a FIFO discipline). The core of the problem is that these independent decisions may cause cyclic waiting dependencies that cannot be broken once they exist. Therefore, we must prevent these cyclic waiting dependencies from occurring.

Consider the set of frames W that are waiting in the switch. W_j denotes the set of frames that are waiting on output j. The set of destinations of any given frame f is denoted by $D(f)$. We introduce the following definitions:

Definition 7 (Destination Set) *We define the* destination set $D(S)$ *of a set of frames S to be*

$$D(S) = \left(\bigcup f : f \in S : D(f)\right), \tag{5.6}$$

that is, the union of the destination sets of all frames in S.

Definition 8 (Destination Interference Count) *We define the* destination interference count $I(S,d)$ *of a particular destination d with respect to a set of frames S:*

$$I(S,d) = \left(\sum f : f \in S \wedge d \in D(f) : 1\right). \tag{5.7}$$

In words: $I(S,d)$ equals the number of frames in S which are destined for destination d.

Definition 9 (Interference Destination Set) *We define the* interference destination set $I(S)$ *of a set of frames S:*

$$I(S) = \left(\bigcup d : d \in D(S) \wedge I(S,d) > 1 : \{d\}\right), \tag{5.8}$$

i.e., $I(S)$ equals the set of all destinations $d \in D(S)$ for which more than one frame is destined.

Definition 10 (Interfering Frame Set) *A subset S of the waiting frames W, $S \subseteq W$, is called* interfering *if and only if*

$$\|I(S)\| > 0. \tag{5.9}$$

The $\|S\|$ operator is used to denote the cardinality of set S, that is, the number of (distinct) elements in the set.

Definition 11 (Non-interfering Frame Set) *A subset $S \subseteq W$ is called* non-interfering *if and only if*

$$\|I(S)\| = 0. \tag{5.10}$$

Corollary 1 *A set S consisting of zero or one frame(s) is non-interfering.*

Proof: For an empty set S, $I(S) = \emptyset$ holds, as there are no destinations at all. Hence $\|I(S)\| = 0$. For a set S with one frame, $I(S, d) = 1$ for all $d \in D(S)$. Therefore, by Definition 9 $I(S) = \emptyset$, and thus $\|I(S)\| = 0$, completing the proof. □

We now define the notion of *non-critical frames*:

Definition 12 (Non-critical Frames) *Given an interfering subset S of the waiting frames W, a frame f is called* non-critical *with respect to S if the following holds:*

$$\| \left(\bigcup g : g \in (S \setminus f) : D(g) \right) \cap D(f) \| \leq 1. \tag{5.11}$$

In words: the cardinality of the intersection of the destination set of f with the union of the destination sets of all frames but f in S is less than or equal to one. In plain English, any frame f that has at most one destination in common with the total destination set of S is a non-critical frame.

Corollary 2 *All unicast frames are non-critical.*

Proof: This follows directly from the fact that a unicast frame f has only one destination, $\|D(f)\| = 1$, and therefore the intersection of its destination set with any other destination set can never have more than one element. □

Removing all non-critical frames from an interfering set S yields a related set S' that may still contain non-critical frames that have been uncovered by the removal of the other non-critical frames. Recursively repeating the process until no more non-critical frames are found will result in a sequence $S \to S' \to \cdots \to S^\star$, where $S^\star$ is either the empty set or a *critically interfering set*, which will shortly be defined formally. We call $S^\star$ the *reduced set* of S.

Note that for this sequence of sets the following relation holds:

$$\|I(S)\| \geq \|I(S')\| \geq \cdots \geq \|I(S^\star)\|.$$

In other words, the interference destination count of the set sequence is monotonously non-increasing.

Any interfering subset S of the waiting frames W that can be reduced to the empty set by the afore-mentioned reduction procedure is *non-critically interfering*.

Definition 13 (Critically Interfering Frame Set) *An interfering subset $S \subseteq W$ is called* critically interfering *if and only if*

$$\|I(S^\star)\| > 0. \tag{5.12}$$

In words, the cardinality of the interference destination set of the reduced set $S^\star$ of S is larger than zero.

Any interfering set that is not critically interfering is called non-critically interfering, hence

Definition 14 (Non-critically Interfering Frame Set) *An interfering subset S of the waiting frames W, $S \subseteq W$, is called* non-critically interfering *if and only if*

$$\|I(S^\star)\| = 0. \tag{5.13}$$

In summary, any set of frames $S \subset W$ is either

- critically interfering: $\|I(S^\star)\| > 0$, or
- non-critically interfering: $\|I(S^\star)\| = 0$, with two special cases, namely
  - non-interfering: $\|I(S)\| = 0$, and
  - empty: $\|S\| = 0$.

Proposition 1 *Only within critically interfering subsets S of the waiting frames W does the possibility of deadlock through cyclic dependencies exist.*

Lemma 3 *If there is only unicast traffic, no deadlocks through cyclic dependencies can occur.*

Proof: Any subset $S \subseteq W$ is either non-critically interfering or non-interfering. This can be seen by taking any subset S, and then removing all non-critical frames. As there are only unicast frames, all frames can be removed (Corollary 2), rendering S non-critically interfering and thus, by Proposition 1, no deadlocks can occur. $\square$

Definition 15 (Time Ordering) *We assign to each frame f a time stamp $T_j(f) = (t(f), \delta_j(f))$ for each of its destinations $j \in D(f)$. The first element t of the duple (t, δ_j) is the arrival time of the frame at the switch. If multiple frames arrive at a common output queue j at the same time t, they are ordered relatively with respect to each other, with δ_j expressing this relative order. For two frames f and g having a common destination j, the following relation is defined*

$$T_j(f) < T_j(g) \equiv (t(f) < t(g)) \vee (t(f) = t(g) \wedge \delta_j(f) < \delta_j(g)). \tag{5.14}$$

If $T_j(f) < T_j(g)$ holds, we say that f is *earlier* than g. Note that either $T_j(f) < T_j(g)$ or $T_j(f) > T_j(g)$ holds, but never $T_j(f) = T_j(g)$, that is, the time stamps are *unique*.

Corollary 3 *The time order relation is transitive per output j:*

$$T_j(f) < T_j(g) \wedge T_j(g) < T_j(h) \Rightarrow T_j(f) < T_j(h). \tag{5.15}$$

Proof: Follows immediately from Definition 15. □

Definition 16 (Strict Time Ordering) *We impose the following key restriction on the ordering of Definition 15:*

$$(\forall i, j : i, j \in D(f) \cap D(g) \wedge i \neq j : T_i(f) < T_i(g) \Rightarrow T_j(f) < T_j(g)) \,. \tag{5.16}$$

In words, for every pair (i, j) of common destinations of frames f and g, if f is earlier than g on destination i then it must also be earlier on j.

Corollary 4 presents the key aspect of the strict time order relation.

Corollary 4 *The strict time order relation is not only transitive locally per output (see preceding corollary), but also* globally*:*

$$T_i(f) < T_i(g) \wedge T_j(g) < T_j(h) \Rightarrow T_k(f) < T_k(h), \tag{5.17}$$

for any $i \in I(\{f, g\})$, any $j \in I(\{g, h\})$ and any $k \in I(\{f, h\})$.

Proof: Follows immediately from Definition 16. □

Definition 17 (Scheduling Policy) *A scheduling policy P on a set of frames S is a mapping that, at a given instant, decides which frame f out of the set of frames S is the next to become active, $P(S) = f$.*

If, for every subset S of the waiting frames W we can find a scheduling policy P such that no cyclic dependencies occur, we have solved the deadlock problem.

5.7 Deadlock-Free Scheduling Policy

Based on the result of the preceding section, we can now proceed to define a deadlock-free scheduling policy.

First, we create a partition Λ on the set of waiting frames W by means of the following algorithm.

1. Let $\Lambda = \emptyset$.
2. For all frames $f \in W$,
   if $(\forall S : S \in \Lambda : D(S) \cap D(f) = \emptyset)$, **then** create a new set $S' = \{f\}$, and let $\Lambda := \Lambda \cup S'$,
   else take $S'' \in \Lambda$ for which $D(S'') \cap D(f) \neq \emptyset$ and let $S'' := S'' \cup f$.
3. Note that for all $f \in W$, $S' \in \Lambda$, $f \in S' \Rightarrow (\forall S : S \in \Lambda \wedge S \neq S' : f \notin S)$, which ensures that Λ is in fact a partition on W.

Now we classify all sets $S \in \Lambda$ according to Definitions 11, 13 and 14, and schedule them as described below.

When a frame f has completed transmission on a destination d, then $D(f) := D(f) \setminus d$. When a frame f has completed transmission on all its destinations ($D(f) = \emptyset$), then $W := W \setminus f$. The set of newly arriving frames A is added to W, $W := W \cup A$.

5.7.1 Non-interfering sets

For a non-interfering set, the scheduling policy P is irrelevant, as there are no output conflicts. Any frame in a non-interfering set can be selected for transmission.

5.7.2 Non-critically interfering sets

For a non-critically interfering set S, we define the scheduling policy P to be $P(S) = t^{-1}((\underline{\min}\, f : f \in S : t(f)))$.

5.7.3 Critically interfering sets

For a critically interfering set S, we define the scheduling policy P_j on output j to be

$$P_j(S) = T_j^{-1}(\underline{\min}\, f : f \in S : T_j(f)), \tag{5.18}$$

where $T_j^{-1}(\cdot)$ is the inverse function of $T_j(\cdot)$,

$$\begin{aligned} T_j(T_j^{-1}((t, \delta_j))) &= (t, \delta_j), \\ T_j^{-1}(T_j(f)) &= f. \end{aligned}$$

This inverse function is uniquely defined because the time stamps are unique.

Policy 5.18 reads as follows: “on output j the frame $f \in S$ with the smallest time stamp is elected to be transmitted.”

The key point is that Definition 16, the strict time ordering, in combination with scheduling policy 5.18 guarantees that if for any frames f and g of any critically interfering subset $S \in \Lambda$ and if frame f is scheduled to become active before frame g on output i, then frame f will also become active before frame g on output j for all common destinations i and j.

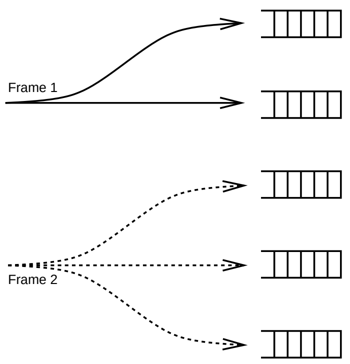
The strict time ordering relationship is satisfied if and only if (a) frames waiting in the same output queue are served in FIFO order according to their arrival times, and (b) multicast frames that arrive simultaneously at different outputs are stored *in the same relative* order in *all* corresponding output queues.

This guarantees that, on every output, there is always a unique earliest frame that does not have to wait for any other frame on any of its destinations. Therefore, cyclic waiting cannot occur and hence the deadlock condition is solved.

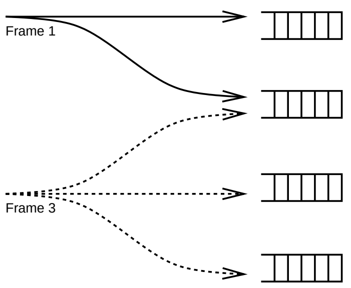
5.7.4 Examples

This section presents a few examples. Fig. 5.6a shows a set S of two non-interfering frames ($\|I(S)\| = 0$), whereas Fig. 5.6b shows a set of two non-critically interfering frames (both frames are non-critical, hence S can be reduced to the empty set, so $\|I(S^\star)\| = 0$).

Fig. 5.7 shows a critically interfering set S. We show the reduction of S to S' (Fig. 5.7b) by removing non-critical frame 4, and subsequently the reduction of S' to $S^\star$ by removing non-critical frame 1. $S^\star$ no longer contains non-critical frames and is obviously non-empty, therefore it is critically interfering.



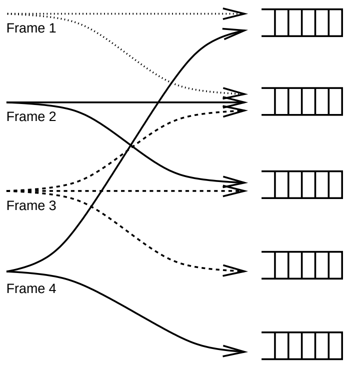
(a) Non-interfering



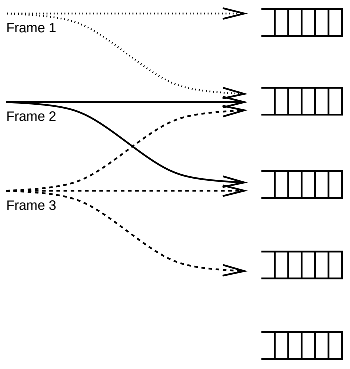
(b) Non-critically interfering

Figure 5.6: Non-interfering and non-critically interfering frame sets

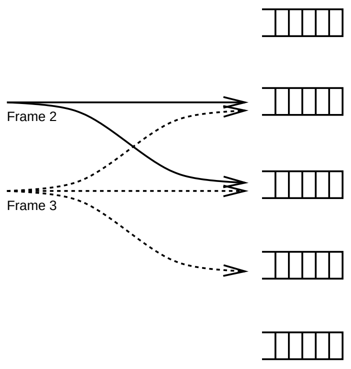
Note how frame 1, while being critical with respect to S, becomes non-critical with respect to S'.



(a) Initial set S



(b) First reduction to S'



(c) Final reduction to $S^\star$

Figure 5.7: A critically interfering frameset S and its reduction to $S^\star$

5.8 Frame Mode and Priorities

Adding QoS support by means of priorities (preemptive, weighted, or otherwise) introduces another dimension to the deadlock problem. We assume P to be the number of traffic priorities supported by the switch. Every frame has an individual priority p, $0 \leq p < P$. The ordering relations imposed on frames only apply to frames of the same priority. Frames of different priorities are allowed to be served in any order, decided by some given transmission arbitration scheme. One possibility is to always give precedence to frames of higher priority, so-called strict, or preemptive priorities. Cells of frames of different priorities may be interleaved at both inputs and outputs.

Complications arise because frames from different priorities are allowed to be interleaved on an output. Therefore, it can happen that an output is flagged active on more than one input, but for different priorities. If no additional measures are taken, it can happen that, while in almost full state, cells from multiple frames are allowed to enter the output queue, whereas only one can exit. Thus, the active concept is broken.

Adding priorities implies that the input status flags have to kept on a per-priority basis, as it can happen that multiple frames are active simultaneously on the same output (one per priority), so the input status arrays from Definition 6 are now indexed per priority as well: $\mathbf{A}_{p,i}(\cdot)$. Similarly, the shared-memory grant and output-queue grant information is also provided on a per priority basis.

Switch operation in almost full mode can be restored by only giving grant to the priority for which a cell has left the switch in the current cycle. This is done by keeping track, for every output, from which input and priority a cell was last transmitted. The scheduling of cells from different priorities is governed by an independent entity at each output.

Definition 18 (Output Transmit Status) *The output transmit status $\mathbf{B}_{p,j}(i)$ is defined to be true when a cell of priority p from input i has been transmitted on output j in the current cycle or no cell has been transmitted at all, and false otherwise.*

Definition 19 extends the extended active concept from Definition 6 to incorporate priorities.

Definition 19 (Transmit Active Concept) *An input i is defined as being* active with respect to priority p and output j *when on output j a frame of priority p that* arrived on input i *is being transmitted* and it is the last copy of the frame. *The active states are represented by the boolean vectors $\mathbf{A}_{p,i}(j)$. We maintain the notion of "active input", indicated by $\mathbf{A}^{\star}_{p,i}$:*

$$\mathbf{A}^{\star}_{p,i} := (\exists j : 0 \leq j < N : \mathbf{A}_{p,i}(j) \equiv \mathtt{true})\,. \tag{5.19}$$

Additionally, an input is said to be transmit-active *for priority p, represented by $\mathbf{A}^{\bullet}_{p,i}$, when it is active with respect to at least one output that is marked active for this priority and input according to Definition 18:*

$$\mathbf{A}^{\bullet}_{p,i} := (\exists j : 0 \leq j < N : \mathbf{A}_{p,i}(j) \equiv \mathtt{true} \wedge \mathbf{B}_{p,j}(i) \equiv \mathtt{true})\,. \tag{5.20}$$

The shared-memory grant is newly defined by Eq. (5.21).

$$\mathbf{G}_{p,i,SM} := (\neg(O_{SM} \geq T_{AF}) \vee \mathbf{A}^{\bullet}_{p,i}) \wedge \neg(O_{SM} \geq T_{SM}), \tag{5.21}$$

with $0 \leq p < P$, $0 \leq i < N$ and $0 \leq j < N$.

Finally, we add a shared-memory threshold per priority, thus arriving at Eq. (5.22):

$$\begin{aligned}\mathbf{G}_{p,i,SM} \;:=\; & (((\neg(O_{p,SM} \geq T_{p,SM}) \vee \mathbf{A}^{\star}_{p,i}) \wedge \neg(O_{SM} \geq T_{AF})) \vee ((O_{SM} \geq T_{AF}) \wedge \mathbf{A}^{\bullet}_{p,i})) \\ & \wedge\neg(O_{SM} \geq T_{SM}),\end{aligned} \tag{5.22}$$

where $O_{p,SM}$ and $T_{p,SM}$ represent occupancy counters and thresholds per priority, respectively.

$$\mathbf{G}_{p,i}(j) := \neg(O_{p,Q}(j) \geq T_{p,Q}(j)) \vee \mathbf{A}_{p,i}(j), \tag{5.23}$$

Eq. (5.23) assumes programmable thresholds per priority as well as queue occupancy counters per priority, although this is not of importance for the functioning of deadlock prevention.

Eqs. (5.22) and (5.23) nicely demonstrate the four different deadlock-prevention mechanisms we have introduced:

- The active flag $\mathbf{A}_{p,i}(j)$ in Eq. (5.23) prevents output-queue deadlock due to both unicast and multicast.
- The term $\mathbf{A}^{\star}_{p,i}$ in Eq. (5.22) prevents shared-memory deadlock on the per-priority thresholds.
- The same term also prevents shared-memory multicast deadlock by means of the extended active concept (only active when last copy is being processed).
- The term $\mathbf{A}^{\bullet}_{p,i}$, also in Eq. (5.22), prevents shared-memory deadlock when the shared memory is almost full. It incorporates both shared-memory multicast and shared-memory priority deadlock prevention.

5.9 Practical Implementation

This section describes some practical considerations with regard to implementation.

5.9.1 Shared Memory

Each shared-memory address consists of storage for one cell, plus an associated multicast counter. Each input controller has an address available in which to store the next arriving cell. When a cell arrives, several things happen:

- the cell data is stored at the available memory address,
- the address is forwarded to each destination output queue, as indicated by the destination bitmap in the cell header, and
- the number of destinations is counted and stored in the counter associated with the memory address.

5.9.2 Frame Mode

The frame mode will be a configurable mode. When enabled, each output keeps track of which frame it is currently processing and will not process any other frames until the current one has been completed.

Each cell carries a flag indicating whether it is a start-of-frame (SoF), continuation (CoF), or end-of-frame (EoF) cell. When a SoF cell of priority p, received on input i, is transmitted on output j **and** it is the last copy of the cell (the associated counter equals one), the output is marked active, and the corresponding bit j is set in the active register $\mathbf{A}_{p,i}$. When an EoF cell, received on input i, is transmitted on output j the corresponding bit j in register $\mathbf{A}_{p,i}$ is cleared, if set. When any cell of priority p, received on input i, is transmitted on output j, the corresponding bit is set in the output transmit status register $\mathbf{B}_{p,j}$.

These registers, $\mathbf{A}_{p,i}$ and $\mathbf{B}_{p,j}$, $0 \leq i, j < N$, are used to modify the grant signals as described in Eqs. (5.22) and (5.23), for the shared-memory and output-queue grant, respectively.

5.9.3 Output Queues

To enable frame mode, the output queues must maintain order information, first to govern the order in which the stored frames are transmitted—in compliance with the strict time ordering rule (Definition 16) to avoid deadlock conditions, and second to maintain the order of SoF, CoF, and EoF cells for each frame. This can be achieved by means of a two-dimensional linked list as shown in Fig. 5.8. In the first, horizontal dimension, all the SoF cells are linked together. In the second dimension, the CoF and EoF cells of the respective frames are linked. Such a 2D queue structure is required for each priority.

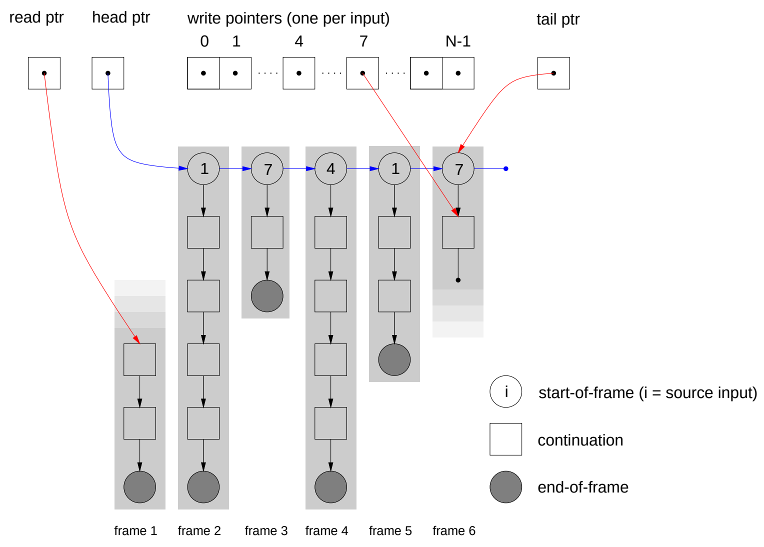


Figure 5.8: Logical output-queue implementation. Frame 1 is currently being transmitted with the read pointer indicating the next cell to be read, whereas frame 6 is being received from input 7, with the corresponding input pointer indicating the last CoF cell of this frame. Note how the SoF cells are linked together in the horizontal direction, with the head and tail pointers indicating the first and last frame in the queue.

Logical operation

Every SoF cell requires two next-cell pointers, one to the SoF cell of the next frame, and one to the first CoF cell of its own frame. CoF cells only require one next-cell pointer, namely to the next continuation cell. EoF cells require no next-cell pointer at all. Note that a cell may be simultaneously a SoF and EoF cell, if the frame consists of only a single cell.

