POLITECNICO DI TORINO

SCUOLA DI DOTTORATO
Dottorato in Ingegneria Elettronica e delle Comunicazioni - XVIII Ciclo

Tesi di Dottorato

High-Performance Packet Switching Architectures

Enrico Schiattarella

Tutore:
Prof. Fabio Neri

Coordinatore del corso di dottorato:
Prof. Ivo Montrosset

Marzo 2006

Abstract

Packet switches are at the heart of modern communication networks. Initially deployed for local- and wide-area computer networking, they are now being used in different contexts, such as interconnection networks for High-Performance Computing (HPC), Storage Area Networks (SANs) and Systems-on-Chip (SoC) communication. Each application domain, however, has peculiar requirements in terms of bandwidth, latency, scalability and delivery guarantee. These requirements must be carefully taken into account and have a major impact on the design of the switch.

In this thesis we present two novel switching architectures, aimed at shared-memory supercomputing and storage networking respectively. We describe the general architecture of the two systems and discuss how specific requirements and current technology trends have impacted the design. More important, we present some architectural innovations that address important issues concerning performance and scalability of input-queued switches.

In particular, we propose techniques that enable the construction of distributed (multi-chip) schedulers for large crossbars, develop a scheme for integrated scheduling of unicast and multicast traffic and study flow-control mechanisms that allow switches to achieve lossless behavior while providing fine-grained control of active flows. Simulation is used to understand the impact of the proposed solutions and evaluate system performance.

Table of contents

Acknowledgments

Despite the numerous challenges and the many difficulties found along the way, my years as a Ph.D. student at Politecnico di Torino have really flown by.

The Telecommunication Networks Group has been an excellent environment to undertake doctoral studies. I am sincerely grateful to professors Marco Ajmone Marsan and Fabio Neri for attracting so many talented individuals over the years and forming such a brilliant research group. I wish to thank personally the people I have closely worked with and who have had a major role in my education as a researcher: professors Emilio Leonardi, Andrea Bianco and Paolo Giaccone. My gratitude also goes to all the other members of the group, including fellow students, for being friendly, helpful and creating such a lively and entertaining environment. A special thank to prof. Monica Visintin, for her gentle attitude, dedication and encouraging words.

A significant part of my work has been performed at external research institutions. I wish to thank Ronald Luijten of the IBM Zurich Research Lab, for giving me the opporunity to visit his group for more than one year, Dr. Cyriel Minkenberg, for his precious mentoring, and all the other members of the group, for being relentless sources of inspiration and enthusiasm. Thanks also to Dr. Francesco Masetti-Placci, for allowing me to spend a summer at Alcatel R&I, in beautiful Paris.

I gratefully acknowledge prof. Andrzej Jajszczyk of the AGH University of Science and Technology and prof. Angelos Bilas of the University of Crete, who kindly accepted to review the entire manuscript.

Finally, a whole-hearted thank to my family. Your trust made this possible.

Chapter 1

Introduction

1.1 Background

The history of packet-switched networks dates back to the '60s, when deployment of the ARPANET, ancestor of the Internet, was initiated. In the '90s the Internet became a global and ubiquitous networking infrastructure, used for business, entertainment and scientific purposes. Since then, the bandwidth demand of the Internet community has been steadily increasing at exponential rates. To satisfy it, researchers and engineers have studied extensively the design of high-performance switching fabrics, that are at the heart of Internet routers. Today's commercial Internet routers offer aggregate bandwidths on the order of terabits per second and employ sophisticated algorithms for packet buffering, processing and scheduling.

The success of this technology has led researchers to investigate its usage in other domains, where the communication subsystem has become a primary performance bottleneck. Packet switching is being used to build interconnection networks for High-Performance Computing (HPC) systems, where a large number of computing nodes and memory banks must be interconnected. It is replacing the traditional bus-based interconnection between servers and storage devices, giving birth to Storage Area Networks (SANs). More recently, it is also being used for Systems-on-Chip (SoC) interconnects.

While the benefits of using packet switching in these domains have long been recognized, it is important to remember that each of them has its specific set of requirements, significantly different from those typical of computer networks. Table 1.1 summarizes the requirements for packet switches used in computer networks as well as in the two other domains we are considering. The most significant differences are in terms of latency, aggregate bandwidth and delivery guarantee.