Furthermore, every output queue maintains a number of additional pointers to manage its operation:

- a read pointer, which indicates the next cell to be read from the queue,
- a head pointer, which indicates the first SoF cell in the queue,
- a tail pointer, which points to the last SoF cell in the queue, in order to be able to immediately append newly arriving frames (SoF cells) at the right position, and
- an array of N write pointers, one per input, which point to the last cell of any uncompleted frame from the corresponding input (at any given time, only one frame can be incomplete for any input-output-priority combination). Newly arriving CoF cells are appended at the position indicated by the corresponding write pointer.

All pointers assume the value *nil* if they are currently invalid. In case multiple priorities are to be supported, one set of all of above-mentioned pointers is required for each priority.

We assume there is space for Q cells in every output queue, and the queue slots are numbered 0 through $Q-1$. We assume that a given queue-management scheme provides an empty queue slot q to store an incoming entry. Next, we will describe output queue operation in detail. Initially, the output queue is empty. All pointers are *nil*. When a cell arrives, its memory storage address is passed to the destination output queue, along with the input number the cell arrived on, and the cell type (SoF, CoF, EoF). Depending on the type of the cell being written, the following three cases must be distinguished:

- SoF: If the output queue is empty, the cell is entered in the queue at slot q, and the head, tail, and read pointers are all set to q. If not, the cell is appended at the tail of the queue as indicated by the tail pointer, i.e., the next-SoF pointer of the tail pointer's entry is set to q, and the tail pointer is also set to q. In both cases, the corresponding write pointer is also set to q. This write pointer should be *nil*; if it is not, there is an error condition, because the preceding frame from the same input has not yet been completed. The last CoF (or SoF, if no CoF is present) should then be marked as EoF, to force an end to the incomplete frame. If the head pointer equals *nil*, it is updated to q.
- CoF: the cell is entered in the queue at slot q. If the corresponding write pointer does not equal *nil*, the cell is linked to the preceding cell of its frame by updating the next-CoF pointer of the entry indicated by this write pointer, setting it to q, and the write pointer is also updated to q. If the write pointer equals *nil*, there are two possibilities, depending on the input the cell arrived on. If the input equals the one of the currently active frame (if there is one), the cell must belong to that same frame, and both the read pointer and the corresponding write pointer are set to q. Otherwise, there is an error condition, because apparently no SoF or CoF from the same input immediately preceded this cell, nor does it belong to the currently active frame; the incoming cell should then be flagged as SoF, and treated according to the previous bullet to remedy the situation. Additionally, if the output queue is empty, the read pointer is set to q. (This can only occur if the cell belongs to the frame currently being transmitted, otherwise there is an error condition. If the new cell's input differs from that of the frame currently being transmitted or if no frame is currently being transmitted, an error must have occurred. All cells from this input up to the next SoF should be discarded.)

- EoF: same as CoF, with the addition that the corresponding write pointer is set to *nil* to indicate that the frame has been completed.

Output-queue read operations are executed as follows. If the read pointer equals *nil*, no cell is read, i.e., the queue is idle. Note that this is *not* equivalent to the queue being empty: the queue may be stalled because of absent CoF cells of the frame currently being served. If the read pointer is not *nil*, the cell indicated by the read pointer is read. Depending on the type of the cell being read, the following three cases must be distinguished:

- SoF: the read pointer is updated to the next-CoF entry of the cell. If the next-CoF value equals *nil*, then the read pointer will also become *nil*. This happens when a frame has not yet been completely received by the output queue. The head pointer is set to the cell's next-SoF entry (may be *nil)*. If the head and tail pointer were equal, both are updated to *nil*.
- CoF: the read pointer is updated to the next-CoF entry of the cell.
- EoF: the read pointer is updated to the value of the head pointer. If the head pointer now equals *nil*, the tail pointer is also set to *nil*.

In all three cases one must also check whether the queue entry just read is being pointed to by the write pointer of the corresponding input (the input that the cell being read has arrived on). If this is the case, both the write pointer in question and the read pointer must be set to *nil*. The output queue also keeps track of the input number of the frame currently being transmitted.

Physical implementation

The price we pay for supporting frame mode is that *every* queue entry needs *two* next-cell pointers (one to point to the next SoF, one to point to the next CoF cell), even though only the SoF cells really need both pointers. Fig. 5.9 shows the fields that constitute a single entry in an output queue in such a naive implementation. The switch dimension equals $N \times N$ ports, and the shared memory size equals M cells. Every output queue is of size Q, i.e., the queue can store up to Q entries.

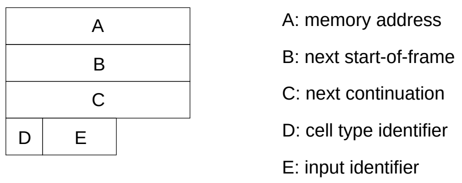


Figure 5.9: Output-queue entry format.

Table 5.1 shows the sizes of the individual queue entry fields, which amount to a total of $\log_2 M + 2\log_2 Q + \log_2 N + 2$ bits per entry, for a total of $NQ(\log_2 M + 2\log_2 Q + \log_2 N + 2)$. We call this implementation alternative '*a*'.

Fig. 5.10 shows the shared memory, consisting of M packet locations numbered 0 through $M - 1$ as shown. Additionally, two output-queue implementation alternatives are shown. The

Table 5.1: Queue entry field sizes.

field	meaning	size (bits)
A	data memory address	$\log_2 M$
B	next start-of-frame address	$\log_2 Q$
C	next continuation cell address	$\log_2 Q$
D	cell type identifier	2
E	input identifier	$\log_2 N$

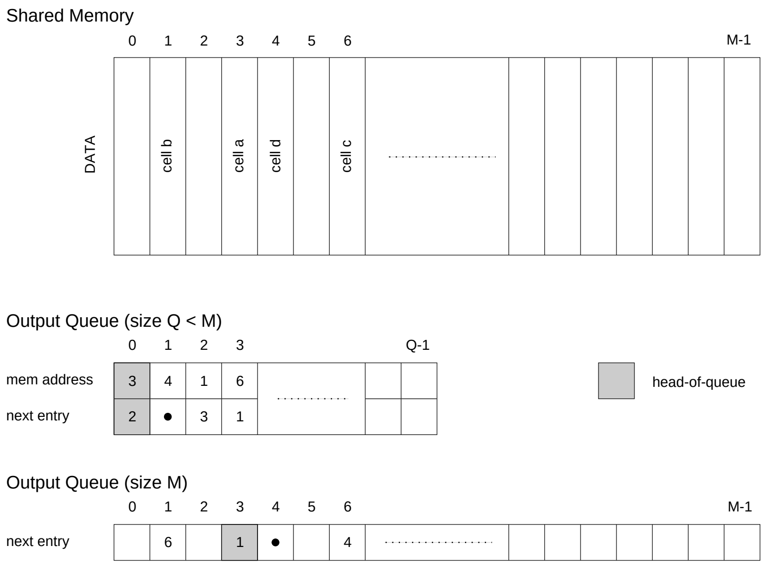


Figure 5.10: Output-queue size vs. implementation.

first has a queue size Q that is smaller than the memory size M. The queue slots are numbered 0 through $Q-1$. This implementation requires two fields per queue entry to build the linked list, namely the memory address of the current entry and the queue slot of the next queue entry. The second implementation, with queue size $Q = M$, requires only the next queue slot to build the linked list, as shown.

An improvement can be achieved by sizing the output queues up to the shared memory size, $Q = M$. This may seem wasteful at first, but there are several advantages as we will see with the help of Fig. 5.10. First, although the queue has more entries, the individual entries can become smaller, because the memory address no longer needs to be actually stored in the queue by instituting a one-to-one correspondence between the place in the queue and the memory address, which is not possible for a queue that is smaller than the memory. On top of that, a smaller queue will need its own free queue with according queue management logic (not shown in Fig. 5.10), which in this case we can dispense with. As an example, the figure demonstrates how the sequence of cells $\langle a, b, c, d\rangle$ is linked up in both implementations. The queue entry size in this implementation equals $2\log_2 M + \log_2 N + 2$, for a total of $NM(2\log_2 M + \log_2 N + 2)$. This is implementation alternative *'b'*.

Further improvements are possible by observing that SoF entries only require a next-SoF pointer, whereas CoF entries only require a next-CoF pointer. Additionally, the input identifier need only be stored with SoF entries. These observations suggest an implementation in which queues for SoF and CoF entries are separated. We can realize a considerable saving by noting that the order of CoF cells of a given frame is identical on all outputs by the definition of a frame; that is, CoF order does not have to be maintained on a per-output basis. This implies that only one CoF queue (also of size M) is needed, along with N SoF queues.

In terms of Table 5.1, this means that the SoF queues store fields B and E, and the CoF queue stores field C. As the type of cell is already implicit in the queue it is stored in, only a 1-bit cell-type identifier to flag EoF cells is required (for entries in all queues).

The resulting implementation requires $NM(\log_2 M + \log_2 N + 1)$ bits for the SoF queues plus $M(\log_2 M + 1)$ bits for the CoF queue, for a total of $(N + 1)M(\log_2 M + 1) + NM \log_2 N$ bits. This is the final implementation alternative '*c*'.

Table 5.2 compares the storage implementation complexities of the three proposed alternatives. Fig. 5.11a shows the total complexity in bytes of storage as a function of M, with $N = 32$ ($Q = M$ for implementation a). Fig. 5.11b shows the relative improvement that alternative *c* offers compared to *a* and *b*, expressed as the ratio $a(N, M, M)/c(N, M)$ and $b(N, M)/c(N, M)$ respectively, for $N = 32, 64, 128,$ again as a function of M.

Table 5.2: Total implementation complexity (in bits of storage) of three output-queue implementation alternatives, expressed in terms of switch size N, memory size M, and queue size Q.

implementation	complexity (bits)
$a(N, Q, M)$	$NQ(\log_2 M + 2\log_2 Q + \log_2 N + 2)$
$b(N, M)$	$NM(2\log_2 M + \log_2 N + 2)$
$c(N, M)$	$(N + 1)M(\log_2 M + 1) + NM \log_2 N$

It can easily be shown that, for a given N, the asymptotic ratios as $M \to \infty$ equal

$$\lim_{M\to\infty} \frac{a(N, M, M)}{c(N, M)} = \frac{3N}{N + 1}, \tag{5.24}$$

and

$$\lim_{M\to\infty} \frac{b(N, M)}{c(N, M)} = \frac{2N}{N + 1}. \tag{5.25}$$

However, because the graphs have a logarithmic character, the convergence is very slow. For practical values of N and M, the complexity reduction factor is in the range 1.25 to 1.75 compared to *b*, and in the range 1.5 to 2.25 compared to *a*.

Similarly, for a given M, the asymptotic ratios as $N \to \infty$ equal

$$\lim_{N\to\infty} \frac{a(N, M, M)}{c(N, M)} = 1, \tag{5.26}$$

and

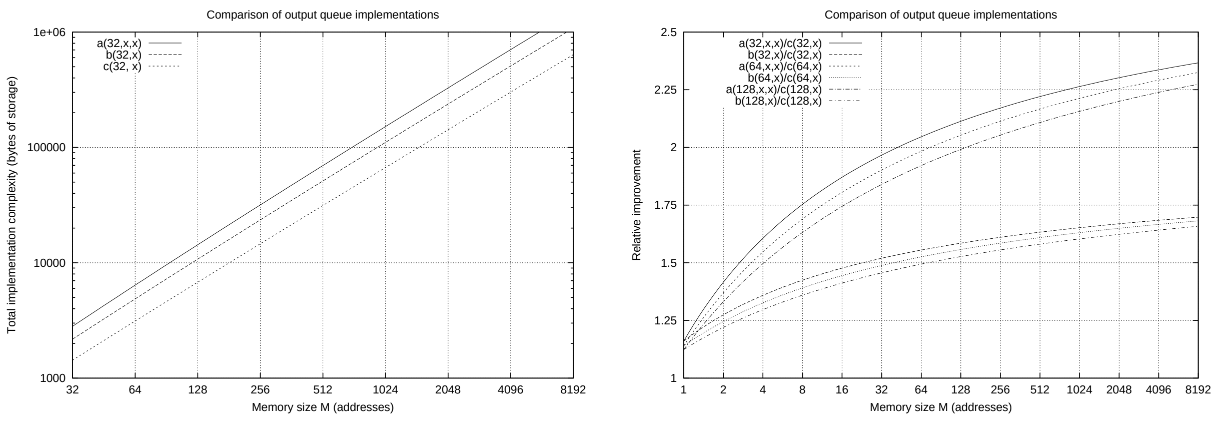
$$\lim_{N\to\infty} \frac{b(N, M)}{c(N, M)} = 1. \tag{5.27}$$

Note that the order in which the limits are taken matters. In fact,

$$\lim_{N,M\to\infty} \frac{a(N, M, M)}{c(N, M)} = 2, \tag{5.28}$$

and

$$\lim_{N,M\to\infty} \frac{b(N,M)}{c(N,M)} = \frac{3}{2}. \tag{5.29}$$



(a) Implementation complexities as a function of memory size, $N = 32$.

(b) Relative improvement of alternative c compared to a and b.

Figure 5.11: Comparison of the storage complexity of the three output-queue implementation alternatives.

To avoid cyclic waiting deadlock conditions, the strict time-ordering condition of Definition 16 must be satisfied. This can be achieved as follows: The output queues follow a strict FIFO discipline on a per-frame basis. Frames that arrive at one output queue simultaneously are ordered according to a predetermined order of their respective input numbers. This can be round robin, or any other suitable ordering scheme. The key point in realizing the strict timing order is to ensure that this order is identical at all outputs. However, the order of inputs must not be identical in subsequent time slots. For reasons of fairness, the order may be suitably rearranged at every new time slot.

5.10 Conclusions

We have presented a way to integrate support for frame-mode operation in an output-queued shared-memory cell switch. It has been shown how deadlock situations involving both unicast and multicast frames and frames of multiple priorities can be prevented from occurring by suitably modifying the output-queue and shared-memory grant signals. An overview of how to implement this scheme has been given.

Some of the major advantages of this approach are

- the seamless integration, from a switching point of view, of unicast and multicast frames,
- the cost-efficient merging of three highly desirable features into one switching fabric.

Whereas the traditional cell switch's application range was usually quite limited, the addition and integration of frame mode, multicast, and QoS functionality enables this type of switch fabric to find applications in a wide range of products, including

- the traditional ATM world,
- the router world (native support for variable-length TCP/IP packets),
- the Ethernet world (native support for variable-length Ethernet frames),
- the (mainframe) server world (as a high-speed interconnect to cluster servers together, instead of the traditional mesh), or
- the parallel processing world (as a replacement for the traditional bus structure to interconnect processors and memory).

Chapter 6

A Practical Application: The PRIZMA Switch

This chapter presents a practical application of the concepts developed and studied in the preceding chapters within the PRIZMA packet switch family. An overview of the PRIZMA history is given, and features of the various switch generations and requirements for the next generation are discussed. Finally, this work is placed into the context of the PRIZMA project.

6.1 The PRIZMA Switch Family

The history of the PRIZMA[1] switch family starts in the late 1980s, when research at the IBM Zurich Research Laboratory was conducted towards a "high-performance switch fabric for integrated circuit and packet switching" [Ahmadi89a]. The basic structure of the fabric presented in [Ahmadi89a] is already that of a self-routing, non-blocking, output-queued fixed-length-packet switch. Scalability was as a key focus of the fabric design and both the single- and multi-stage port expansion modes were presented. With 16 ports running at speeds in the range of 32 Mb/s, it was foreseen to be implemented in a single chip. However, the name PRIZMA did not exist yet, and this fabric has never been implemented as such.

By 1993, this architecture had evolved into what has become the first member of the PRIZMA family, as described in [Denzel95]. Its basic design points remain unchanged, with one notable exception: the output queues were now realized as a dynamically shared memory, which led to better memory efficiency. More emphasis than before was put on the scalability issues, now encompassing three dimensions, namely the number of ports (*port expansion*, both single- and multi-stage, as before), port speed (*speed expansion*, with the introduction of the master/slave concept), and memory size (*performance expansion*).

This architecture has been implemented as a 16×16 switch fabric with a 400 Mb/s port speed resulting in an aggregate throughput of 6.4 Gb/s.[2] Its shared memory can store 128 packets of

[1]The official IBM designation of the PRIZMA switch family is *PowerPRS Packet Routing Switch*. PRIZMA is the name that has traditionally been used internally.

[2]We adopt the convention that the aggregate throughput of a switch with N inputs and N outputs, each with a speed of B b/s, equals NB.

64 bytes[3] each. It offers support for multicast and link paralleling (2- and 4-way), but not for priorities. The implementation complexity is on the order of 2.4 million transistors in a 472-pin package.

A cost-reduction iteration of the original PRIZMA has been done, called PRIZMA Prime ("PRIZMA-P"). This is not much more than a technology mapping from CMOS4S to CMOS5S0, leading to a smaller and cheaper chip. Additionally, this chip is equipped with support for two traffic priorities. However, there is *no overtaking* of low-priority packets by high priority packets, so that the output queues can still be implemented as simple FIFOs. The priorities are only used for flow control by means of programmable output-queue thresholds.

PRIZMA was used in a range of IBM products, among which are the 8265 ATM Backbone Switch [Alaiwan99], the 8260 Multiprotocol Switching Hub, the 8220 Broadband Switch, and the 8285 ATM Workgroup Switch.

The second generation of the PRIZMA switch ("PRIZMA-E"), internally called PRIZMA Atlantic, was developed next. From a black-box point of view, the most important changes with respect to its predecessor are the increase in port speed, the better performance resulting from a larger shared memory, and four preemptive traffic priorities. Port speed, originally targeted at 2 Gb/s, turned out to be 1.6 to 1.77 Gb/s, depending on internal clock speed.

From the inside however, there are significant architectural differences between the two, especially with respect to the shared memory. Whereas the original PRIZMA was based on parallelism to obtain the bandwidth to the shared memory (all inputs and outputs can have access to the memory simultaneously and independently through parallel input and output routing trees and multi-ported memory), the Atlantic architecture reverts to a shared-bus architecture to obtain the required shared-memory bandwidth (memory access is performed in a RR time-division-multiplexed manner, instead of space-division-multiplexed, for implementation cost reasons), see also Section 3.6 and Fig. 3.18. More PRIZMA-E implementation details can be found in [Colmant98c]. This shared-bus approach will certainly be infeasible for the new third-generation PRIZMA, for the simple reason that at the targeted port speeds the required bus bandwidth can no longer be realized anymore. Therefore, a return to a more parallelized architecture is inevitable.

The priorities are preemptive on the transmission side, that is, higher priorities always overtake lower priorities in an output queue. To this end, the output queues are implemented as linked lists, so there are four logical queues per output. Programmable output-queue and memory-full thresholds are employed to establish per-priority flow control, which is transmitted to the input adapters. There is one occupancy counter per output queue, and one for the shared memory, as opposed to counters *per priority*. Therefore, we call the thresholds *nested*.

An interesting implementation detail is that the actual PRIZMA-E chip consists of two 800 Mb/s-port Atlantic switch cores running in speed-expansion mode, integrated on the same chip. However, because the cores are already running in speed expansion, the minimum logical unit size from a user point of view is 32 bytes, instead of 16.

PRIZMA-EP is basically a 32×32 version of PRIZMA-E with a larger shared memory and improved priority management, which enables guaranteed bandwidth (GB) per priority, something

[3]The mininum size of the data unit the switch can handle is 16 bytes. This size determines the speed requirement of the control section, and basically limits how fast the switch can run. This minimum size is called the *logical unit size* (LU size) in PRIZMA terminology.

that was not possible with PRIZMA-E's preemptive priority transmission scheme. In addition, PRIZMA-EP supports 4-way link paralleling.

Finally, PRIZMA-T is the designation for the next member of the PRIZMA switch family, which is currently under development. Table 6.1 lists the salient features of all PRIZMA generations and versions.[4,5]

Table 6.1: An overview of three PRIZMA generations—from the original PRIZMA, via the current PRIZMA-E and EP, to the future PRIZMA-T (projected values).

PRIZMA gen.	I.		II.		III.	
		"P"	"E"	"EP"	"T"	
Input ports	16	16	16	32	32	
Output ports	16	16	16	32	32	
Port speed	0.4	0.4	1.77	2.0	8.0	Gb/s
Aggr. thruput	6.4	6.4	28.4	64	256	Gb/s
Min. LU size	16	16	32	32	16	bytes
Max. ext SpEx	4	4	2	2	4	
Memory size	128	128	256[a]	1024[b]	1024	packets
Priorities	none	2[c]	4[c]	4[d]	4[d]	
Link paralleling	yes	yes	no	yes	yes	
CMOS techn.	4S	5S0	5S6	6SF	(TBD)	
Feature size	0.8	0.35	0.32	0.25	(TBD)	μm
L_{eff}	—	—	0.25	0.18	(TBD)	μm
Complexity	2.4	2.4	3.9	37	(TBD)	10^6 fets
Package	472	472	624	1088	(TBD)	pins
Throughput						
(single chip)	**6.4**	**6.4**	**28.4**	**64**	**256**	Gb/s
(w/ full SpEx)	**25.6**	**25.6**	**56.8**	**128**	**1024**	Gb/s

[a] 512 packets in external speed-expansion mode.
[b] 2048 packets in external speed-expansion mode.
[c] strict priorities.
[d] virtual lanes, with guaranteed bandwidth provisioning.

6.2 PRIZMA Features

In this section we will provide an overview of the most significant features of the PRIZMA family of switch chips.

The main features are listed below.

- It is primarily designed to switch fixed-size data units.

[4]LU size: Logical Unit size; this is the minimum size of the logical data unit that the switch can handle.

[5]SpEx: Speed Expansion factor; this is an expansion mode in which multiple switches are combined to support a higher port speed. The speed-expansion factors here assume a maximum data unit size of 64 bytes.

- Single chip solution (“Switch-on-a-Chip”).
- High performance: high sustainable throughput under a wide range of traffic patterns and low latency under non-saturated conditions.
- Losslessness: no packets should be lost because of buffer overflow conditions. This can be ensured by using proper flow-control schemes. This property is especially important in frame mode because there losing one packet implies loss of an entire frame.
- Preservation of packet sequence: packets belonging to a given flow must be delivered in the sequence they arrived in.
- Scalability: This is one of the key requirements to enable great flexibility and expandability using a single building block. The scalability features of the former and current members of the PRIZMA family have proven to be key contributors to PRIZMA’s success in the marketplace. Scalability comes in several forms:
  - Port speed expansion (speed expansion, or spex for short): this mode allows N switch chips to be combined into a single fabric with N times the port speed.
  - Single-stage port expansion: this mode allows the construction of switch fabrics with N times the number of input and output ports, while maintaining the single-stage property. It comes at the cost of requiring N^2 switch elements.
  - Multi-stage port expansion: when building very large switch networks, this is a less expensive—but also less performant—solution to increase the number of switch-fabric ports.
  - Link paralleling: this useful feature, which was supported in the original PRIZMA, but disappeared with PRIZMA-E, has been brought back in PRIZMA-EP. Link paralleling entails the ability to combine L physical switch ports to act as one logical port with L times the port speed. This can be done for inputs and outputs separately and should be configurable in a flexible manner. This feature adds another dimension in terms of scalability—one switch can handle much fatter pipes without requiring any additional hardware.

  Especially the speed expansion mode is considered to be of great importance, because of its capability to extend the lifespan of the switch product further out into the future.
- QoS support, encompassing guaranteed bandwidth provisioning, traffic priority support, and support for best-effort traffic.
- Multicast support: incoming packets or frames must be able to be duplicated to multiple output ports according to a multicast routing ID, which can be available directly in the form of a header bitmap or indirectly through a lookup table.
- Frame mode: support for switching long packets. This has not yet been implemented in any PRIZMA switches.

Most of the above features have been described in detail in the preceding chapters. In the next sections, we will discuss the features that have not yet been treated in-depth before and we will evaluate their implications on switch design.

6.3 Scalability

In this section, the speed- and port-expansion modes introduced above will be elaborated on. Link paralleling is treated separately in Section 6.6.

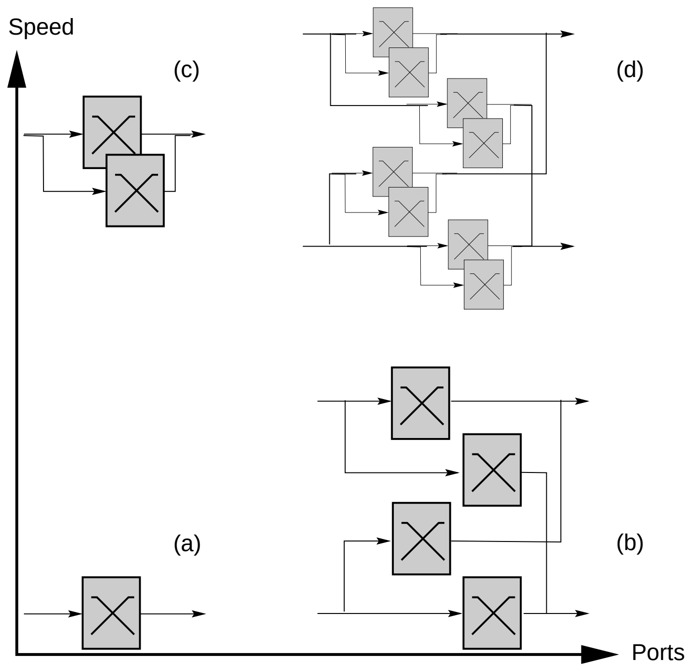


Figure 6.1: Speed-, port- and combined expansion modes.

Table 6.2: Speed-, port- and combined expansion modes.

figure	expansion mode	#ports	speed
(a)	none	N	S
(b)	single-stage port expansion	$2N$	S
(c)	speed expansion	N	$2S$
(d)	(b) and (c) combined	$2N$	$2S$

Figure 6.1 shows two dimensions in scability, one in terms of port speed (vertical axis), one in terms of number of ports (horizontal axis). Table 6.2 describes the expansion modes. Note how expansion modes can also be combined to achieve systems with both more ports *and* higher port speed.

6.3.1 Speed expansion

As mentioned, the ability to do speed expansion is a key feature. First, we will have a closer look at how speed expansion works conceptually, see Fig. 6.2. In principle, speed expansion entails the capability to stack multiple switch chips in parallel to effectively widen the data path and create a switch fabric with the same number of ports but higher port speed. This concept is similar to that of the *Parallel Packet Switch* (PPS) [Iyer00] because the external line rate is higher than the rate the individual switch memories can support, thus allowing linear scaling of the port speed.

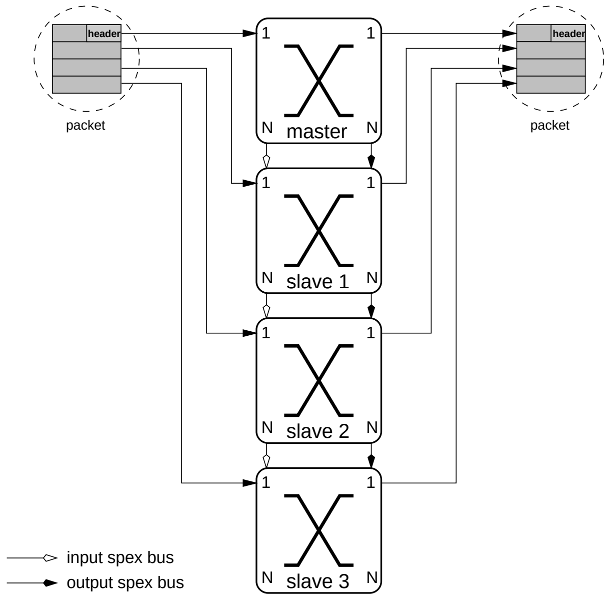


Figure 6.2: Four-way speed expansion: the packet is split into four chunks of equal size that are switched in parallel by four separate switches. Only the master receives the packet header.

For a speed expansion factor of S, we need S switches. The trick is that every one of these S switches handles the $1/S$-th part of each packet (it is the responsibility of the input adapter to split up the packets properly). One of the switches is called the *master*, the others are the *slaves*. The master handles the packet headers and drives all the slaves, whose control sections are disabled as they only handle data. For each arriving packet, the master provides the write addresses to the slaves, and for each departing packet it provides the read addresses. These addresses are passed over the *speed expansion bus*. The output adapter reassemble the S pieces of each packet to reproduce the original packet before passing it on.

Owing to the fully-shared-memory architecture, the write addresses can be pre-fetched and passed to the slaves before the corresponding packets have arrived at the switch, so that a packet can be stored immediately in all chips. The shared-memory architecture allows this because the write address is independent of the actual destination of the incoming packet. This reduces latency at the input side. Unfortunately, this does not work at the output side, because the appropriate read address can only be passed to the slaves when the packet has arrived at the output queue.

The drawback of speed expansion is that the packet size granularity as seen by the user is multiplied by S. This implies that if a minimum granularity of B bytes must be supported, each individual switch must be capable of handling data units of size B/S. The smaller the data unit, the more complex the control section becomes, because it has to handle the same amount of reads and writes in a shorter time. Given the constraints of the technology, this puts a lower limit on the unit size that can be supported by one switch, which then determines, given the granularity to be supported in speed expansion mode, what the maximum speed expansion factor is. For instance, if a minimum packet size of 64 bytes is to be supported and the shortest data unit that each switch chip can handle, given the technology, is 16 bytes, then the maximum speed expansion factor is 4.

The second problem lies with the speed-expansion bus, which carries the addresses from master to slave(s). In PRIZMA and PRIZMA-E, it is possible to accomplish this within one LU cycle, but as the port speed increases, the time it takes before the addresses arrive at the slaves (transmission delay plus transmit and receive logic) becomes significant, so that this approach can no longer be used, and pipelining must be applied. Secondly, in particular with larger speed-expansion factors, there is the problem of connecting the speed-expansion bus from a single master to multiple slaves. At the envisioned speeds, there is no I/O technology available that can reliably drive point-to-multipoint connections. Possible solutions are duplicating the speed-expansion bus on the master (requires many extra pins on master), or chain master and slaves together in a ring or bus structure (requires extra pins on slaves; each chip will have a speed-expansion input and output bus to pass on the addresses).

6.3.2 Single-stage port expansion

Single-stage port expansion is a way to expand the number of switch fabric ports while maintaining the single-stage property. Figure 6.1b shows how K^2 switch elements can be combined to produce a switch fabric with K times the number of input and output ports. Additionally, the following external functions must be provided, as indicated in Fig. 6.3 for $K = 2$:

- At every input port a splitting function is required to duplicate the incoming packets to one or more of the K switch elements it is connected to and insert the correct bitmaps for each switch.
- At every output port a merging function is required to merge the traffic from K switch elements by performing transmission arbitration among these.

Inside each switch element, packets are filtered out based on their destination, or alternatively this function can be performed by the splitters. Refering to the numbers in Fig. 6.3, Table 6.3 specifies which switch is responsible for switching the packets for a given input/output port combination. This is sometimes also referred to as Block-Crosspoint buffering [Katevenis95].

Table 6.3: Single-stage port expansion: each switch only switches traffic for a subset of the input and output ports. The port numbers refer to the switch fabric as a whole.

switch	inputs	outputs
1	$1 \cdots$ N	$1 \cdots$ N
2	$1 \cdots$ N	N+1 $\cdots$ 2N
3	N+1 $\cdots$ 2N	$1 \cdots$ N
4	N+1 $\cdots$ 2N	N+1 $\cdots$ 2N

For large fabrics ($K \geq 4$) this is obviously a very expensive method because of the quadratic dependency: for an $M \times M$ switch fabric, with $M = KN$, $(M/N)^2 = K^2$ switch elements are required.

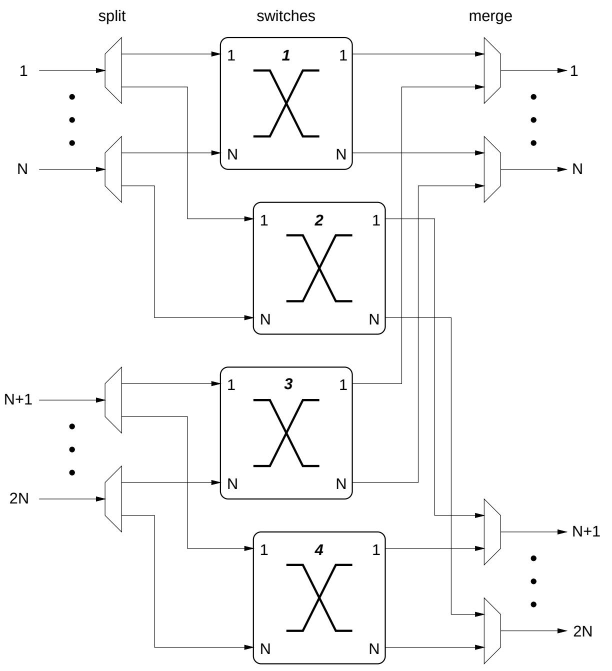


Figure 6.3: Single-stage port expansion, including splitting and merging functions.

Additionally, there are some fairness concerns regarding the arbitration at the merge point. The core of this problem is that the output queue for one given fabric output is actually distributed over physically separate queues in two (or more) switch elements.

6.4 Quality-of-Service Support

Both in telecommunications and in parallel-processing environments there is a need for QoS guarantees, in terms of guaranteed bandwidth, packet-loss rates, packet delay, packet delay jitter, etc., the main requirement being guaranteed bandwidth. To satisfy these requirements, some type of scheduling or arbitration algorithm is required.

We call the grain of QoS support a *traffic class*.[6] A traffic class is essentially a “slice” of the switch. Supposing we have F classes, the switch is conceptually divided into F slices, one per class. Ideally, the QoS experienced by one class should not be affected by the other classes, but in practice, this is difficult to achieve as physically the classes will be sharing resources in the switch. To identify which class a packet belongs to, the packet carries a class identifier.

Note that because of speed and silicon restrictions, we cannot do very complex scheduling on large numbers of traffic classes.

The original PRIZMA did not offer any QoS support. PRIZMA-P and E employ a preemptive priority mechanism, whereas PRIZMA-EP employs a table-based WRR mechanism to provide bandwidth guarantees, derived from the virtual lane mechanism. We will describe these mechanisms below.

[6]Often, the term *priority* is used interchangeably, although this is really only one flavor of traffic-class differentiation.

In the PRIZMA implementation, the output queues are implemented as linked lists, such that there is one linked list per output per traffic class. Packets from the same traffic class are always served in FIFO order. Flow control information is also provided on a per-class basis, i.e., both memory grant and output-queue grant are provided per class. The PRIZMA switches employ *nested thresholds*: per resource (shared memory or output queue) there is one occupancy counter counting *total* resource occupancy and multiple thresholds (one per resource per class). When the counter exceeds a given threshold, the grants for the associated class and all classes with lower thresholds are removed.

Best-effort traffic is only served if resources allow. Naturally, no bandwidth guarantees or any other QoS requirements apply to this type of traffic. If a best-effort packet is flagged as droppable, the switch can choose to drop rather than delay it in case there is a lot of contention.

6.4.1 Preemptive priorities

Each traffic class is assigned one out of N_p strict priorities, 0 being the highest, $N_p - 1$ the lowest priority. When selecting a packet for transmission, packets of a higher priority class are always given precendence over those of a lower priority class.[7]

One major disadvantage of this scheme that has been recognized in trying to map ATM requirements onto it is that the lower priorities can be completely starved, so there can be neither bandwidth nor delay guarantees for them at all; if high-priority packets are available in every time slot, low-priority ones contending for the same resources will never be served. For this reason, the preemptive priority scheme on its own is considered insufficient.

6.4.2 Virtual lanes

The virtual-lane (VL) arbitration is a type of weighted round-robin scheduling, based on a time-division multiplex structure consisting of N_s packet slots. Each of N_{vl} virtual lanes can be guaranteed a fixed part of each N_s-slot superframe, the reserved (guaranteed) bandwidth R^i_{vl}, $0 \leq i < N_{\text{vl}}$.[8] Each virtual lane has one of N_p priorities associated with it. Should any slots remain unreserved, then these will be filled according to an arbitration based on the priority of each VL. The remaining slots will be filled with packets from the highest priority VL. If no more packets are available from this VL, the remaining slots are filled with packets from the VL with the next-lower priority, and so on, until no more slots remain. If there are multiple VLs with equal priority, packets from these VLs are interleaved on a per packet basis.[9] This scheme is illustrated in Fig. 6.4.

Note that when no bandwidth is reserved for any lane, this scheme reduces to the preemptive-priorities scheme presented in Section 6.4.1 if the VL priorities are programmed accordingly. However, in contrast to the latter scheme, here it is possible to guarantee a certain bandwidth to each flow by setting the R^i_{vl} parameters accordingly.

[7]This scheme is implemented in the PRIZMA-E switch with $N_p = 4$ priorities.

[8]The values R^i_{vl} must satisy the following condition: $\sum_{i=0}^{N_{\text{vl}}-1} R^i_{\text{vl}} \leq N_s$.

[9]The SP2 switch implements this algorithm with $N_{\text{vl}} = 8$ virtual lanes, $N_p = 4$ priorities and a multiplex frame size $N_s = 64$.

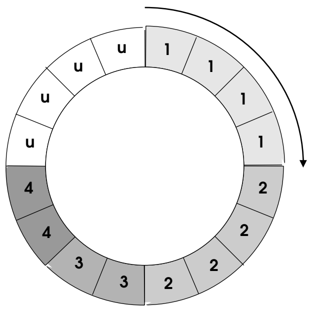


Figure 6.4: A sample VL configuration with $N_{\text{vl}} = 4$ VLs, arbitration period $N_s = 16$, $R^1_{\text{vl}} = R^2_{\text{vl}} = 4$, $R^3_{\text{vl}} = R^4_{\text{vl}} = 2$, leaving four slots unreserved. The effective guaranteed bandwidth amounts to 25% for lanes 1 and 2, 12.5% for lanes 3 and 4.

A disadvantage is the fixed frame size N_s, which limits the bandwidth reservation granularity. Also, a packet of a virtual lane whose reserved-bandwidth slots have just passed may, in the worst case, incur a delay of $N_s - 1$ slots before being transmitted. Enlarging the frame size N_s improves granularity, but at the same time exacerbates the latter effect.

6.4.3 Credit table

PRIZMA-EP employs a virtual-lane-like arbitration mechanism referred to as *credit table*.[10] There are N_{ct} traffic classes (lanes); each lane has a fixed priority. Each output has a table consisting of N_s entries, each entry consisting of a valid bit and one of N_{ct} lane identifiers. A pointer to the current entry in maintained for each table. When an output-queue read operation is initiated, a packet will be fetched from the output queue according to the following rule: if a packet of the lane indicated by the current table entry is present, that packet is served, otherwise a packet from the highest-priority non-empty lane is served. In each cycle, the current-entry pointer is moved to the next table entry in a RR fashion, regardless of the outcome of the previous cycle.

The table can be programmed to assign fractions of the available output link bandwidth to lanes as desired.

In the PRIZMA-EP implementation, $N_{\text{ct}} = 4$, and $N_s = 256$. The main disadvantages of the credit table scheme are that the granularity of bandwidth allocation is limited by $1/N_s$ and that the class priorities are fixed. The latter problem can be fixed by making the class priorities programmable, and applying RR service among lanes of equal priority, as is done in the virtual lane scheme.

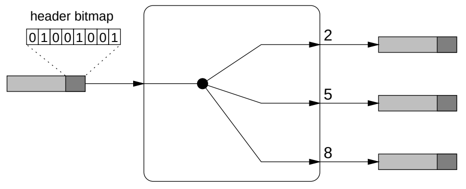
6.4.4 Best-effort traffic

Best-effort traffic is traffic for which no QoS guarantees are made at all. If resources permit and no other packets are available, BE packets can be served. If flagged as droppable, they can be discarded. Every traffic class can have a best-effort sub-class. PRIZMA-EP implements a BE dropping scheme.

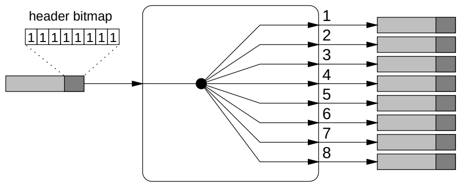
[10]The credit table scheme has nothing to do with credit-based flow control.

6.5 Multicast Support

Efficient duplication of incoming packets and frames to multiple destinations (see Fig. 6.5) to support point-to-multipoint communication is an important requirement. This topic has been treated in-depth in Chapter 4, and in Chapter 5 a method to support deadlock-free multicast in conjunction with long-packet support (frame mode) has been proposed.



(a) Multicast operation: a copy of the incoming packet is transmitted on multiple (in this case three) outputs, as indicated by the header bitmap.



(b) Broadcast operation, a special case of multicast: the incoming packet is duplicated to *all* outputs.

Figure 6.5: Multicast and broadcast switch functions.

6.6 Link Paralleling

The link-paralleling (LP) feature of the original PRIZMA allows a logical grouping of ports, input ports as well as output ports, to obtain fewer (logical) ports with a port speed that is a multiple of the physical port speed. This feature has been recognized as highly desirable because of the additional flexibility it offers to the user. With link paralleling all paralleled ports receive full packets with headers, whereas with speed expansion only the master receives a header. Link paralleling works at a coarser granularity, paralleling entire packets, whereas speed expansion increases physical port speed by paralleling chunks of packets. Both mechanisms can be combined to obtain larger expansion factors.

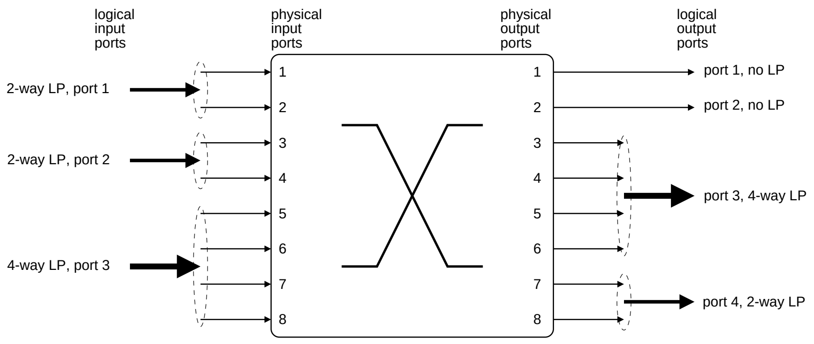
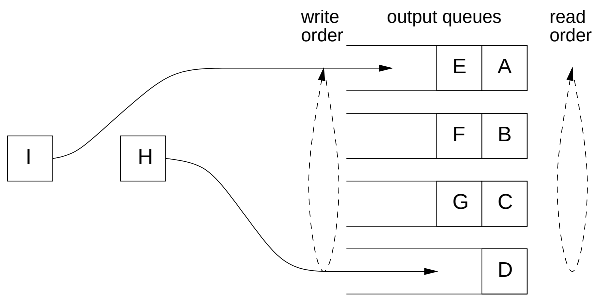


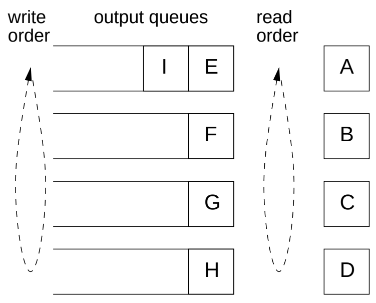
Figure 6.6: Link paralleling: Of the 8×8 switch depicted here, the input ports are grouped into three sets: $\{1, 2\}$, $\{3, 4\}$, and $\{5, 6, 7, 8\}$. On the output side, ports 1 and 2 are not link-paralleled, but $\{3, 4, 5, 6\}$ and $\{7, 8\}$ are.

Grouping inputs and outputs together to form a logical port requires that packet sequence be maintained across all paralleled ports. At the input side, this means that packets must be sent by the adapters on the paralleled inputs in a predefined order (first packet on port 1, second on port 2, etc.). The switch must ensure that these packets are stored in the same order in the destination output queue(s). At the output side, packet order must be maintained across output queues of paralleled ports, and packets must be transmitted on the paralleled ports in a predefined order (again, first packet on port 1, second packet on port 2, etc., in a RR fashion).

Implementation-wise, it means that it must be possible to group output queues together, and more importantly, packet order must be maintained. This may be achieved by writing arriving packets to the output queues of a group in a RR fashion, keeping track of the queue to which the last packet was written. Consequently, the queues of one group are also read out in a RR fashion, so that packet order is guaranteed. Figure 6.7 illustrates this mode of operation. Switch fabric adapters at the in- and output side of the switch must be aware of link paralleling and transmit c.q. receive packets on the paralleled links according to these rules to maintain packet sequence.