Moreover, current technology trends are playing a significant role in the design of packet switches. Issues such as power consumption, chip I/O bandwidth limitations and packaging constraints are becoming primary concerns for designers.

	IP Routers	**Fibre Channel SAN Switches**	**HPC Interconnects**
Throughput	*Very Important*	*Very Important*	Moderately Important
Latency	Not Important	Moderately Important	*Very Important*
Delivery Guarantee	Can Tolerate Small Losses	*Losses not Acceptable*	*Losses not Acceptable*
Line Rate/ Port Count	$<$ 10 Gb/s $\sim$ 100 ports	$<$ 10 Gb/s $\sim$ 100 ports	$\geq$ *40 Gb/s* $\sim$ *1000 ports*

Table 1.1. Requirements of domain-specific interconnection networks.

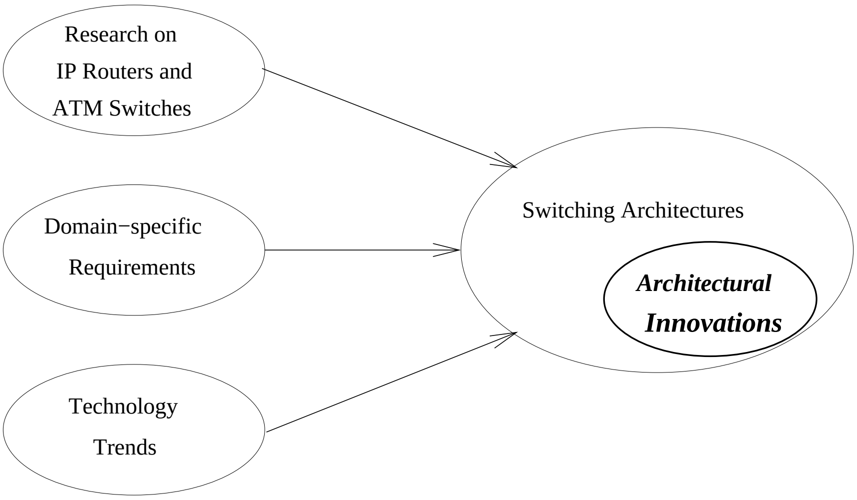

Research on IP Routers and ATM Switches
Domain−specific Requirements
Technology Trends
Switching Architectures
Architectural Innovations

Figure 1.1. Contributions and context of the thesis.

1.2 Contributions

In this thesis we present two novel switching architectures, aimed at HPC interconnects and Storage Area Networks respectively. We discuss how the specific requirements of the respective domains and current technology trends have influenced the design. More importantly, we present some of the architectural innovations that allow them to satisfy the demanding needs of their operating environments. The contributions and the context of this work are illustrated in Figure 1.1.

Work described in Part I was performed in the context of OSMOSIS, a research project developed at the IBM Zurich Research Lab, in collaboration with Corning, Inc. The project aims at building a demonstrator interconnect for HPC systems, whose building block is a switch featuring an all-optical data-path and electronic control-path. The system is designed to provide state-of-the-art performance and scalability.

In Part II we discuss the architecture of a director-class Fibre Channel switch designed for today's data-center. The architecture presents a number of important features, such as an asynchronous design and the presence of a central arbiter that allow the switch to achieve lossless behavior and isolate congesting flows.

Although the solutions we present have been developed to specifically address the challenges posed by the design of these two architectures, we believe that they are valuable in a more general context, as they address important issues concerning the performance and scalability of packet switches.

1.3 Outline of the Thesis

The thesis is organized as follows:

Chapter 2 introduces basic concepts and the terminology used in the rest of the thesis. It provides an overview of switching architectures and a brief survey of scheduling algorithms.

Chapter 3 contains an overview of supercomputing systems and interconnection networks. It explains how several factors, such as node architecture and partitioning of the memory space influence the requirements of the communication subsystem and describes the two most important classes of interconnection networks.

Chapter 4 describes the OSMOSIS project, explains the rationale for a hybrid opto-electronic architecture and illustrates the switch data- and control-path.

Chapter 5 is devoted to the first specific problem we have considered: how to build schedulers for large crossbars using multiple chips and overcoming the delay and I/O bandwidth limitations caused by distribution.

Chapter 6 addresses the problem of scheduling unicast and multicast traffic concurrently over a single fabric, achieving high overall performance and providing fairness guarantees.

Chapter 7 summarizes work described in Part I and the results we have obtained.