(a) Packets are stored in a RR fashion in the link-paralleled output queues. Packets H and I will be stored at the positions indicated.



(b) Packets are also read in a RR manner. After four reads, the queue status is as depicted.

Figure 6.7: Implementation of link paralleling in an output-queued switch architecture.

6.7 The Context of the PRIZMA Project

This section places the work described in this dissertation in the larger context of the PRIZMA project.

6.7.1 Design methodology

Figure 6.8 illustrates the design methodology employed in the PRIZMA project, in particular in the PRIZMA-E and EP projects.

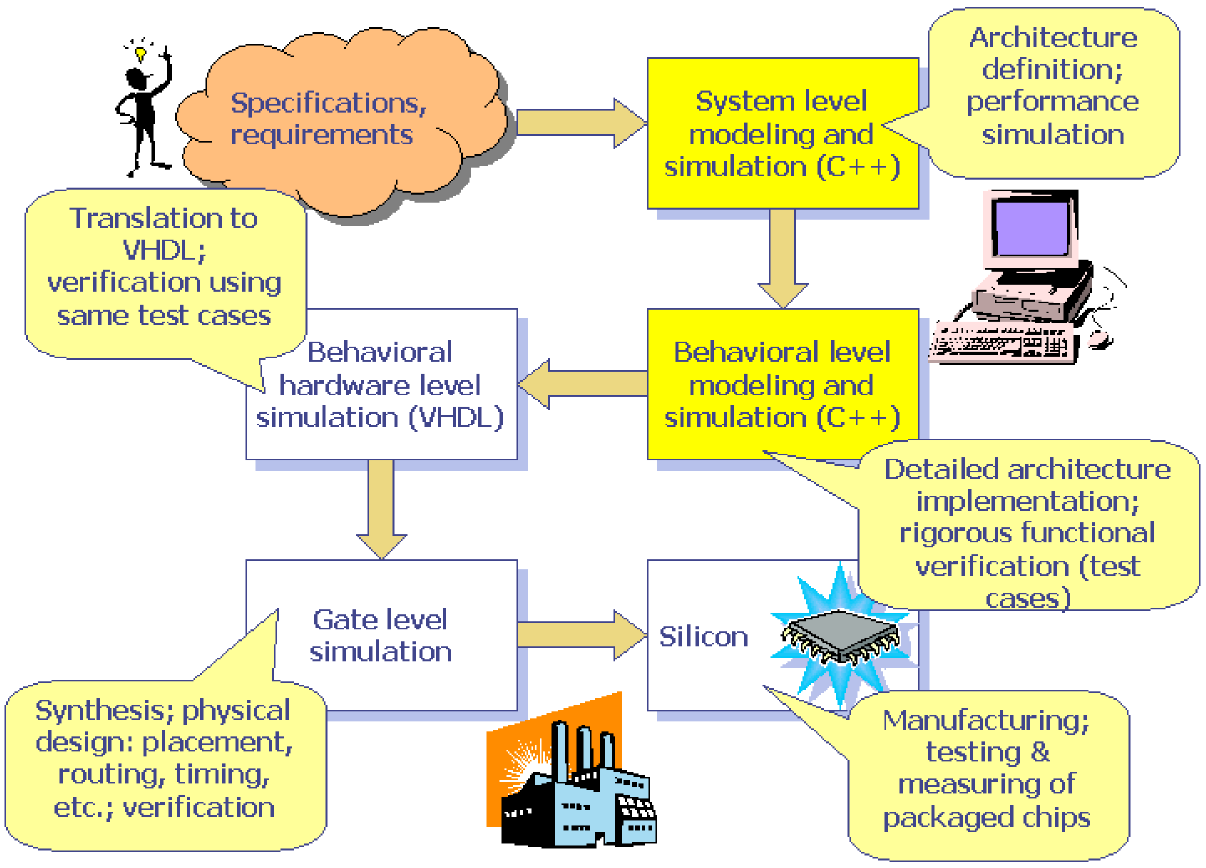

Specifications, requirements
System level modeling and simulation (C++)
Architecture definition; performance simulation
Translation to VHDL; verification using same test cases
Behavioral hardware level simulation (VHDL)
Behavioral level modeling and simulation (C++)
Detailed architecture implementation; rigorous functional verification (test cases)
Gate level simulation
Silicon
Synthesis; physical design: placement, routing, timing, etc.; verification
Manufacturing; testing & measuring of packaged chips

Figure 6.8: Design methodology. The contributions of this dissertation are in the area of architecture definition and system level modeling and simulation.

The work presented in this dissertation is positioned at a high level, just below the specifications and requirements phase, leading to an initial architecture that will be refined and improved by means of performance evaluation. This evaluation consists of simulating a high-level, abstract model of the system, with just enough detail to allow the impact of the system parameters under study on the system performance to be analyzed. In our case, this model and the simulation engine are written in C++. The results of the simulation are evaluated and fed back into the design process to improve and optimize the architecture.

The advantages of such a performance evaluation step in the design process are clear: it is an inexpensive and fast way to evaluate performance of a proposed architecture, test hypotheses on such an architecture, evaluate the impact of various system parameters, as well as their interaction, and gain confidence in system correctness.

The main limitations are that detailed questions cannot be answered with a high-level model and that assumptions on system input (stimuli) are often only distantly related to reality.

Chapters 3 and 4 have described how simulation allows architectural issues to be addressed and new ideas evaluated. This leads to a better understanding of the system and its behavior, thus paving the way for further improvement.

6.7.2 Impact of the work

The work presented in this dissertation has primarily impacted the PRIZMA project in the following two ways:

- It has led to a thorough understanding of the application and performance of VOQ in a CIOQ system using a small output-buffered switch.
- It has served as a catalyst for new approaches to the implementation of said output-buffered switch, leading to more scalable designs.

The former aspect has been discussed in-depth throughout Chapters 3 and 4, so let us now elaborate on the latter aspect.

6.7.3 Shared-memory implementation

As described in Section 3.6, one of the biggest implementation challenges is the shared memory, in particular the routing trees to and from the memory. Fig. 6.9 illustrates this problem. Taking as an example a 32×32 switch with storage for $M = 1024$ packets, an interconnect is required that connects *each* input with *each* memory location. Hence, 32 1-to-1024 (de)multiplexers are required at both the input and the output side. The main complexity in these 1-to-1024 routers is the repowering required to drive the entire tree.

Additionally, because the switch cannot be clocked faster than 2 ns (a limit imposed by technology), the datapath must be 16 bits wide to obtain a port speed of 8 Gb/s (original PRIZMA-T plan-of-record). This means that $32 \cdot 1024 \cdot 16 = 524,288$ wires cross the vertical plane between inputs and memory and the same number between memory and outputs. This clearly poses immense wiring problems that, although not unsolvable (a very special arrangement of the individual memory banks has been found that provides enough room to route all the wires), render this implementation unscalable to more ports or larger memories, because the number of wires grows as N^3. Also, increasing the port speed requires increasing the width of the datapath (e.g., in the above example a port speed of 16 Gb/s requires a 32-bit-wide datapath) because the clock speed cannot be increased, which further complicates the wiring problem.

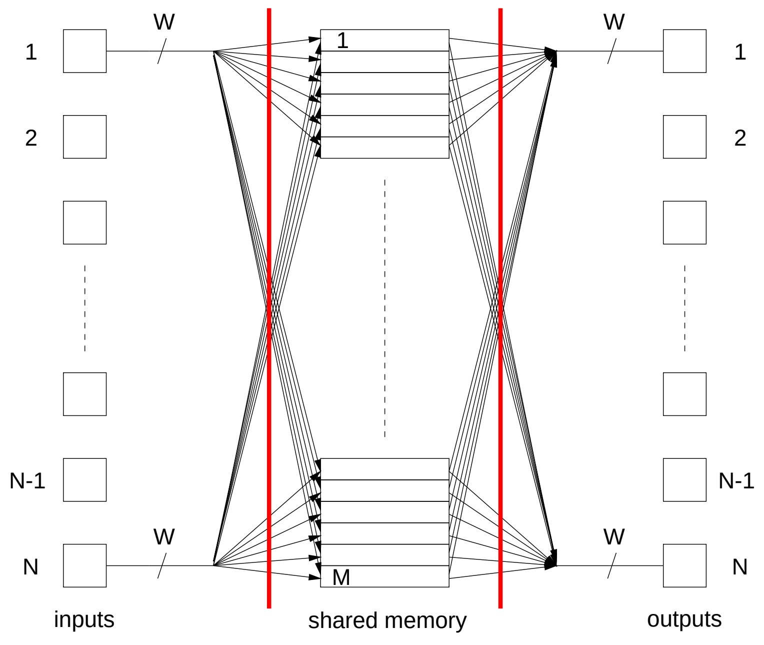


Figure 6.9: Shared memory wiring complexity.

6.7.4 Partitioning per output

The core of the wiring problem is that each memory location must be reachable from each input and output, i.e., the memory is fully shared among all inputs and outputs. This architecture stems from the old adage that full sharing yields best performance. However, as the results in Chapter 3 have demonstrated, this is not true for the proposed CIOQ architecture. In fact, the results suggest that partitioning the memory per output offers best delay–throughput performance. This modification would significantly reduce the size of the output wiring, from 32 1024-to-1 muxes to 32 32-to-1 muxes, for instance. Logically, this constitutes a reduction in complexity by a factor of 32, but in practice the reduction is more on the order of a factor of 1024, because no repowering is required here.

However, in practice, partitioning the memory so that each output has a dedicated block of memory has some serious drawbacks. For one, the speed-expansion mode (introduced in Section 6.3.1), does not work if the memory is partitioned per output: the input controllers cannot prefetch memory addresses, because they do not know in advance to which output the incoming packet is destined. Because the slaves do not receive any header information, an additional bus would be necessary to tell the slaves to which output the incoming packet is destined, introducing additional latency. Furthermore, this approach also presents a drawback with multicast, because packets (including payload!) must be duplicated to all destination outputs. Also, output-queue grant transmission becomes much more critical, because violation of output-queue grant now implies packet loss.

6.7.5 Partitioning per input

An alternative approach is to partition the memory per input, that is, each input has a memory that can store N packets, but all packet locations can be accessed by all outputs. This significantly reduces the wiring at the input, similar to the partitioning described above. However, a full-size output router is required. Figure 6.10 illustrates this architecture. The control section can either remain centralized as in the fully-shared-memory implementation, or it can be distributed as well, so that there are N smaller control sections, one per input buffer. The advantage of the latter is that the write bandwidth of the individual output queues is reduced, because only one packet per cycle can enter and because the queues are much shorter. The free queue at each input (not shown in the figure), however, must still be able to store N addresses in each cycle. For performance reasons, flow-control information (queue full and queue empty information) must be based on the *global* output-queue state.

In addition, an arbiter for each output is required to arbitrate among the N queues for that output. Simulations have shown that a centralized control section employing an oldest-cell-first (OCF) policy leads to lower delay variation because the FIFO order is maintained.

Speed expansion works just as before because the prefetching of addresses is done on a per-input basis, i.e., no a-priori knowledge of the incoming packet destinations is required. When operating in frame mode the possibility of shared-memory deadlocks no longer exists, because frames from different inputs can no longer interfere with each other. Also, multicast requires no duplication of data into more than one memory, and thus works exactly as with the fully shared memory

The performance implications of this partitioning have been studied by means of simulation. The results show that under the simulated traffic conditions there is hardly any impact in terms of delay–throughput performance, with $M = N$ packets per input, so that the total amount of memory in the switch is not changed.

6.7.6 Buffered crossbar

Having eliminated the input router, the questions remains whether further improvements are possible.

The issue of wiring complexity can be completely solved by partitioning the memory per input *and* per output. This leads to a system architecture with a square array of memories as depicted in Fig. 6.11, typically referred to as a buffered crossbar (see Section 2.4.1). While this implementation has no complex wiring at all, it still has all the disadvantages associated with partitioning the memory per output.

This can be solved as follows: all memories associated with one input line are viewed not as N distinct memories, but as N banks of the same memory, with M packet locations per bank.

This architecture differs significantly from the regular buffered crossbar: An incoming packet is duplicated to the same address in *all* the memories associated with the input it arrives on. With each input N queues are associated, one per output. The pointer to the memory address is stored only in those queues the packet is destined to. On the output side an arbiter (one for each output) decides from which of the associated queues a packet is transmitted (several policies are possible, such as OCF, RR, LQF). Note that this architecture is *identical* to that with partitioned memory per input (as described in Section 6.7.5) if the same output-queue service discipline is used, and will consequently perform identically. The performance impact of the various service disciplines remains to be studied.

The total memory space available on the chip equals MN^2 of which only $1/N$th is effectively

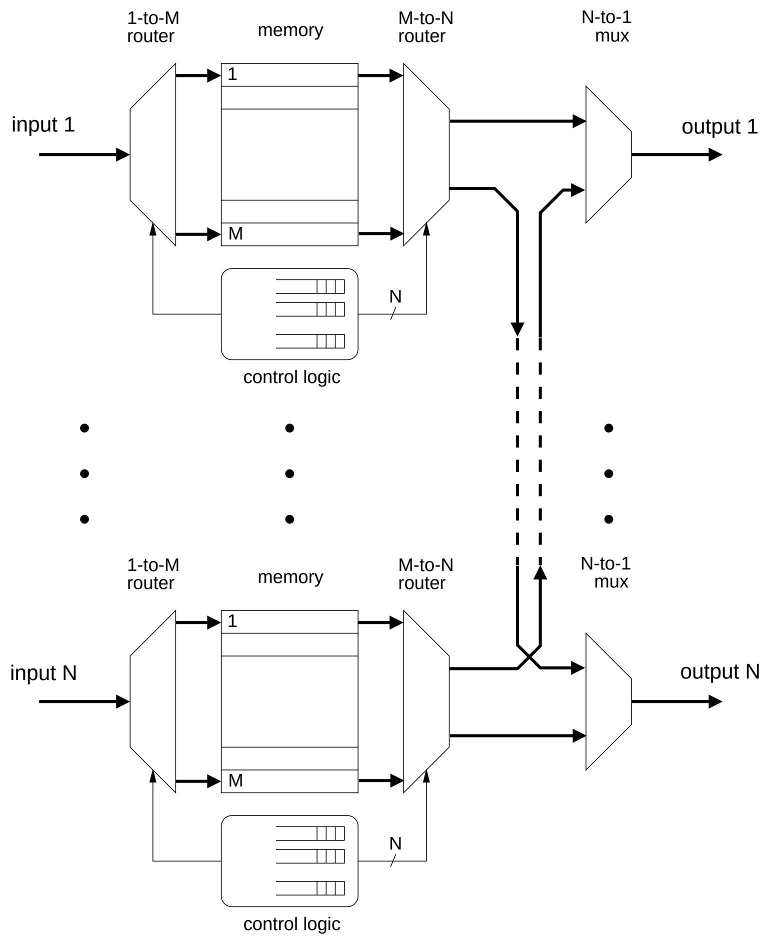


Figure 6.10: Partitioned memory architecture with dedicated buffers at each input.

used. In return, the wiring complexity associated with the fully shared memory is eliminated. Thus, wiring complexity is traded for highly redundant memory space. The advantage is that the memories can be implemented with very dense SRAMs. Because SRAMs are the structure that in terms of integration density scales best with Moore's Law, the buffered crossbar implementation actually uses *less* space with current CMOS technologies. In older CMOS technologies, this is not true, because the integration density of the SRAMs is not high enough.

The drawback clearly is that the amount of memory scales with N^3, but because memory density scales much better than wiring capacity does, it seems likely that this will be the approach of choice for the coming generations of PRIZMA.

The control section can be centralized or distributed. In the latter case, one queue would be associated with each memory (see Fig. 6.11). The read and write bandwidths on this queue both equal just one packet per packet cycle. Again, for performance reasons, flow-control information must be based on the *global* output-queue state.

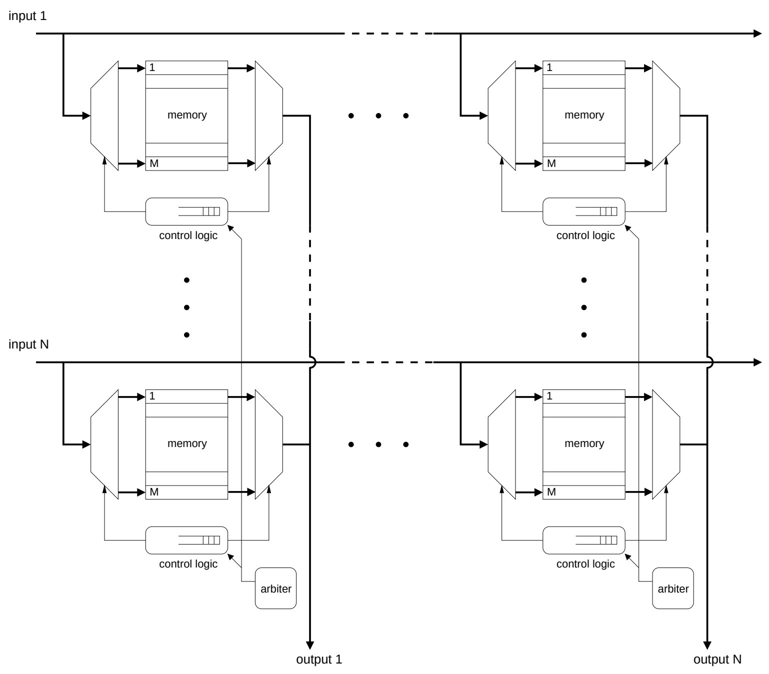


Figure 6.11: Buffered crossbar with distributed control section. Each cycle, every arbiter selects one of the N queues in its column according to some policy (e.g., OCF, RR, LQF).

6.7.7 Conclusion

The main contributions of this dissertation, in terms of the requirements as listed in Section 6.2 are:

- High performance: The main part of this dissertation (see Chapter 3) has contributed an architecture and provided detailed insight into its performance, which has been shown to be high and robust, regardless of switch size, and traffic type and characteristics.
- Multicast support: A scheme for integrated and fair support for multicast traffic in the proposed architecture has been presented in Chapter 4.
- Frame mode: A scheme to support frame mode has been proposed in Chapter 5.

Furthermore, the results obtained here have directly led to a reconsideration of the originally planned implementation, and resulted in both novel architectures and novel implementation alternatives thereof that improve scalability, thus leveraging the PRIZMA architecture to future high-speed packet switches.

Chapter 7

Conclusions and Future Work

The subject of this dissertation has been the design of high-performance packet-switch architectures for high-speed communication networks. The main requirements imposed on such a switch are that link utilization must be as high as possible and packet latency must be kept to a minimum. Therefore, the focus in this work has been on switch *performance*. However, maximum performance does not come for free. For a given number of switch inputs and outputs and a given bandwidth per port, it is equally important that an envisioned switch architecture is implementable in current VLSI technology at a reasonable cost. Furthermore, the ability of an architecture to be expanded to a larger system with either more ports, faster ports, or both, is an important criterion.

As no work of research exists in a vacuum, a large effort has been invested in cataloguing and analyzing existing packet switch architectures and related work. In doing so, we have identified the pros and cons of various existing approaches and gathered a fundamental understanding of the problem at hand. Equipped with this knowledge, we have proceeded to develop an architecture that combines design principles from existing architectures in such a way as to gain the best of both worlds.

7.1 Contributions

This dissertation has made the following contributions to the field of packet-switch design:

- Distributed contention resolution by combining output queuing with virtual output queuing at the input. Thus, the proposed architecture falls into the class of *combined input- and output-buffered* (CIOQ) switches.

- Static output buffer sharing strategies for the proposed architecture and their implications on performance in terms of throughput, delay and loss rates.

- Distributed, fair, and straightforward-to-implement multicasting scheme for the proposed architecture.

- Deadlock-free support of long packets (*frame mode*).

In the course of the work, three patent applications [Colmant98a, Colmant98b, Minkenberg00a] have been submitted, all of which are directly related to the work presented.

The proposed architecture is amenable to implementation in high-speed VLSI. A 1 Tb/s switch fabric based on this architecture is feasible in current technology. Owing to its distributed nature, the architecture can be scaled to support more and/or faster ports.

Within the context of the PRIZMA switch project, this work has helped in gaining a better understanding of how to architect real-world (as opposed to purely theoretical paper designs) packet switches using an output-buffered switch-on-a-chip.

7.1.1 Distributed Contention Resolution

The concept of *distributed contention resolution* by means of a *combined input- and output-queued* packet switch employing *virtual output queuing* at the input side has been introduced. By means of performance simulations, this arrangement has been shown to offer excellent performance in terms of delay–throughput characteristics. Moreover, the architecture behaves robustly with respect to varying traffic characteristics, ranging from benign, uncorrelated traffic to unfavorable, highly bursty traffic.

This arrangement has the following important features:

- The output-queued switch element enables distributed arbitration of the VOQs, i.e., every input can make a decision independent of all other inputs. Therefore, the complexity of VOQ arbitration is linear with the number of ports N. This is a great improvement over the purely input-queued approach that requires a complex centralized scheduler.
- The size of the shared memory must be on the order of N^2. Although this is a quadratic dependency on N, this number generally is much smaller than the size of the shared memory one would typically require in a purely output-buffered switch to achieve a certain loss probability. In our CIOQ approach, the bulk of the queues is pushed back to the input queues, whereas the shared-memory switch acts as an efficient contention resolution device. This is advantageous because the input buffers are low-bandwidth ($2B$), and thus relatively cheap compared to the high-bandwidth ($2NB$) shared-memory buffer.

A patent application for this architecture has been submitted [Colmant98a].

7.1.2 Output-Buffer-Sharing Strategies

Furthermore, new insights into the concepts of shared-memory output queuing have been provided. In contrast to the systems studied in literature so far, the proposed architecture exhibits the following characteristics:

- To achieve *maximum throughput*, the available memory space must be *partitioned* over all outputs, so as to guarantee that no output can interfere with the output-contention resolution process of any of the other outputs.
- To achieve *minimum packet-loss rate*, the available memory must be shared among all outputs at low to medium load levels, whereas at high loads, it must be partitioned.

These findings are instrumental in achieving the best possible performance with the proposed architecture under a wide range of operating conditions, and differ fundamentally from the established strategies in conventional CIOQ systems employing FIFO input queues.

Furthermore, these results suggest that a fixed buffer allocation strategy cannot be optimum under all traffic conditions. Therefore, an opportunity exists to dynamically adjust buffer allocation according to current traffic patterns with the aim of optimizing performance.

7.1.3 Distributed Fair Multicasting Scheme

It is generally agreed that *multicast* traffic will constitute an increasingly significant portion of traffic in future communication networks. Therefore, efficient integration of multicast support is a key requirement for any packet switch design. A novel, distributed multicasting scheme that fits perfectly with the proposed architecture has been proposed. Not only does the proposed scheme offer superior performance to any previously proposed architecture or scheme, it also *integrates* unicast and multicast scheduling in a fair, elegant, and cheap-to-implement way that has not been achieved before. The method and algorithm have been formulated in a patent application [Minkenberg00a].

7.1.4 Deadlock-Free Frame Mode

Although not an integral part of this dissertation, the *deadlock-free frame mode* is a useful extension of the CIOQ switch architecture that enables non-interleaved transmission of long data units ("frames") by suitably modifying flow-control information between shared-memory switch element and input queues. The invention has also been patented [Colmant98b].

7.2 Future Work

This section presents some topics that are of interest in the context of this dissertation and require further research.

The general trend is that the demand for switch capacity continues to grow. Future switches will have more ports, higher data rates per port, and more sophisticated switch functionality (e.g. QoS support). Continued innovation at both the architecture and the implementation level is required to sustain this growth rate.

7.2.1 Adaptive buffer sharing

As indicated above, a fixed buffer allocation strategy cannot be optimum under all conditions. The question naturally arises how to adaptively share the buffer so that optimum resource utilization is achieved under the widest possible range of conditions. One possibility is to adopt a hybrid strategy, where each output queue has dedicated space for N packets, and the remaining $M - N^2$ locations are shared among all outputs. This clearly requires a shared memory size $M > N^2$. Another, more sophisticated approach would be dynamic, adaptive buffer-sharing strategies, possibly by means of adaptive threshold programming.

7.2.2 Real-world traffic patterns

The impact on system behavior and performance of more realistic traffic patterns, such as self-similar traffic models or real Internet traffic traces rather than synthetic traffic sources, remains to be studied.

7.2.3 QoS mechanisms

The integration of Quality-of-Service or Class-of-Service support in the proposed architecture has only been treated in passing here. QoS must be tackled with a system-wide approach, incorporating scheduling, flow control, and congestion control at every queuing point in the system.

7.2.4 Multi-stage fabrics

It is clear that the single-stage approach cannot scale to systems with hundreds or thousands of ports. Such large systems will certainly require a multi-stage approach, but a number of important issues remain to be solved, such as fabric topology, performance, routing, flow and congestion control, and multicast. Can the distributed contention-resolution scheme and VOQ concepts be extended for application in multi-stage switch systems?

7.2.5 Parallel packet switches

To overcome memory bandwidth limitations, parallel packet-switch (PPS) architectures have been proposed, e.g. [Iyer00]. Such a packet switch combines a number, say K, of separate $N \times N$ output-queued switch fabrics with line rate B into one $N \times N$ output-queued switch fabric with line rate KB. This approach is strongly related to speed expansion and link paralleling (see Sections 6.3.1 and 6.6), in that all of these techniques allow line rates to be much faster than the memory. A PPS performs load balancing across ports on different switch chips, thus enabling aggregate system throughput to be grown by a factor K. However, to remain work conserving, a central scheduling algorithm is required and the fabric must be sped up internally by a factor of two to three [Iyer00].

Further research to eliminate the need for speed-up and distribute the scheduling algorithm while maintaining performance is warranted.

7.2.6 Optical and hybrid electro-optical switches

The predominance of optical transmission technology in the Internet backbone has triggered a lot of research in all-optical switches. Such switches, in particular those of the MEMS variety, are now becoming economical. However, these switches have certain important drawbacks compared to electronic switches, so that it is more likely that they will coexist in future networks, with the optical switches handling huge numbers of ports and very high data rates at a rather coarse granularity, and the electronic switches handling fewer ports and lower data rates, but at a much finer granularity. So far, very little research has been done with respect to hybrid electro-optical switches that attempt to combine the strengths of both.

Bibliography

[Aanen92] Aanen, E., J.L. van den Berg, and R.J.F. de Vries, “Cell Loss Performance of the Gauss ATM Switch,” in *Proc. IEEE INFOCOM ’92*, vol. 2, 1992, pp. 717-726.

[Abel00] Abel, F., “Design and Verification of Modern High-Speed Switches,” in *Proc. SAME 2000*, Sophia Antipolis, France, Oct. 2000, pp. 30-38.

[Agrawal97] Agrawal, J.P., S.V. Reddy, R. Halker and F.T. Yap, “An Integrated Approach to Multicasting in ATM Switches,” in *Proc. IEEE Int’l Performance, Computing, and Communications Conf. ’97*, Phoenix AZ, Feb. 1997, pp. 238-244.

[Ahmadi89a] Ahmadi, H. and W.E. Denzel, C.A. Murphy, E. Port, “A High-Performance Switch Fabric for Integrated Circuit and Packet Switching,” *Int. J. Digital and Analog Cabled Syst.*, vol. 2, no. 4, 1989, pp. 277-287.

[Ahmadi89b] Ahmadi, H. and W.E. Denzel, “A Survey of Modern High-Performance Switching Techniques,” *IEEE J. Sel. Areas Commun.*, vol. 7, no. 7, Sep. 1989, pp. 1091-1103.

[Ajmone97] Ajmone Marsan, M.G., A. Bianco and E. Leonardi, “RPA: A Simple, Efficient, and Flexible Policy for Input Buffered ATM Switches,” *IEEE Commun. Letters*, vol. 1, no. 3, May 1997, pp. 83-86.

[Ajmone98] Ajmone Marsan, M., A. Bianco, E. Leonardi and L. Milia, “Quasi-Optimal Algorithms for Input Buffered ATM Switches,” in *Proc. Third IEEE Symposium on Computers and Communications ISCC ’98*, pp. 336-342.

[Ajmone00] Ajmone Marsan, M., E. Leonardi, M. Mellia and F. Neri, “On the Stability of Input-Buffer Cell Switches with Speed-up,” in *Proc. INFOCOM 2000*, Tel Aviv, Israel, Mar. 2000, vol. 3, pp. 1604-1613.

[Alaiwan99] Alaiwan, H., “IBM 8265 ATM Backbone Switch Hardware Architecture,” *Computer Networks*, vol. 31, no. 6, 1999, pp. 527-539.

[Anderson93] Anderson, T.E., S.S. Owicki, J.B. Saxe and C.P. Thacker, “High Speed Switch Scheduling for Local Area Networks,” Digital Research Paper No. 99, Apr. 26, 1993.

[Andersson96] Andersson, P., and C. Svensson, “A VLSI Architecture for an 80 Gb/s ATM Switch Core,” in *Proc. 8th Annual IEEE Int’l Conf. Innovative System in Silicon*, Austin, TX, Oct. 9-11, 1996.

[Andrews99] Andrews, A., S. Khanna and K. Kumaran, “Integrated Scheduling of Unicast and Multicast Traffic in an Input-Queued Switch,” in *Proc. IEEE INFOCOM ’99*, Dec. 1999, pp. 1144-1151.

[Awdeh95] Awdeh, R.Y. and H.T. Mouftah, “Survey of ATM Switch Architectures,” *Computer Networks and ISDN Systems*, vol. 27 (1995), pp. 1567-1613.

[Bakka82] Bakka, R. and M. Dieudonné, “Switching Circuit for Digital Packet Switching Network,” U.S. Patent 4 314 367, Feb. 2, 1982.

[Bianco99] Bianco A., M. Ajmone Marsan, P. Giaccone, E. Leonardi and F. Neri, “Scheduling in Input-Queued Cell-Based Packet Switches,” in *Proc. GLOBECOM ’99*, Rio de Janeiro, Brazil, Dec. 1999.

[Bigo00] Bigo, S., and W. Idler, “Multi-Terabit/s Transmission over Alcatel Teralight Fiber,” *Alcatel Telecommunications Review*, 4th Quarter 2000, pp. 288-296.

[Bishop01] Bishop, D.J., C.R. Giles, and S.R. Das, “The Rise of Optical Switching,” Scientific American, Jan. 2001, pp. 74-79.

[Cao95] Cao, X.-R., “The Maximum Throughput of a Nonblocking Space-Division Packet Switch with Correlated Destinations,” *IEEE Trans. Commun.*, vol. 43, no. 5, May 1995, pp. 1898-1901.

[Chan97] Chan, S., E.W.M. Wong and K.T. Ko, “Fair Packet Discarding for Controlling ABR Traffic in ATM Networks,” *IEEE Trans. Commun.*, vol. 45, no. 8, Aug. 1997, pp. 913-916.

[Chang94] Chang, C.-Y., A.J. Paulraj and T. Kailath, “A Broadband Packet Switch Architecture with Input and Output Queueing,” in *Proc. IEEE GLOBECOM ’94*, pp. 448-452.

[Chang00] Chang, C.-S., W.-J. Chen and H.-Y. Huang, “Birkhoff-von Neumann Input Buffered Crossbar Switches,” in *Proc. INFOCOM 2000*, Tel Aviv, Israel, Mar. 2000, vol. 3, pp. 1614-1623.

[Chao91] Chao, H.J., “A Recursive Modular Terabit/Second ATM Switch,” *IEEE J. Sel. Areas Commun.*, vol. 9, no. 8, Oct. 1991, pp. 1161-1172.

[Charny98a] Charny, A., P. Krishna, N. Patel and R.J. Simcoe, “Algorithms for Providing Bandwidth and Delay Guarantees in Input-Buffered Crossbar Switches with Speedup,” in *Proc. IWQoS ’98*, Napa Valley CA, May 1998, pp. 225-234.

[Charny98b] Charny, A., “Providing QoS Guarantees in Input-Buffered Crossbar Switches with Speedup,” Ph.D. Thesis, MIT, 1998.

[Chen91] Chen, J.S.-C. and R. Guerin, “Performance Study of an Input Queueing Packet Switch with Two Priority Classes,” *IEEE Trans. Commun.*, vol. 39, no. 1, Jan. 1991, pp. 117-126.

[Chen92] Chen, X., and J.F. Hayes, “Call scheduling in multicasting packet switching,” in *Proc. IEEE ICC ’92*, pp. 895-899.

[Chen94] Chen, X., J.F. Hayes and M.K. Mehmet-Ali, “Performance Comparison of two Input Access Methods for a Multicast Switch,” *IEEE Trans. Commun.*, vol. 42, no. 5, May 1994, pp. 2174-2177.

[Chen00a] Chen, W.-T., C.-F. Huang, Y.-L. Chang, and W.-Y. Hwang, “An Efficient Cell-Scheduling Algorithm for Multicast ATM Switching Systems,” *IEEE/ACM Trans. Networking*, vol. 8, no. 4, Aug. 2000, pp. 517-525.

[Chen00b] Chen, Y. and J.S. Turner, “WDM Burst Switching for Petabit Capacity Routers,” in *Proc. IEEE MILCOM 1999*, vol. 2, 1999, pp. 968-973.

[Chiussi97] Chiussi, F.M., Y. Xia and V.P. Kumar, “Performance of Shared-Memory Switches under Multicast Bursty Traffic,” *IEEE J. Sel. Areas Commun.*, vol. 15, no. 3, Apr. 1997, pp. 472-487.

[Choudhury98] Choudhury, A.K. and E.L. Hahne, “Dynamic Queue Length Thresholds for Shared-Memory Packet Switches,” *IEEE/ACM Trans. Networking*, vol. 6, 1998, pp. 130-140.

[Chuang98] Chuang, S-T., A. Goel, N. McKeown and B. Prabhakar, “Matching Output Queueing with a Combined Input Output Queued Switch,” Stanford CSL-TR-98-758.

[Chuang99] Chuang, S.-T., A. Goel, N. McKeown and B. Prabhakar, “Matching Output Queueing with a Combined Input Output Queued Switch,” *IEEE J. Sel. Areas Commun.*, vol. 17, no. 6, Jun. 1999, pp. 1030-1039.

[Cohen90] Cohen, E., *Programming in the 1990s*, Springer-Verlag, New York, 1990.

[Colmant98a] Colmant, M., A.P.J. Engbersen, F. Gramsamer and C.J.A. Minkenberg, “Combination of Virtual Output Queuing and Output Buffering to Achieve Robust Switch Performance,” pending patent application CH 8-1999-0013, 1998.

[Colmant98b] Colmant, M., F. Gramsamer and C.J.A. Minkenberg, “Variable Length Packet Switching over a Fixed Length Cell Shared Buffer Switch,” pending patent application CH 8-1998-0088, 1998.

[Colmant98c] Colmant, M., and R. Luijten, “A Single-Chip, Lossless 16×16 Switch Fabric with 28 Gb/s Throughput,” *IBM Research Report*, RZ 3087, Dec. 1998.

[Daddis89] Daddis, G.E. and H.C. Torng, “A Taxonomy of Broadband Integrated Switching Architectures,” *IEEE Commun. Mag.*, May 1989, pp. 32-42.

[Dai00] Dai, J.G. and B. Prabhakar, "The Throughput of Data Switches with and without Speedup," in *Proc. INFOCOM 2000*, Tel Aviv, Israel, Mar. 2000, vol. 2, pp. 556-564.

[DelRe93] Del Re, E. and R. Fantacci, "Performance Evaluation of Input and Output Queueing Techniques in ATM Switching Systems," *IEEE Trans. Commun.*, vol. 41, no. 10, Oct. 1993.

[Denzel92] Denzel, W.E., A.P.J. Engbersen, I. Iliadis and G. Karlsson, "Highly Modular Packet Switch for Gb/s Rates," in *Proc. XIV International Switching Symposium*, Yokohama, Japan, Oct. 25-30, 1992, pp. 237-240.

[Denzel95] Denzel, W.E., A.P.J. Engbersen and I. Iliadis, "A Flexible Shared-Buffer Switch for ATM at Gb/s Rates," *Computer Networks and ISDN Systems*, vol. 27, no. 4, Jan. 1995, pp. 611-624.

[Devault88] Devault, M., J.Y. Cochennec and M. Servel, "The Prelude ATD Experiment: Assessments and Future Prospects," *IEEE J. Sel. Areas Commun.*, vol. 6, no. 9, Dec. 1988, pp. 1528-1537.

[DeVries90a] De Vries, R.J.F, "Gauss: A Single-Stage ATM Switch with Output Buffering," in *Proc. Int. Conf. Broadband Services and Networks*, 1990, pp. 248-252.

[DeVries90b] De Vries, R.J.F, "ATM Multicast Connections using the Gauss Switch," in *Proc. IEEE GLOBECOM '90*, vol. 1, 1990, pp. 211-217.

[Dias84] Dias, D.M. and M. Kumar, "Packet Switching in $N \log N$ Multistage Networks," in *Proc. GLOBECOM '84*, Atlanta, GA, Dec. 1984, pp. 114-120.

[Dinic70] Dinic, E.A., "Algorithm for a Solution of a Problem of Maximum Flow in a Network with Power Estimation," *Soviet Math. Dokl.*, vol. 11, pp. 1277-1280, 1970.

[Duan97] Duan, H., J.W. Lockwood, S.M. Kang and J.D. Will, "A High-Performance OC12/OC48 Queue Design Prototype for Input Buffered ATM Switches," in *Proc. IEEE INFOCOM '97*, Kobe, Japan, Apr. 1997, vol. 1, pp. 20-28.

[Duan98] Duan, H., J.W. Lockwood, and S.M. Kang, "Matrix Unit Cell Scheduler (MUCS) for Input-Buffered ATM Switches," *IEEE Commun. Letters*, vol. 2, no. 1, Jan. 1998, pp. 20-23.

[Eckberg88] Eckberg, A.E. and T.-C. Hou, "Effect of Output Buffer Sharing on Buffer Requirements in an ATDM Packet Switch," in *Proc. INFOCOM '88*, New Orleans, LA, Mar. 1988, pp. 459-466.

[Eng88a] Eng, K.Y., M.G. Hluchyj and Y.S. Yeh, "Multicast and Broadcast Services in a Knockout Packet Switch," in *Proc. IEEE INFOCOM '88*, New Orleans LA, Mar. 1988, pp. 29-34.

[Eng88b] Eng, K.Y., "A Photonic Knockout Switch for High-Speed Packet Networks," *IEEE J. Sel. Areas Commun.*, vol. 6, no. 7, Aug. 1988, pp. 1107-1116.

[Eng89] Eng, K.Y., M.J. Karol and Y.S. Yeh “A Growable Packet (ATM) Switch Architecture: Design Principles and Applications,” in *Proc. IEEE GLOBECOM ’89*, pp. 1159-1165.

[Engbersen92] Engbersen, A.P.J., “Multicast/Broadcast Mechanism for a Shared Buffer Packet Switch,” *IBM Technical Disclosure Bulletin*, vol. 34, no. 10a, Mar. 1992, pp. 464 -465.

[Fantacci96] Fantacci, R., M. Forti and M. Marini, “A Cellular Neural Network for Packet Selection in a Fast Packet Switching Fabric with Input Buffers,” *IEEE Trans. Commun.*, vol. 44, no. 12, Dec. 1996, pp. 1649-1652.

[Gale62] Gale, D. and L.S. Shapley, “College Admissions and the Stability of Marriage,” *American Mathematical Monthly*, vol. 69, 1962, pp. 9-15.

[Ghosh91] Ghosh, D. and J.C. Daly, “Delta Networks with Multiple Links and Shared Output Buffers: A High Performance Architecture for Packet Switching,” in *Proc. IEEE GLOBECOM ’91*, Phoenix, AZ, Dec. 1991, pp. 949-953.

[Giacopelli91] Giacopelli, J., J. Hickey, W. Marcus, D. Sincoskie and M. Littlewood, “Sunshine: A High-performance Self-routing Broadband Packet Switch Architecture,” *IEEE J. Sel. Areas Commun.*, vol. 9, no. 8, Oct. 1991, pp. 1289-1298.

[Gomes96] Gomes, R.C. de Souza, “A Simulation Study of an Input Buffered ATM Switch Based on the Request-Grant-Status Iterative Scheduling Algorithm,” Master Thesis, University of Kansas, Dec. 1996.

[Goudreau00] Goudreau, M.W., S.G. Kolliopoulos and S.B. Rao, “Scheduling Algorithms for Input-Queued Switches: Randomized Techniques and Experimental Evaluation,” in *Proc. INFOCOM 2000*, Tel Aviv, Israel, Mar. 2000, vol. 3, pp. 1634-1643.

[Guerin97] Guérin, R.A. and K.N. Sivarajan, “Delay and Throughput Performance of Speeded-Up Input-Queueing Packet Switches,” IBM Internal Research Report, RC 20892 (92583), Jun. 24, 1997.

[Guo98] Guo, M.H. and R.S. Chang, “Multicast ATM Switches: Survey and Performance Evaluation,” *Computer Commun. Review*, vol. 28, no. 2, 1998, pp. 98-131.

[Gupta91a] Gupta, A.K. and N.D. Georganas, “Analysis of a Packet Switch with Input and Output Buffers and Speed Constraints,” in *Proc. IEEE INFOCOM ’91*, Bal Harbour, FL, Apr. 1991, pp. 694-700.

[Gupta91b] Gupta, A.K., L.O. Barbosa and N.D. Georganas, “A 16 $\times$ 16 Limited Intermediate Buffer Switch Module for ATM Networks,” in *Proc. IEEE GLOBECOM ’91*, Phoenix, AZ, Dec. 1991, pp. 939-943.

[Gupta98] Gupta, P. and N. McKeown, “Design and Implementation of a Fast Crossbar Scheduler,” in *Proc. Hot Interconnects VI*, Stanford University, Aug. 1998.

[Gupta99] Gupta, P. and N. McKeown, "Designing and Implementating a Fast Crossbar Scheduler," *IEEE Micro Magazine*, vol. 19, no. 1, Jan.-Feb. 1999, pp. 20-28.

[Gusfield89] Gusfield, D. and R.W. Irving, *The Stable Marriage Problem: Structure and Algorithms*, MIT Press, 1989.

[Hahne91] Hahne, E.L., "Round-Robin Scheduling for Max-Min Fairness in Data Networks", *IEEE J. Sel. Areas Commun.*, vol. 9, no. 7, Sep. 1991, pp. 1024-1039.

[Haung96] Haung, Y.R. and M.C. Yuang, "A High-Performance Input Access Scheme for ATM Multicast Switching," in *Proc. IEEE ICC '96*, Dallas, TX, Jun. 1996, pp. 1035-1039.

[Hayes89] Hayes, J.F., M.K. Mehmet-Ali, R. Breault and J.L. Houle, "Performance of a Multicast Switch for Broadband ISDN," in *Proc. IEEE Pacific Rim Conf. Communications, Computers and Signal Processing*, Jun. 1989, pp. 427-430.

[Hayes91] Hayes, J.F., R. Breault and M.K. Mehmet-Ali, "Performance Analysis of a Multicast Switch," *IEEE Trans. Commun.*, vol. 39, no. 4, Apr. 1991, pp. 581-587.

[Herkersdorf97] Herkersdorf, A. and W. Lemppenau, "A Scaleable SONET/SDH ATM Transceiver with Integrated Add/Drop and Cross-connect," IBM Research Report RZ 2911, 1997.

[Hluchyj88a] Hluchyj, M.G. and M.J. Karol, "Queueing in Space-Division Packet Switching," in *Proc. IEEE INFOCOM '88*, Mar. 1988, pp. 334-343.

[Hluchyj88b] Hluchyj, M.G. and M.J. Karol, "Queueing in High-Performance Packet Switching," *IEEE J. Sel. Areas Commun.*, vol. 6, no. 9, Dec. 1988, pp. 1587-1597.

[Hopcroft73] Hopcroft, J.E. and R.M. Karp, "An $n^{5/2}$ Algorithm for Maximum Matching in Bipartite Graphs," *Soc. Ind. Appl. Math. J. Computation*, vol. 2, no. 4, Dec. 1973, pp. 225-231.

[Huang84] Huang, A. and S. Knauer, "Starlite: A Wideband Digital Switch," in *Proc. GLOBECOM '84*, Atlanta, GA, Dec. 1984, pp.121-125.

[Hui87] Hui, J.Y. and E. Arthurs, "A Broadband Packet Switch for Integrated Transport," *IEEE J. Sel. Areas Commun.*, vol. 5, no. 8, Oct. 1987, pp. 1264-1273.

[Hui90] Hui, J.Y. and T. Renner, "Queueing Strategies for Multicast Packet Switching," in *Proc. IEEE GLOBECOM '90*, San Diego, CA, 1990, pp. 1431-1437.

[Iliadis90a] Iliadis, I. and W.E. Denzel, "Performance of Packet Switches with Input and Output Queueing," in *Proc. ICC '90*, Apr. 1990, pp. 747-753.

[Iliadis90b] Iliadis, I., “Head of the Line Arbitration of Packet Switches with Input and Output Queueing,” in *Proc. IFIP TC6 Fourth International Conf. Data Communication Systems and their Performance*, Barcelona, Spain, 1990, pp. 129-142.

[Iliadis91a] Iliadis, I., “Head of the Line Arbitration of Packet Switches with Combined Input and Output Queueing,” *Int. J. Digital and Analog Commun. Syst.*, vol. 4, 1991, pp. 181-190.

[Iliadis91b] Iliadis, I., “Performance of a Packet Switch with Shared Buffer and Input Queueing,” in *Proc. Teletraffic and Datatraffic in a Period of Change, ITC-13*, 1991, pp. 911-916.

[Iliadis92a] Iliadis, I., “Synchronous versus Asynchronous Operation of a Packet Switch with Combined Input and Output Queuing,” *Performance Evaluation 16*, 1992, pp. 241-250.

[Iliadis92b] Iliadis, I., “Performance of a Packet Switch with Input and Output Queuing under Unbalanced Traffic,” in *Proc. IEEE INFOCOM ’92*, May 1992, pp. 743-752.

[Iliadis93] Iliadis, I. and W.E. Denzel, “Analysis of Packet Switches with Input and Output Queuing,” *IEEE Trans. Commun.*, vol. 41, no. 5, May 1993, pp. 731-740.

[Iyer00] Iyer, S., A. Awadallah, and N. McKeown, “Analysis of a Packet Switch with Memories Running Slower than the Line-Rate,” in *Proc. IEEE INFOCOM 2000*, vol. 2, 2000, pp. 529-537.

[Jung95] Jung, Y.C. and C.K. Un, “Banyan Multipath Self-Routing ATM Switches with Shared Buffer Type Switch Elements,” *IEEE Trans. Commun.*, vol. 43, no. 11, Nov. 1995, pp. 2847-2857.

[Jurczyk96] Jurczyk, M., and T. Schwederski, “Phenomenon of Higher Order Head-of-Line Blocking in Multistage Interconnection Networks under Non-uniform Traffic Patterns,” *IEICE Trans. Information and Systems*, vol. E79-D, no. 8, Aug. 1996, pp. 1124-1129.

[Karol87] Karol, M.J., M.G. Hluchyj and S.P. Morgan, “Input vsȮutput Queueing on a Space-division Packet Switch,” *IEEE Trans. Commun.*, vol. 35, no. 12, 1987, pp. 1347-1356.

[Karol92] Karol, M.J., K.Y. Eng and H. Obara, “Improving the Performance of Input-queued ATM Packet Switches,” in *Proc. IEEE INFOCOM ’92,* Florence, vol. 1, May 1992, pp. 110-115.

[Karp90] Karp, R.M., U.V. Vazirani and V.V. Vazirani, “An Optimal Algorithm for On-line Bipartite Matching,” in *Proc. 22nd Annual ACM Symposium on the Theory of Computing*, May 1990, pp. 352-358.

[Katevenis91] Katevenis, M., S. Sidiropoulos, and C. Courcoubetis, “Weighted Round-Robin Cell Multiplexing in a General-Purpose ATM Switch Chip,” *IEEE J. Sel. Areas Commun.*, vol. 9, no. 8, Oct. 1991, pp. 1265-1279.

[Katevenis95] Katevenis, M., P. Vatsolaki and A. Efthymiou, “Pipelined Memory Shared Buffer for VLSI Switches,” in *Proc. ACM SIGCOMM ’95*, Aug. 1995, pp. 39-48.

[Katevenis96a] Katevenis, M., D. Serpanos and P. Vatsolaki, “ATLAS I: A General-Purpose, Single-Chip ATM Switch with Credit-Based Flow Control,” in *Proc. IEEE Hot Interconnects IV Symposium*, Stanford, CA, Aug. 15-17, 1996.

[Katevenis96b] Katevenis, M., D. Serpanos and E. Spyridakis, “Credit-Flow-Controlled ATM versus Wormhole Routing,” Technical report TR-171, FORTH-ICS, Heraklion, Crete, Jul. 1996.

[Katevenis97] Katevenis, M., D. Serpanos and E. Spyridakis, “Switching Fabrics with Internal Backpressure using the ATLAS I Single-Chip ATM Switch,” in *Proc. GLOBECOM ’97*, Phoenix, AZ, Nov. 1997.

[Katevenis98] Katevenis, M., D.N. Serpanos and G. Dimitriadis, “ATLAS I: a single-chip, gigabit ATM switch with HIC/HS links and multi-lane back-pressure,” *Elsevier Microprocessors and Microsystems*, vol. 21, 1998, pp. 481-490.

[Kato88] Kato, et al., “Experimental Broadband ATM Switching System,” in *Proc. GLOBECOM ’88*, Hollywood, FL, Nov. 1988, pp. 1288-1292.

[Kim90] Kim, H.S and A. Leon-Garcia, “A Self-Routing Multistage Switching Network for Broadband ISDN,” *IEEE J. Sel. Areas Commun.*, vol. 8, no. 3, Apr. 1990, pp.459-466.

[Kleinrock75] Kleinrock, L., *Queueing Systems*, Wiley and Sons, New York, 1975.

[Kornaros99] Kornaros, G., D. Pnevmatikatos, P. Vatsolaki, G. Kalokerinos, C. Xanthaki, D. Mavroidis, D. Serpanos and M. Katevenis, “ATLAS I: Implementing a Single-Chip ATM Switch with Backpressure,” *IEEE Micro Magazine*, Jan.-Feb. 1999, pp. 30-41.

[Kozaki91] Kozaki, T., N. Endo, Y. Sakurai, O. Matsubara, M. Mizukami and K. Asano, “32 $\times$ 32 Shared Buffer Type ATM Switch VLSI’s for B-ISDN’s,” *IEEE J. Sel. Areas Commun.*, vol. 9, no. 8, Oct. 1991, pp. 1239-1247.

[Krishna98] Krishna, P., N.S. Patel, A. Charny and R.J. Simcoe, “On the Speedup Required for Work-Conserving Crossbar Switches,” in *Proc. 6th IEEE/IFIP IWQoS ’98*, Napa Valley, CA, May 1998, pp. 225-234.

[Krishna99] Krishna, P., N.S. Patel, A. Charny and R.J. Simcoe, “On the Speedup Required for Work-Conserving Crossbar Switches,” *IEEE J. Sel. Areas Commun.*, vol. 17, no. 6, 1999, pp. 1057-1066.

[Kumar95] Kumar, P.R. and S.P. Meyn, "Stability of Queueing Networks and Scheduling Policies," *IEEE Trans. Automatic Control*, vol. 40, no. 2, Feb. 1995, pp. 251-260.

[Kuwahara89] Kuwahara, H., N. Endo, M. Ogino and T. Kozaki, "A Shared Buffer Memory Switch for an ATM Exchange," *Proc. ICC '89*, Boston MA, Jun. 1989, pp. 118-122.

[LaMaire94] LaMaire, R.O. and D.N. Serpanos, "Two-Dimensional Round-Robin Schedulers for Packet Switches with Multiple Input Queues," *IEEE/ACM Trans. Networking*, vol. 2, no. 5, Oct. 1994, pp. 471-482.

[Lea93] Lea, C.-T., "A Multicast Broadband Packet Switch," *IEEE Trans. Commun.*, vol. 41, no. 4, Apr. 1993, pp. 621-630.

[Lee88a] Lee, T.T., R. Boorstyn and E. Arthurs, "The Architecture of a Multicast Broadband Packet Switch," in *Proc. IEEE INFOCOM '88*, New Orleans, LA, Mar. 1988, pp. 1-8.

[Lee88b] Lee, T.T, "Nonblocking Copy Networks for Multicast Packet Switching," *IEEE J. Sel. Areas Commun.*, vol. 6, no. 9, Dec. 1988, pp. 1455-1467.

[Lee95] Lee, M.J. and D.S. Ahn, "Cell Loss Analysis and Design Trade-Offs of Nonblocking ATM Switches with Nonuniform Traffic," *IEEE/ACM Trans. Netw.*, vol. 3, no. 2, Apr. 1995, pp. 199-210.

[Leland94] Leland, W.E., M.S. Taqqu, W. Willinger and D.V. Wilson "On the Self-Similar nature of Ethernet Traffic (Extended Version)," *IEEE/ACM Trans. Networking*, vol. 2, no. 1, Feb. 1994, pp. 1-15

[Li90] Li, S.-Q., "Nonuniform Traffic Analysis on a Nonblocking Space-Division Packet Switch," *IEEE Trans. Commun.*, vol. 38, Jul. 1990, pp. 21-31.

[Li92] Li, S.-Q., "Performance of a Nonblocking Space-Division Packet Switch with Correlated Input Traffic," *IEEE Trans. Commun.*, vol. 40, no. 1, Jan. 1992, pp. 97-108.

[Liew94] Liew, S.C., "Performance of Various Input-buffered and Output-buffered ATM Switch Design Principles under Bursty Traffic: Simulation Study," *IEEE Trans. Commun.*, vol. 42 no. 2/3/4, Feb/Mar/Apr 1994, pp. 1371-1379.

[Liu97] Liu, X. and H.T. Mouftah, "Queueing Performance of Copy Networks With Dynamic Cell Splitting for Multicast ATM Switching," *IEEE Trans. Commun.*, vol. 45, no. 4, Apr. 1997, pp. 464-472.

[Luijten98] Luijten, R.P., "An OC-12 ATM Switch Adapter Chipset," in *Proc. IEEE ATM Workshop: Meeting the Challenges of Deploying the Global Broadband Network Infrastructure*, IEEE, NY, 1998, pp. 26-33.

[Luijten01] Luijten, R.P., T. Engbersen, and C. Minkenberg, "Shared Memory Switching + Virtual Output Queuing: A Robust and Scalable Switch," in *Proc. ISCAS 2001*, Sydney, Australia, May 6-9, 2001.

[Lund96] Lund, C., S. Phillips and N. Reingold, “Fair Prioritized Scheduling in an Input-Buffered Switch,” in *Proc. Broadband Commun.*, Montreal, Canada, 1996, pp. 358-369.

[Matsunaga91] Matsunaga, M. and H. Uematsa, “A 1.5 Gb/s 8×8 Cross-Connect Switch Using a Time Reservation Algorithm,” *IEEE J. Sel. Areas Commun.*, vol. 9, no. 8, Oct. 1991, pp. 1308-1317.

[McKeown93] McKeown, N., P. Varaiya and J. Walrand, “Scheduling Cells in an Input-queued Switch,” *Electron. Lett.*, vol. 29, no. 25, Dec. 1993, pp. 2174-2175.

[McKeown95] McKeown, N.W., “Scheduling Algorithms for Input-Queued Switches,” Ph.D. Thesis, University of California at Berkeley, 1995.

[McKeown96a] McKeown, N., V. Anantharam and J. Walrand, “Achieving 100% Throughput in an Input-Queued Switch,” in *Proc. INFOCOM ’96*, San Francisco, CA, Mar. 1996, vol. 1, pp. 296-302.

[McKeown96b] McKeown, N. and B. Prabhakar, “Scheduling Multicast Cells in an Input-Queued Switch,” in *Proc. INFOCOM ’96*, San Francisco, CA, Mar. 1996, vol. 1, pp. 271-278.

[McKeown98] McKeown, N. and T.E. Anderson, “A Quantitative Comparison of Scheduling Algorithms for Input-Queued Switches,” Computer Networks and ISDN Systems, vol. 30, no. 24, Dec. 1998, pp. 2309-2326.

[McKeown97a] McKeown, N., M. Izzard, A. Mekkittikul, W. Ellersick and M. Horowitz, “The Tiny Tera: A Packet Switch Core,” *IEEE Micro Magazine*, Jan.-Feb. 1997, pp. 26-33.

[McKeown97b] McKeown, N., B. Prabhakar and M. Zhu, “Matching Output Queueing with Combined Input and Output Queueing,” in *Proc. 35th Annual Allerton Conf. Communication, Control and Computing*, Monticello, IL, Oct. 1997.

[McKeown99a] McKeown, N., “The iSLIP Scheduling Algorithm for Input-Queued Switches,” *IEEE/ACM Trans. Networking*, vol. 7, no. 2, Apr. 1999, pp. 188-201.

[McKeown99b] McKeown, N., “Fast Switched Backplane for a Gigabit Switched Router,” Cisco Systems white paper, http://www.cisco.com/warp/public/cc/cisco/mkt/core/12000/tech/fasts_wp.pdf.

[McKeown99c] McKeown, N., A. Mekkittikul, V. Anantharam and J. Walrand, “Achieving 100% Throughput in an Input-Queued Switch,” *IEEE Trans. Commun.*, vol. 47, no. 8, Aug. 1998, pp. 1260-1267.

[McMillen84] McMillen, R.J., “Packet Switched Multiple Queue $N \times M$ Switch Node and Processing Method,” US Patent no. 4,623,996, filed Oct. 18, 1984, published Nov. 18, 1986.

[Mehmet89] Mehmet-Ali, M. and H. Tri Nguyen, “A Neural Network Implementation of an Input Access Scheme in a High-Speed Packet Switch,” in *Proc. IEEE GLOBECOM ’89*, vol. 2, pp. 1192-1196.

[Mehmet91] Mehmet-Ali, M.K. and M. Youssefi, “The Performance Analysis of an Input Access Scheme in a High-Speed Packet Switch,” in *Proc. IEEE INFOCOM ’91*, Bal Harbour, FL, Apr. 1991, pp. 454-461.

[Mehmet96] Mehmet-Ali, M.K. and Shaying Yang, “Performance Analysis of a Random Packet Selection Policy for Multicast Switching,” *IEEE Trans. Commun.*, vol. 44, no. 3, Mar. 1996, pp. 388-398.

[Mekkittikul96] Mekkittikul, A. and N. McKeown, “A Starvation-free Algorithm for Achieving 100% Throughput in an Input-Queued Switch,” in *Proc. ICCCN ’96*, Rockville, MD, Oct. 1996 pp. 226-231.

[Mekkittikul98] Mekkittikul, A. and N. McKeown, “A Practical Scheduling Algorithm to Achieve 100% Throughput in Input-Queued Switches,” in *Proc. IEEE INFOCOM ’98*, San Francisco, CA, Apr. 1998, pp. 792-799.

[Minkenberg96] Minkenberg, C.J.A., “Performance Simulations Of The PrizmaPlus Switch,” Master Thesis, Eindhoven University of Technology, The Netherlands, Oct. 1996.

[Minkenberg00a] Minkenberg, C.J.A., “An Integrated Method for Unicast and Multicast Scheduling in a Combined Input- and Output-Buffered Packet Switching System,” pending patent application CH 8-1999-0098, 1999.

[Minkenberg00b] Minkenberg, C., T. Engbersen and M. Colmant, “A Robust Switch Architecture for Bursty Traffic,” in *Proc. Int. Zurich Seminar on Broadband Commun. IZS 2000*, Zurich, Switzerland, Feb. 2000, pp. 207-214.

[Minkenberg00c] Minkenberg, C., “Integrating Unicast and Multicast Traffic Scheduling in a Combined Input- and Output-Queued Packet-Switching System,” in *Proc. ICCCN 2000*, Las Vegas, NV, Oct. 2000, pp. 127-134.

[Minkenberg00d] Minkenberg, C., and T. Engbersen, “A Combined Input- and Output-Queued Packet-Switch System Based on PRIZMA Switch-on-a-Chip Technology,” *IEEE Commun. Mag.*, vol. 38, no. 12, Dec. 2000, pp. 70-77.

[Nojima87] Nojima, S., E. Tsutsui, H. Fukuda and M. Hashimoto, “Integrated Services Packet Network Using Bus Matrix Switch,” *IEEE J. Sel. Areas Commun.*, vol. SAC-5, pp. 1284-1292, Oct. 1987.

[Nong99] Nong, G., J.K. Muppala and M. Hamdi, “Analysis of Nonblocking ATM Switches with Multiple Input Queues,” *IEEE/ACM Trans. on Networking*, vol. 7, no. 1, Feb. 1999, pp. 60-74.

[Obara89] Obara, H. and T. Yasushi, “An efficient contention resolution algorithm for input queueing ATM cross-connect switches,” *Int. J. Digital and Analog Cabled Syst.*, vol. 2, Dec. 1989, pp. 261-267.

[Obara91] Obara, H., “Optimum architecture for input queueing ATM switches,” *IEE Electron. Lett.*, 28th Mar. 1991, pp. 555-557.

[Obara92a] Obara, H., S. Okamoto and Y. Hamazumi, “Input and Output Queueing ATM Switch Architecture with Spatial and Temporal Slot Reservation Control,” *IEE Electron. Lett.*, Jan. 1992, pp. 22-24.

[Obara92b] Obara, H. and Y. Hamazumi, “Parallel Contention Resolution Control for Input Queueing ATM Switches,” *IEE Electron. Lett.*, vol. 28, no. 9, Apr. 1992, pp. 838-839.

[Oie89] Oie, Y., M.Murata, K. Kubota and H. Miyahara, “Effect of Speedup in Non-blocking Packet Switch,” in *Proc. ICC ’89*, Jun. 1989, pp. 410-414.

[Oie90a] Oie, Y., T. Suda, M. Murata and H. Miyahara, “Survey of the Performance of Nonblocking Switches with FIFO Input Buffers,” *Proc. ICC ’90*, Apr. 1990, pp. 133-167.

[Oie90b] Oie, Y., T. Suda, D. Kolson and H. Miyahara, “Survey of Switching Techniques in High-Speed Networks and Their Performance,” in *Proc. INFOCOM ’90*, Jun. 1990, pp. 1242-1251.

[Park88] Park, S.K. and K.W. Miller, “Random Number Generators: Good ones are hard to find,” *Commun. of the ACM*, vol. 31, no. 10, Oct. 1988, pp. 1192-1201.

[Pao98] Pao, D.C.W. and S.P. Lam, “Cell Scheduling for ATM Switch with Two Priority Classes,” in *Proc. IEEE ATM Workshop 1998*, Fairfax.

[Patel81] Patel, J.K., “Performance of Processor-Memory Interconnections for Multiprocessors,” *IEEE Trans. Computing*, vol. 30, no. 10, Oct. 1981, pp. 771-780.

[Pattavina88] Pattavina, A., “Multichannel Bandwidth Allocation in a Broadband Packet Switch,” *IEEE J. Sel. Areas Commun.*, vol. 6, no. 9, Dec. 1988, pp. 1489-1499.

[Pattavina93] Pattavina, A., “Nonblocking Architectures for ATM Switching,” *IEEE Commun. Mag.*, Feb. 1993, pp. 38-48.

[Phillips99] Phillips, G., S. Shenker and H. Tangmunarunkit, “Scaling of Multicast Trees: Comments on the Chuang-Sirbu Scaling Law,” in *Proc. SIGCOMM ’99*, pp. 41-51.

[Prabhakar95] Prabhakar, B. and N. McKeown, “Designing a Multicast Switch Scheduler,” in *Proc. 33rd Annual Allerton Conference*, Urbana-Champaign, IL, 1995.

[Prabhakar96] Prabhakar, B., N. McKeown and J. Mairesse, “Tetris Models for Multicast Switches,” in *Proc. 30th Annual Conference on Information Sciences and Systems*, Princeton, NJ, 1996.

[Prabhakar97] Prabhakar, B., N. McKeown and R. Ahuja, “Multicast Scheduling for Input-Queued Switches,” *IEEE J. Sel. Areas Commun.*, vol. 15, no. 5, Jun. 1997, pp. 855-866.

[Prabhakar99] Prabhakar, B., and N. McKeown, “On the Speedup Required for Combined Input and Output Queued Switching,” *Automatica*, vol. 35, 1999.

[PRIZMA-E] *PRIZMA-E Homepage*, http://www.lagaude.ibm.com/prizma

[Putten97] Putten, P.H.A. v.d. and J.P.M. Voeten, “Specification of Reactive Hardware/Software Systems (The Method Software/Hardware Engineering (SHE)),” Ph.D. Thesis, University of Technology at Eindhoven, The Netherlands, 1997.

[Rathgeb88] Rathgeb, E.P., T.H. Theimer, M.N. Huber, “Buffering Concepts for ATM Switching Networks,” in *Proc. GLOBECOM ’88*, Hollywood, FL, Nov. 1988, pp. 1277-1281.

[Rodeheffer98] Rodeheffer, T.L. and J.B. Saxe, “An Efficient Matching Algorithm for a High-Throughput, Low-Latency Data Switch,” Compaq/Digital SRC Research Report no. 162, Nov. 5, 1998.

[Schultz94] Schultz, K.J. and P.G. Gulak, “Distributed multicast contention resolution using content addressable FIFOs,” in *Proc. IEEE ICC ’94*, New Orleans, LA, pp. 1495-1500.

[Schultz96] Schultz, K.J. and P.G. Gulak, “Multicast contention resolution with single-cycle windowing using content addressable FIFOs,” *IEEE/ACM Trans. Networking*, vol. 4, no. 5, Oct. 1996, pp. 731-742.

[Schultz97] Schultz, K.J. and P.G. Gulak, “Physical Performance Limits for Shared Buffer ATM Switches,” *IEEE Trans. Commun.*, vol. 45, no. 8, Aug. 1997, pp. 997-1007.

[Serpanos00] Serpanos, D.N. and P.I. Antoniadis, “FIRM: A Class of Distributed Scheduling Algorithms for High-Speed ATM Switches with Multiple Input Queues,” in *Proc. INFOCOM 2000*, Tel Aviv, Israel, Mar. 2000, vol. 2, pp. 548-555.

[Sivaram98] Sivaram, R., C.B. Stunkel and D.K. Panda, “HIPIQS: A High-Performance Switch Architecture using Input Queuing,” in *Proc. 12th Int. Parallel Processing Symposium*, Orlando, FL, Apr. 1998, pp. 134-143.

[Souza94] Souza, R.J., P.G. Krishnakumar, C.M. Özveren, R.J. Simcoe, B.A. Spinney, R.E. Thomas and R.J. Walsh, “GIGASwitch System: A High-performance Packet-switching Platform,” Digital Technical J., vol. 6, no. 1, 1994, pp. 9-22.

[Stallings92] Stallings, W., *ISDN and Broadband ISDN*, second edition, MacMillan, New York, 1992.

[Stiliadis95] Stiliadis, D. and A. Varma, “Providing Bandwidth Guarantees in an Input-Buffered Crossbar Switch,” in *Proc. INFOCOM ’95*, Boston, MA, Apr. 1995, pp. 960-968.

[Stoica98] Stoica, I., and H. Zhang, “Exact Emulation of an Output Queueing Switch by a Combined Input Output Queueing Switch,” in *Proc. 6th IEEE/IFIP IWQoS ’98*, Napa Valley, CA, May 1998, pp. 218-224.

[Stunkel95] Stunkel, C.B., D.G. Shea, B. Abali, M.G. Atkins, C.A. Bender, D.G. Grice, P. Hochschild, D.J. Joseph, B.J. Nathanson, R.A. Swetz, R.F. Stucke, M. Tsao and P.R. Varker, “The SP2 High-Performance Switch,” *IBM Systems Journal*, vol. 34, no. 2, 1995, pp. 185-204.

[Stunkel97] Stunkel, C.B., R. Sivaram and D.K. Panda, “Implementing Multidestination Worms in Switch-Based Parallel Systems: Architectural Alternatives and their Impact,” To appear in the *Proc. 24th ACM Annual International Symposium on Computer Architecture (ISCA ’97)*, May 1997, Denver, CO, pp. 50-61.

[Suzuki89] Suzuki, H., H. Nagano and T. Suzuki, “Output-buffer Switch Architecture for Asynchronous Transfer Mode,” in *Proc. ICC ’89*, Boston, MA, Jun. 1989, pp. 99-103.

[Tamir92] Tamir, Y. and G.L. Frazier, “Dynamically Allocated Multi-Queue Buffers for VLSI Communication Switches,” *IEEE Trans. Computers*, vol. 41, no. 6, Jun. 1992, pp. 725-737.

[Tamir93] Tamir, Y. and H.-C. Chi, “Symmetric Crossbar Arbiters for VLSI Communication Switches,” *IEEE Trans. Parallel and Distributed Systems*, vol. 4, no. 1, Jan. 1993, pp. 13-27.

[Tanenbaum96] Tanenbaum, A., *Computer Networks*, third edition, Prentice Hall, 1996.

[Tao94] Tao, Z. and S. Cheng, “A New Way To Share Buffer - Grouped Input Queueing In ATM Switching,” in *Proc. IEEE GLOBECOM ’94*, vol. 1, pp. 475-479.

[Tarjan83] Tarjan, R.E., “Data Structures and Network Algorithms,” Bell Laboratories, Murray Hill NJ, 1983.

[Tassiulas92] Tassiulas, L. and A. Ephremides, “Stability Properties of Constrained Queueing Systems and Scheduling Policies for Maximum Throughput in Multihop Radio Networks,” *IEEE Trans. on Automatic Control*, vol. 37, no. 12, Dec. 1992, pp. 1936-1948.

[Tassiulas98] Tassiulas, L., “Linear Complexity Algorithms for Maximum Throughput in Radio Networks and Input Queued Switches,” in *Proc. IEEE INFOCOM ’98*, Mar.-Apr. 1998, vol. 2, pp. 533-539.

[Thompson97] Thompson, K., G.J. Miller and R. Wilder, “Wide-Area Internet Traffic Patterns and Characteristics,” *IEEE Network - Magazine of Global Information Exchange*, vol. 11, no. 6, Nov.-Dec. 1997, pp. 10-23.

[Tobagi90] Tobagi, F., “Fast Packet Switch Architectures for Braodband Integrated Services Digital Networks,” in *Proc. IEEE*, vol. 78, no. 1, Jan. 1990, pp. 133-167.

[Troudet91] Troudet, T.P. and S.M. Walters, “Neural Network Architecture for Crossbar Switch Control,” *IEEE Trans. Circuits and Syst.*, vol. 38, no. 1, Jan. 1991, pp. 42 -56.

[Tsybakov97] Tsybakov, B. and N.D. Georganas, “On Self-Similar Traffic in ATM Queues: Definitions, Overflow Probability Bound, and Cell Delay Distribution,” *IEEE/ACM Trans. Networking*, vol. 5, no. 3, Jun. 1997, pp. 397-409.

[Turner88] Turner, J.S., “Design of a Broadcast Packet Switching Network,” *IEEE Trans. Commun.*, vol. 36, no. 6, Jun. 1988, pp. 734-743.

[Turner98] Turner, J.S., “Terabit Burst Switching,” Washington University Technical Report, WUCS-98-17, 1998.

[Turner99] Turner, J.S., “WDM Burst Switching,” in *Proc. INET ’99*, Jun. 1999.

[Vaidya88] Vaidya, A.K. and M.A. Pashan, “Technology Advances in Wideband Packet Switching,” in *Proc. GLOBECOM ’88*, Hollywood FL, Nov. 1988, pp. 668-671.

[Weller97] Weller, T. and B. Hajek, “Scheduling Nonuniform Trafiic in a Packet-Switching System with Small Propagation Delay,” *IEEE/ACM Trans. Networking*, vol. 5, no. 6, Dec. 1997, pp. 813-823.

[Yao00] Yao, S., B. Mukherjee, and S. Dixit, “Advances in photonic packet switching: An overview,” *IEEE Commun. Mag.*, Feb. 2000, pp. 84-94.

[Yeh87] Yeh, Y., M.G. Hluchyj and A.S. Acampora, “The Knockout Switch: A Simple, Modular Architecture for High-Performance Packet Switching,” *IEEE J. Sel. Areas Commun.*, vol. 5, no. 8, Oct. 1987.

[Yener99] Yener, B., E. Leonardi and F. Neri, “Algorithms for Virtual Output Queued Switching,” in *Proc. GLOBECOM ’99*, Rio de Janeiro, Brazil, Dec. 1999, vol. 2, pp. 1203-1210.

[Yoon95] Yoon, H., M.T. Liu, K.Y. Lee and Y.M. Kim, “The Knockout Switch Under Nonuniform Traffic,” *IEEE Trans. Commun.*, vol. 43, no. 6, Jun. 1995, pp. 2149-2156.

Appendix A

Bufferless Switch Throughput

Consider an $N \times N$ switch element without buffers. Assuming persistent traffic sources with uniformly distributed packet destinations, we can easily derive the switch throughput. The probability $P_N(X = m)$ that m packets arrive simultaneously for the same output equals

$$P_N(X = m) = \binom{N}{m} \left(\frac{1}{N}\right)^m \left(1 - \frac{1}{N}\right)^{N-m}. \tag{A.1}$$

The throughput T_N equals the sum over all i of the probability that *at least* one cell is destined to output i, divided by N,

$$T_N = \frac{1}{N} \sum_{i=0}^{N-1} P_{N,i}(X \geq 1), \tag{A.2}$$

which, by symmetry of the input traffic pattern, yields

$$T_N = P_N(X \geq 1) = 1 - P_N(X = 0). \tag{A.3}$$

By substituting Eq. (A.1) in Eq. (A.3) we obtain the throughput

$$T_N = 1 - \left(1 - \frac{1}{N}\right)^N, \tag{A.4}$$

which for large N tends to

$$T_\infty = \lim_{N \to \infty} 1 - \left(1 - \frac{1}{N}\right)^N = 1 - \frac{1}{e} \approx 0.632. \tag{A.5}$$

As there are no buffers, the cell loss ratio equals $1 - T_\infty = \frac{1}{e} \approx 0.368$.

Figure A.1 plots the throughput and the loss rate as a function of the number of switch output ports. The x-axis is logarithmic.

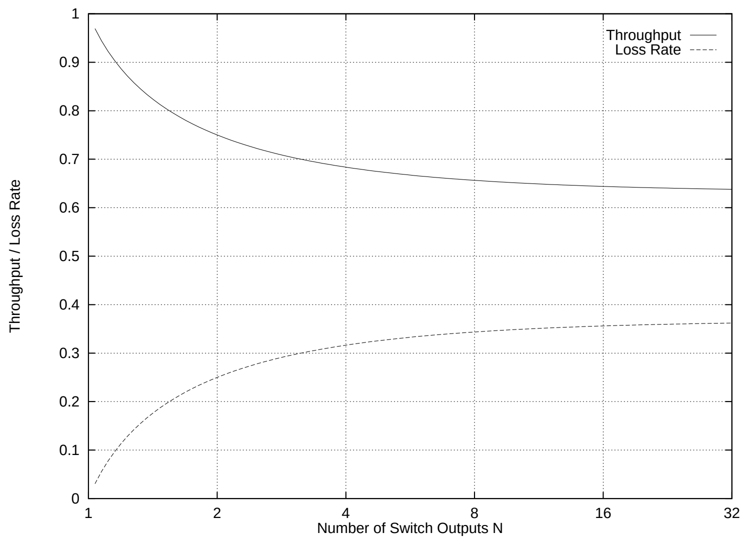


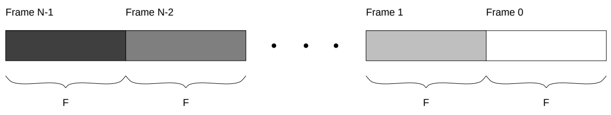
Figure A.1: Bufferless Switch Throughput and Loss Rate

Appendix B

On Frame Interleaving vs. Non-Interleaving

Consider an output port i of the switch element. On the input ports, there are N frames, numbered 0 through $N-1$, currently waiting to be transmitted to output port i, with lengths F_i, expressed in fixed-size packets per frame. The average frame length equals $F = \sum_{i=0}^{N-1} F_i$. The delay of a frame is defined to be equal to the delay of the last cell belonging to that frame, because only when the last cell has been received can the frame be processed further.

Consider operation in non-interleaving mode:



Frame 0 will have a delay $D_0 = F_0$,
frame 1 will have a delay $D_1 = F_0 + F_1$,
$\vdots$
frame $N-1$ will have a delay $D_{N-1} = \sum_{i=0}^{N-1} F_i$.

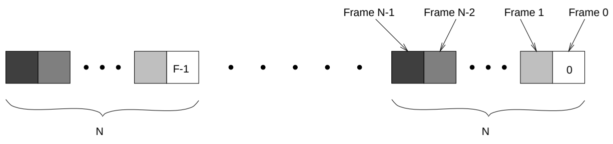
Hence, the average delay in the non-interleaving case $\overline{D_{ni}}$ equals

$$\overline{D_{ni}} = \frac{1}{N}\sum_{i=0}^{N-1} D_0 = \frac{1}{N}\sum_{i=1}^{N} iF_{N-i}. \tag{B.1}$$

Assuming that all frames are of equal size,

$$\overline{D_{ni}} = \frac{1}{N}\sum_{i=1}^{N} iF = \frac{(N+1)F}{2}. \tag{B.2}$$

Now consider the interleaving case:



Frame 0 will have a delay $D_0 = (F-1)N + 1$,
frame 1 will have a delay $D_1 = (F-1)N + 2$,
$\vdots$
frame $N-1$ will have a delay $D_{N-1} = (F-1)N + N = NF$.

For the average delay in the interleaving case $\overline{D_i}$ we find

$$\overline{D_i} = \sum_{i=1}^{N} \frac{(F-1)N + i}{N} = N(F-1) + \frac{N+1}{2}. \tag{B.3}$$

The improvement ratio $R = \frac{\overline{D_{ni}}}{\overline{D_i}}$ can now be expressed in terms of N and F,

$$R = \frac{\overline{D_{ni}}}{\overline{D_i}} = \frac{\frac{F}{F-1}}{\frac{2N}{N+1} + \frac{1}{F-1}}. \tag{B.4}$$

For large N we can simplify Eq. B.4 to

$$\lim_{N\to\infty} R = \frac{F}{2(F-1)+1} = \frac{F}{2F-1}, \tag{B.5}$$

whereas for large F it reduces to

$$\lim_{F\to\infty} R = \frac{1}{2} + \frac{1}{2N}, \tag{B.6}$$

until finally, for both large N and large F, the result is

$$\lim_{N\to\infty}\lim_{F\to\infty} R = \lim_{N\to\infty}\left\{\frac{1}{2} + \frac{1}{2N}\right\} = \frac{1}{2}, \tag{B.7}$$

and to demonstrate that the order in which the limits are taken does not matter

$$\lim_{F\to\infty}\lim_{N\to\infty} R = \lim_{F\to\infty}\left\{\frac{F}{2F-1}\right\} = \frac{1}{2}. \tag{B.8}$$

Figure B.1 plots the value of R versus the frame size F, with N as a parameter [Eq. (B.4)].

Note how for frame size $F = 1$ there is no improvement ($R = 1$ for any N), which is correct because in this case interleaving or non-interleaving makes no difference. Similarly, when there is only one frame to be transmitted ($N = 1$), R also equals one for any frame size F (this curve runs along the top of the figure and is therefore not clearly visible).

For large F or large N the graphs converge quickly to $R = \frac{1}{2}$, which leads to the conclusion that up to 50% can be gained in terms of frame delay by employing a non-interleaving frame mode.

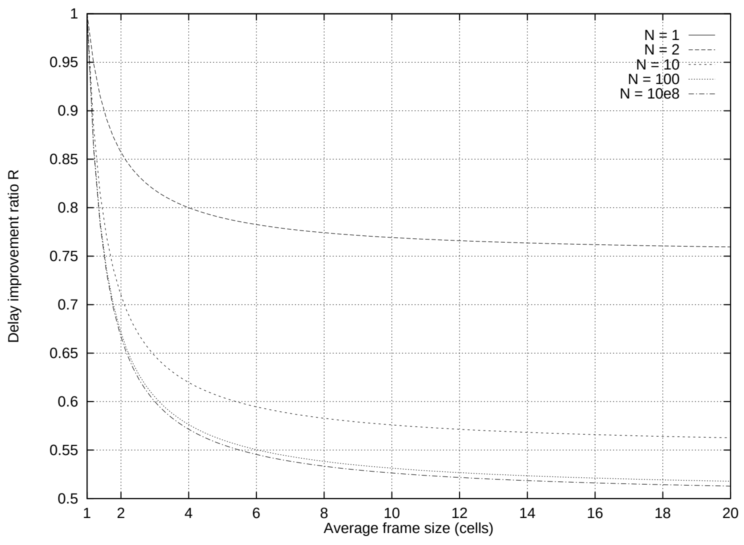


Figure B.1: Delay Improvement Ratio R

Appendix C

Performance Simulation Environment

C.1 Introduction

The performance simulation results in this dissertation have been obtained using a packet-level C++ model of the switch system. This model has been created with a self-written C++ cycle simulator library. Section C.2 gives a brief description of the environment, Section C.3 treats the simulation library, Section C.4 provides a detailed description of the various traffic models, and Section C.5 details the model at system level.

C.2 Environment

The performance simulation environment consists of the simulation library, the model, and a number of scripts used to facilitate pre- and post-processing tasks. It contains a highly configurable, executable model of the switch system.

Starting from a specification (`.spec`) file, a batch file (`.run`) containing the specified runs is generated. Executing this batch file will call the performance simulator (`perfsim`) for every run. The simulator will retrieve configuration information from the specified configuration (`.cfg`) file and from the command line parameters, with the latter taking precedence over the former to enable easy parameter variation. Upon succesful termination, the simulator generates a statistics file (`.stat`) containing statistics recorded during simulation, such as average delay, throughput, queue and memory occupancies, etc. The salient data can then be retrieved from this statistics file to be further processed using suitable plotting tools to generate graphic representations.

Within the model, two layers can be distinguished, see Fig. C.1:

- the core simulation library, which provides generic primitives to construct models, random number generators, statistics gathering facilities, etc. (see Section C.3), and
- the switch system model built using this library.

The following sections will present the features of the simulation library before proceeding to explain the switch system model itself in detail.

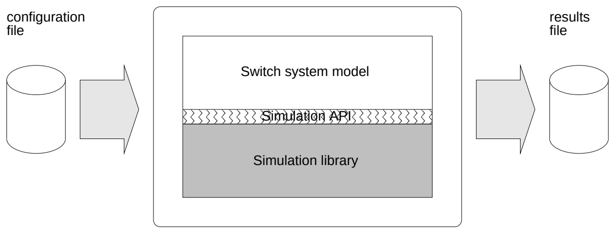


Figure C.1: Two-layer simulation model.

C.3 Simulation Library

The core feature of the simulation engine is that it is *cycle-driven*, as opposed to event-driven, trace-driven or execution-driven simulators, the main reason for this approach being the need for fast simulation. The entire simulation library (`libsim.a`) is written in C++.

C.3.1 Simulator concepts

- **Cycle-based**: the simulator keeps track of simulation time, which is represented by an integer value, starting from 0 (beginning of simulation), and indicating the number of simulator time units (cycles) elapsed. The time is incremented in integer intervals.

- **Process-based**: the entities being simulated are called “processes.” A process is the grain of concurrency, that is, all processes are independent concurrent entities. Associated with each process is a *cycle time* T. Every T time units, the simulator will activate the process by calling the **ClockTick()** method of the process, which is to be provided by the user. Processes have no behavior of their own; any behavior must be provided by the user in the **ClockTick()** method. It must return one of two values: `PROC_CONTINUE`, which causes the process to be rescheduled T time units into the future[1] to be reactivated at that time, or `PROC_FINISH`, which will cause the process to be deactivated and removed from the scheduler. Processes scheduled to be activated at the exact same time are *not* guaranteed to be activated *in any particular order*.

- **“Child” Processes**: Every process instance can create “children” that behave as separate processes, except that upon activation, the **ClockTick()** method will be invoked on their “parent” process, and not on the child. This allows these processes to share data (in particular, all children and the parent share the data members of the parent). Children can be distinguished by the integer child identifier parameter that is passed to the **ClockTick()** method. When creating a child process, you can specify this integer child identifier[2] (the parent always has identifier 0). Each child process can have its own cycle time, independent of all the others. A parent and its children can be viewed as a *cluster* of processes all acting on the same data. It is not possible to create children of children.

[1]Note that it is possible to change the process’s cycle time every time it is activated (by calling the **SetCycleTime()** method), so that processes can be scheduled to any time in the future, not just the initial cycle time. If the cycle time is not altered, the previous cycle time is used.

[2]It is a good idea to keep these identifiers unique among children from the same parent. However, this is not enforced. Identifiers must be larger than zero, as the parent has ID 0.

C.3.2 Communication concepts

Processes can act as stand-alone entities, not interacting with any other processes. However, processes can also communicate with other processes through the following primitives:

- **Channels**: these entities connect processes together. Channels can be used to exchange any type of data, but each channel can only pass one fixed type of data.[3] Channels can be written to and read from by calling the **Write()** and **Read()** methods. Note that channels act as a one-place buffer; that is, a write action to a channel does not take effect until the current simulation cycle is over. Thus, communication is synchronized at every simulator time update.

- **Delay Channels**: these are just like ordinary channels[4] with the additional feature of a built-in, specifiable delay. These channels basically work like a FIFO buffer. Data that is pushed in (written to the channel) pushes forward data that is already in the buffer. A delay of zero can also be used; this implies a bufferless channel, which provides a means for asynchronous, direct communication. However, this is not encouraged, as it implies that the order in which processes become active during the same cycle becomes important! A delay channel with a delay of 1 is functionally equivalent to a regular channel, as described above.

- **Interfaces**: there are three types of interfaces: Input-, output- and input/output-interfaces.[5] A process can only write to or read from a channel through an interface (input interfaces allow only allow reads (**Read()** method), output-interfaces only writes (**Write()** method), whereas i/o-interfaces allow both). Connecting interfaces together, taking into account the direction of the interfaces being connected, enables processes to communicate. Connecting two interfaces together automatically sets up a channel between them (also possible are delay channels, using the **Connect()** and **DelayConnect(int** $delay$**)** methods respectively). Note that the user never needs to either create a Channel explicitly, or call any Channel methods explicitly, as this is all handled automatically. The user will only specify interfaces on processes, connect interfaces together, and operate on interfaces.

- **Aliasing**: A special feature of interfaces that deserves to be mentioned separately is *aliasing*. Instead of being connected to another interface, an unconnected interface can serve as an alias for an already connected interface, using the **Alias(Interface<T> *itf)** method. This is useful for keeping an entity's interface consistent with respect to the outside world, even if the internal architecture and interfaces are subject to change. Naturally, the aliasing interface must be of the same type as the one being aliased.

Figure C.2 shows a sample model, consisting of four processes. Process 1 has an output interface a and an input interface b, process 2 input interfaces b, c and d and output interface a, process 3 input interface a and output interfaces b and c, process 4 input interface a, process 5

[3]In fact, the Channel object type is a template. Declaring a channel of type **Channel**<**T**> will allow objects of type **T** to be passed along this channel.

[4]The **DelayChannel**<**T**> class is a template class, derived from the **Channel**<**T**> base class. It adds no public methods.

[5]The actual classes are again templates, namely **I_Interface**<**T**>, **O_Interface**<**T**> and **IO_Interface**<**T**>. Only objects of type **T** can be passed through an interface.

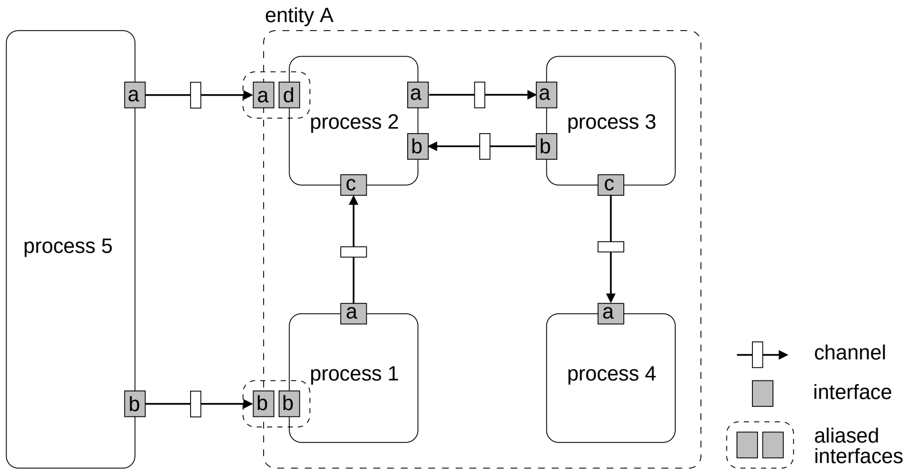


Figure C.2: Processes, interfaces and channels.

output interfaces a and b. Wrapper entity A is not a process, i.e., it has no behavior of its own, but does have two input interfaces a and b.

They are connected as shown, each input interface connecting to exactly one output interface, and vice versa. The small box on each channel symbolizes the built-in one-place buffer of each channel. The data type of each channel and interface is not indicated, but naturally only interfaces of the same data type can be connected. Output interfaces a and b of process 5 are connected to input interfaces a and b of entity A. By making input interface d of process 2 an alias of input interface a of entity A, and similarly input interface b of process 1 an alias of input interface b of entity A, processes 1 and 2 are now effectively connected to process 5 and can read from the connecting channels. If the implementation of entity A was changed such that process 1 now needs to be connected to interface $5a$ and process 2 to interface $5b$, the aliasing can simply be changed, without having to modify process 5 or the way it connects to entity A.

The simulator can be started by calling the static **Process::Simulate()** method. The simulation will continue until either all processes have finished or the static **Process::Terminate()** method is called. A simulation that has been interrupted by a **Process::Terminate()** call can be restarted (continued) by calling the (also static) **Process::Restart()** method.

C.3.3 Statistics gathering

A **Table** class is provided to gather statistical data. The **Record()** method allows you to enter data into a table. Each table keeps track of mean (**Mean()**), variance (**Var()**), standard deviation (**StdDev()**), and number of entries (**Count()**). The tables use the double numeric type. Integers can of course also be recorded; they will automatically be promoted to doubles.

A table can be configured to also record data into a histogram associated with the table. The minimum value, maximum value, and the number of buckets can be specified. A histogram also keeps track of the actual minimum and maximum values recorded.

95%-confidence intervals are computed if multiple simulation runs are executed. These intervals are computed on the mean of the data being recorded. Both the maximum number of runs

(NR) and the target 95%-confidence (CONF) interval can be configured. The simulation will terminate when either the target confidence or the maximum number of runs has been reached, whichever comes first.

To filter out inaccurate datapoints from the transient startup phase, a startup time can be specified during which no statistics will be recorded (calls to the **Record()** method will be ignored). When this time has elapsed, statistics gathering will start.

C.3.4 Random numbers

A random number generating facility is provided. The **RandomGenerator** class provides a random number generator based on the Park & Miller algorithm [Park88]. The **random()** method provides a uniformly distributed deviate between 0 and 1. The **RandomInteger(long** N**)** method returns a uniformly distributed integer deviate between 0 and $N-1$, inclusive. The **SetSeed(long** $seed$**)** method sets the initial seed, which allows exact reproduction of simulation runs.

A facility for non-uniformly distributed random variables is also provided.

C.4 Traffic Models

In general, a traffic model is characterized by two random processes to model both the temporal and the spatial characteristics of the traffic, i.e., one process to model the inter-arrival times between successive packet arrivals and one to model the distribution of packet destinations. The following two sections will describe the various models that are implemented in our performance simulation environment for these two distributions.

C.4.1 Inter-arrival time distributions

The inter-arrival time distribution determines *when* the next packet will arrive. We have the following three inter-arrival time distributions at our disposal: Bernoulli, Bursty (Markov), and "IP."

All traffic processes are always *identical and independently distributed* (i.i.d.) across all inputs.

Bernoulli traffic model

In every packet cycle, a packet is generated with probability p, and no packet is generated with probability $1-p$. Hence, the average offered input load $\lambda^{\circ}_{\text{Bernoulli}}$ equals p. As there is no burstiness in this traffic model, it usually leads to very optimistic performance estimations.

$$\lambda^{\circ}_{\text{Bernoulli}} = p. \tag{C.1}$$

Bursty traffic model

The bursty traffic model is based on a two-state Markov chain (see Fig. C.3), consisting of an ON and an OFF state. In the OFF state no traffic is generated ($\lambda_0 = 0$), whereas in the ON state, traffic is generated at line speed (one packet every cycle, $\lambda_1 = 1$). While in the ON state, the packet stream is divided into consecutive *bursts*; all packets in one burst have the same destination. Both the ON and OFF state have a minimum length of one burst, and each burst has a minimum size of one packet slot (the "bursts" in the OFF state just do not generate any packets). The bursty model is fully determined by three parameters, the probability p_{10} to go from ON to OFF state, vice versa p_{01}, and the burst parameter b.

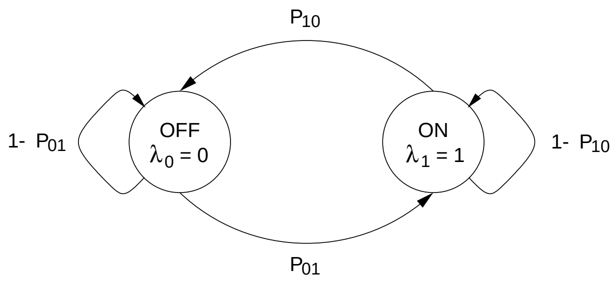


Figure C.3: Two-state Markov chain

The probability that the sojourn in the ON state equals n packet cycles is geometrically distributed:

$$P(T_{\text{on}} = n) = p_{10}(1 - p_{10})^{n-1},\ n \geq 1, \tag{C.2}$$

which leads to an average ON state length $\overline{T_{\text{on}}}$ of

$$\begin{aligned} \overline{T_{\text{on}}} &= \sum_{n=1}^{\infty} nP(B = n) = \sum_{n=1}^{\infty} np_{10}(1 - p_{10})^{n-1} && \text{(C.3)} \\ &= p_{10} \sum_{n=1}^{\infty} n(1 - p_{10})^{n-1} = -p_{10} \frac{d}{dp_{10}} \left\{ \sum_{n=0}^{\infty} (1 - p_{10})^n \right\} && \text{(C.4)} \\ &= -p_{10} \frac{d}{dp_{10}} \left\{ \frac{1}{1 - (1 - p_{10})} \right\} = -p_{10} \frac{-1}{p_{10}^2} = \frac{1}{p_{10}}. && \text{(C.5)} \end{aligned}$$

Similarly, the average OFF state length is

$$\overline{T_{\text{off}}} = \sum_{n=1}^{\infty} nP(I = n) = \frac{1}{p_{01}}. \tag{C.6}$$

In general, the arrival rate determined by the model in Fig. C.3 equals

$$\lambda^{\circ} = P_0 \lambda_0 + P_1 \lambda_1, \tag{C.7}$$

which, by determining the state probabilities P_0 and P_1, can be shown to be

$$\lambda^{\circ} = \frac{p_{10}}{p_{01} + p_{10}} \lambda_0 + \frac{p_{01}}{p_{01} + p_{10}} \lambda_1. \tag{C.8}$$

With $\lambda_0 = 0$ and $\lambda_1 = 1$, the average traffic load equals

$$\lambda^{\circ}_{\texttt{Bursty}} = \frac{\overline{T_{\text{on}}}}{\overline{T_{\text{on}}} + \overline{T_{\text{off}}}} = \frac{p_{01}}{p_{01} + p_{10}}. \tag{C.9}$$

Within each state, the burst sizes are also geometrically distributed, with parameter b:

$$P(B = n) = b(1-b)^{n-1},\ n \geq 1, \tag{C.10}$$

leading to an average burst size $\overline{B} = \frac{1}{b}$.

For OFF states that can be of zero length (cf. IP model), the following holds:

$$\begin{aligned} \overline{T_{\text{off}}} &= \sum_{n=0}^{\infty} nP(I = n) = \sum_{n=0}^{\infty} np_{01}(1-p_{01})^n & \text{(C.11)} \\ &= (1-p_{01}) \left\{ p_{01} \sum_{n=1}^{\infty} n(1-p_{01})^{n-1} \right\} = (1-p_{01})\frac{1}{p_{01}} = \frac{1-p_{01}}{p_{01}}. & \text{(C.12)} \end{aligned}$$

IP traffic model

The IP traffic model implemented in our simulation environment is a bursty model with a burst-size distribution based on [Thompson97], where results of extensive measurements are presented that indicate that a very large portion of internet backbone traffic consists of packets that take only a very limited number of sizes. Some highly characteristic sizes are 44 bytes (TCP acknowledgement and control packets), 552 and 576 bytes (driven by the default maximum segment size in many implementations) and 1500 bytes (characteristic of Ethernet traffic).

Assuming a packet payload size of 48 bytes, these sizes are equivalent to approximately 1, 12 and 30 packets. We approximated the cumulative percentage curve by a rather coarse 9-point discrete distribution, of which the probability density and the probability distribution functions are shown in Figs. C.4 and C.5 respectively. The average burst size equals approx. 8.3 packets.

C.4.2 Destination distributions

The destination distribution determines *where* the next packet will go. Often a uniform distribution is chosen, but traffic non-uniformity can appear in several forms: it can happen that a certain input transmits to one or a few of the outputs more than to the others, or even all of the inputs can prefer to send to certain outputs ("hotspots"). Additionally, the average load may vary from input to input and output to output.

In general, a destination distribution is characterized by an array of probabilities $[d_{ij}]$, with $0 \leq i, j < N$, where d_{ij} represents the probability that a packet (or an entire burst, in the case of Bursty or IP traffic) arriving on input i is destined to output j. These coefficients must satisfy $\sum_{j=0}^{N-1} d_{ij} = 1$ for every i.

Given that λ°_i is the offered load from input i, the resulting offered load on output j, $\lambda^{\bullet}_j$, equals

$$\lambda^{\bullet}_j = \sum_{i=0}^{N-1} \lambda^{\circ}_i d_{ij}. \tag{C.13}$$

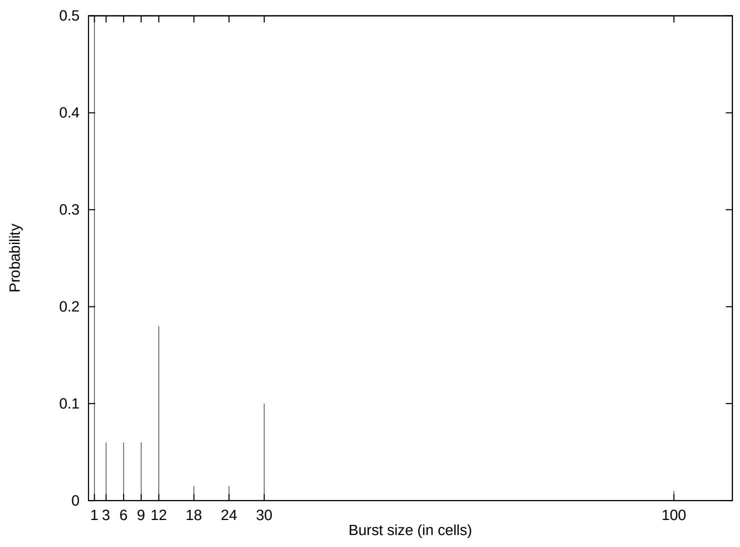


Figure C.4: IP traffic probability density function

If $\lambda_j^\bullet \geq 1$, output j is overloaded. Note that the achievable output load cannot exceed 100%.

The average offered output load $\lambda^\bullet$ equals

$$\lambda^\bullet = \frac{1}{N} \sum_{j=0}^{N-1} \lambda_j^\bullet. \tag{C.14}$$

The actual switch throughput μ satisfies Eq. (C.15):

$$\mu \leq \frac{1}{N} \sum_{j=0}^{N-1} \min\left(\lambda_j^\bullet, 1\right), \tag{C.15}$$

i.e., no output can exceed 100% throughput.

Three types of destination distribution are available in the performance model: *uniform*, *non-uniform*, and *hot-spot*.

Because we do not model input asymmetry, the average load from each input is the same in all models, i.e., $\lambda_i^\circ = \lambda^\circ$ for all i.

Uniform

The destination distribution used most often in simulation is uniform over all destinations. The probability d_{ij} of input i generating a packet for destination j is independent of i and j, and equals

$$d_{ij} = \frac{1}{N}, \tag{C.16}$$

for all i, j, where N is the total number of possible destinations.

Hence, from Eqs. (C.13) and (C.14), $\lambda_j^\bullet = \sum_{i=0}^{N-1} \lambda^\circ \frac{1}{N} = \lambda^\circ$ for all j, and $\lambda^\bullet = \lambda^\circ$.

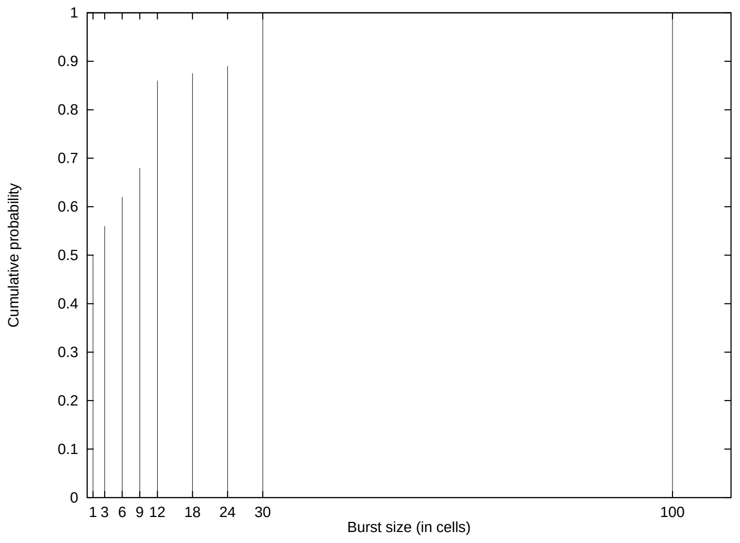


Figure C.5: IP traffic probability distribution function

Non-uniform

In reality, the destination distribution of the traffic need not be uniform at all. In fact, traffic can in general be expected to exhibit strongly non-uniform characteristics. To enable simulation of non-uniform destination distributions, it is possible to specify the matrix $\mathbf{D} = [d_{ij}]$, with $i \leq 0 < N$, $0 \leq j < N$, where d_{ij} represents the probability that a packet from input i is destined to output j. The offered output loads can be computed from Eqs. (C.13) and (C.14). Out of the following three conditions the first two *must* be satisfied:

1. $0 \leq d_{ij} \leq 1$,
2. for every i, $\sum_{j=0}^{N-1} d_{ij} = 1$.
3. for every j, $\sum_{i=0}^{N-1} d_{ij} \leq 1$.

The desired destination distribution must be specified in a separate `.distr` file, containing N lines, one line for each input with N entries, one for each output, per line. When the destination distribution (DM) is specified as non-uniform, this file is automatically read upon instantiation of the traffic sources.

Note that condition 3 is optional, although not satisfying this condition implies that certain output ports will be overloaded because in that case $\sum_{i=0}^{N-1} \lambda_i^{\circ} d_{ij} > 1$, which will lead to heavy congestion on those ports. Also note that all three conditions being satisfied simultaneously implies that for every j, $\sum_{i=0}^{N-1} d_{ij} = 1$.

Finally, the uniform distribution of the preceding section is just a special case with $d_{ij} = \frac{1}{N}$ for all i, j.

Hotspot

To study the effect of non-uniform, asymmetric destination distributions that overload certain output ports, a *hotspot* traffic model is available as well. The hotspot distribution is characterized by two parameters: the hotspot destination probability p_h (PHSP) and the number of hotspots n_h (NHSP).

Thus, we divide the set of destinations into two groups, namely, the hotspots and the non-hotspots. The probability that a packet is destined to a given hotspot destination equals p_h/n_h, and to a given non-hotspot destination $(1-p_h)/(N-n_h)$. The probability distribution within each group is uniform, that is, each hotspot destination has equal probability compared to each other hotspot destination, and the same holds for the non-hotspot destinations. The hotspots are chosen randomly from the set of outputs.

To make sure that the definitions make sense, the parameters have to satisfy the following condition:

$$0 < n_h \leq N \wedge \frac{n_h}{N} < p_h \leq 1, \tag{C.17}$$

i.e., there has to be at least one hotspot, and the amount of traffic sent to the hotspots must be more than that sent to the non-hotspots ($p_h > n_h/N$).

From Eq. (C.13), the offered output load $\lambda_j^\bullet$ of a hotspot equals

$$\lambda_j^\bullet = \sum_{i=0}^{N-1} \lambda^\circ \frac{p_h}{n_h} = \frac{N\lambda^\circ p_h}{n_h}, \tag{C.18}$$

whereas that of a non-hotspot is

$$\lambda_j^\bullet = \sum_{i=0}^{N-1} \lambda^\circ \frac{1-p_h}{N-n_h} = \frac{N\lambda^\circ(1-p_h)}{N-n_h}, \tag{C.19}$$

which is guaranteed to be < 1 by virtue of $p_h > n_h/N$.

The maximum throughput can then be computed as follows:

$$\begin{aligned}\mu_{\max} &= \frac{1}{N}\left(\sum_{j=0}^{n_h-1} \min\left(\frac{N\lambda^\circ p_h}{n_h}, 1\right) + \sum_{j=0}^{N-n_h-1} \frac{N\lambda^\circ(1-p_h)}{N-n_h}\right) \\ &= \frac{n_h}{N}\min\left(\frac{N\lambda^\circ p_h}{n_h}, 1\right) + \frac{(N-n_h)N\lambda^\circ(1-p_h)}{N(N-n_h)} = \min\left(\lambda^\circ p_h, \frac{n_h}{N}\right) + \lambda^\circ(1-p_h).\end{aligned}$$

C.5 System Model

Figure C.6 shows the general system model used for the performance simulations. Solid arrows represent data flow, dashed arrows flow control. The following components are present, from left to right:

- Arrival Process: these processes are the traffic sources; they generate the traffic patterns. The number of these processes per input is configurable (NS*NP). The traffic characteristics are governed by the traffic type and destination distribution (see Section C.4), which are configurable (TT and DM resp.).

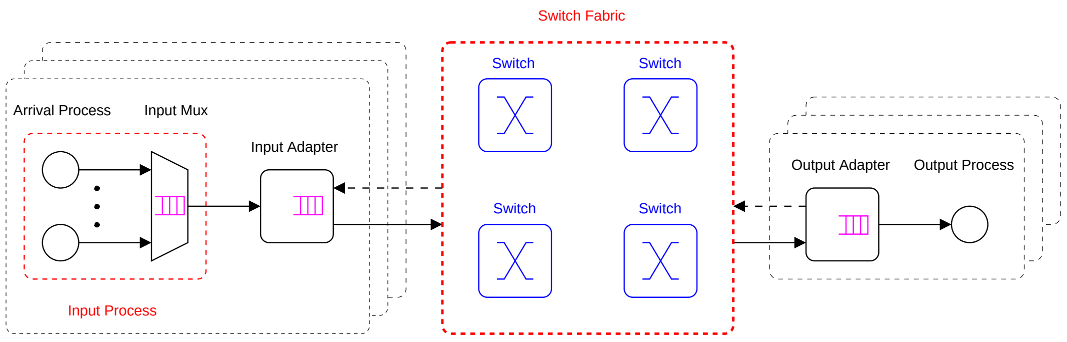


Figure C.6: Performance simulation system model

- Input Mux: the sources on each input are multiplexed together onto one link by the input mux. This mux has a queue (organized per priority) of configurable size (IMQS) to store packets should multiple arrivals occur in one cycle. With proper configuration, this queue should never fill up. If it does, the input is clearly oversubscribed, and excess packets are dropped.[6] There is no flow control between input mux and input adapter.
- Input Process: an input process simply consists of the set of arrival processes and their input mux on one input. It does not have any behavior of its own.
- Input Adapter: the input adapter is there to buffer packets that temporarily cannot be accepted by the switch owing to some full condition. To signal these full conditions from switch to adapter, there are two flow-control channels: the *shared-memory grant* and the *output-queue grant*. Depending on the type of adapter, this flow-control information is used to determine which packet to transmit.
  - FIFO (First In First Out): the FIFO adapter has one queue, sorted per traffic class. Only the HoL packets are eligible to be forwarded. Only if both the shared-memory grant and the output-queue grant allow, the HoL packet will be forwarded. Multicast packets are exempt from the output-queue grant check and are forwarded whenever the shared memory grant allows it.
  - VOQ (Virtual Output Queued): the VOQ adapter has $N + 1$ queues (each sorted per priority), one for each destination (switch output), plus one separate queue for multicast packets. The HoL packets of each of these $N + 1$ queues are eligible to be forwarded. A packet selection is made based on a round robin; destinations that have no grant will be excluded from the round robin until they receive a grant again. The multicast queue is included in the round robin, but its packets are not subjected to the output queue grant check. If the shared-memory grant is negative, no packet will be forwarded.
  - Simplified FIFO adapter: this adapter is similar to the FIFO adapter, because it also has one queue, sorted per priority. However, it does not recognize any output-queue grant information, and only looks at the shared-memory grant, which is a simple on/off signal per priority.

[6]Actually, this queue employs a fair “push-out” strategy. When a packet arrives and finds the queue full, one packet from the longest priority queue is dropped (pushed out) to make room for the incoming packet. This does not work with frame mode, so in that case the queue size must be set to infinite (IMQS = -1).

Note that all flow control is provided per priority, and that the arbitration of transmission among priorities is governed by an entity called FlowArbiter, whose behaviour is configurable. The queue sizes are configurable as well (IAQS). Excess packets are simply dropped, but in contrast to the input mux this also works with frame mode because a frame is either accepted entirely[7] or dropped entirely.

- Switch Fabric: the input adapters connect to the switch fabric, which is nothing more than a shell containing the actual switching elements. The fabric does not add any behaviour of its own, but merely provides the interfaces to the outside world in a unified manner. Both single-stage and multi-stage fabrics are supported. Single-stage fabrics also support port expansion.
- Switch: this is the actual switching element. Depending on the type of switch fabric there can be one or more switch instantiations. Several types of switches are implemented, such as an input-queued switch with FIFO queues, an input-queued switch with VOQ and configurable scheduling algorithms (i-SLIP, RRM, PIM), an output-queued switch, a CIOQ switch, etc.
- Output Adapter: these adapters receive traffic from the switch. They are equipped with a buffer of programmable size (OAQS). They can assert per-priority on/off flow control towards the switch if this buffer is full.
- Output Processes: these are the traffic sinks. They receive the packets from the output adapter, perform a number of checks such as correct destination, sequence number and frame sequence integrity, etc., and record statistics such as packet count (throughput) and packet delay for performance evaluation purposes and finally dispose of the packets.

[7]The queue size is not “hard”, meaning that a frame already being received is allowed to exceed the programmed queue length.