Chapter 8 opens Part II of the thesis, describing the evolution of the server-storage interface and illustrating how Storage Area Networks improve the organization of storage resources.

Chapter 9 introduces the switching architecture for SANs, focusing in particular on the mechanisms used to achieve loss-free operation and isolate congesting flows.

Chapter 10 contains a simulation-based study of system performance under unicast traffic, analyzing the effects of system parameters (buffer sizes, fabric and link speed-up) and traffic characteristics (uniformity, packet size distribution).

Chapter 11 studies performance under multicast traffic.

Chapter 12 draws conclusions from the results of Part II and concludes the thesis.

A table of acronyms used in the thesis can be found at the end of the document.

Chapter 2

Packet Switching Basics

In this chapter we introduce the basic concepts and the terminology used in the rest of the thesis. We first present the general architecture of a packet switch and discuss the main distinguishing feature: buffer placement. After an overview of output-queued (OQ), input-queued (IQ) and combined input-output-queued (CIOQ) switches, we focus on the problem of scheduling unicast and multicast traffic in IQ switches. We provide a survey of the most popular scheduling algorithms and discuss their characteristics in terms of performance and complexity.

Packet switching is a broad field, which has been studied extensively for decades. A comprehensive treatment of the topic can be found in [1], [2] and [3].

2.1 Definitions

A *packet switch* is a network device that receives packets on *input ports* and forwards them on the appropriate *output ports*.

The arrival of packets at the switch inputs can be modeled with a discrete-time stochastic process. At every timeslot at most one fixed-size data unit, called *cell* can arrive on each input. Variable-size packets can be considered as "bursts" of cells received at the same input in subsequent timeslots and directed to the same output.

We denote with λ_{ij} the average arrival rate on input i of cells directed to output j, normalized to the input/output link speed. The *offered load from input* i is the (normalized) rate at which cells enter the switch on input i and is represented by the term $\sum_{j=1}^{N} \lambda_{ij}$, where N is the number of input/output ports. Conversely, the *offered load to output* j is the (normalized) rate at which cells destined to output j enter the switch and is equal to the sum $\sum_{i=1}^{N} \lambda_{ij}$.

Traffic is *admissible* if no input/output links are overloaded, i.e. if the arrival rate at the inputs and the offered load to the outputs are less than or equal to the capacity of the input/output links. Formally, the admissibility conditions can be stated as:

$$\sum_{j=1}^{N} \lambda_{ij} \leq 1 \quad \forall i = 1,\ldots,N$$

$$\sum_{i=1}^{N} \lambda_{ij} \leq 1 \quad \forall j = 1,\ldots,N$$

In these conditions it is theoretically possible for the switch to forward to the outputs all the cells it receives on the inputs in finite time.

Traffic is *uniform* if a cell entering the switch can be directed to any output with equal probability:

$$\lambda_{ij} = 1/N \; \forall \, i,j$$

It is *independent and identically distributed (i.i.d.)*, also called *Bernoulli*, if the probability that a cell arrives at an input in a certain timeslot:

- is identical to and independent from the probability that a cell arrives at the same input in a different timeslot AND

- is independent from the probability that a cell arrives at another input.

The performance of a packet switch is mainly measured in terms of *throughput* and *latency*. Throughput is the (normalized) rate at which the device forwards packets to the outputs, latency is the time taken by a packet to traverse the switch. A switch achieves 100% throughput if it is able to sustain an offered load to all outputs equal to 1, under the hypothesis that traffic is admissible.

2.2 General Architecture of a Packet Switch

Figure 2.1 shows the architecture of a packet switch with N input/output ports. Packets are received on an input port and enter an *ingress adapter*, where they are stored (if necessary) and processed. Processing may include look-up of the destination port, recalculation of header fields (TTL, CRC, etc.) and filtering. Packets are then transmitted through the *switching fabric* and reach the *egress adapters*, where they are stored (if necessary) and prepared for transmission on the output links. If the switching fabric operates only on fixed-size data units, variable-size packets have to be segmented on the ingress adapter and reassembled on the egress adapter. Usually an ingress adapter is coupled to an egress adapter and they physically reside on a single board called *linecard* that can host multiple bi-directional ports.

A switch is *synchronous* if the linecards and the fabric are coordinated by mean of global clock signal and all ingress adapters start cell transmission at the same time. If the

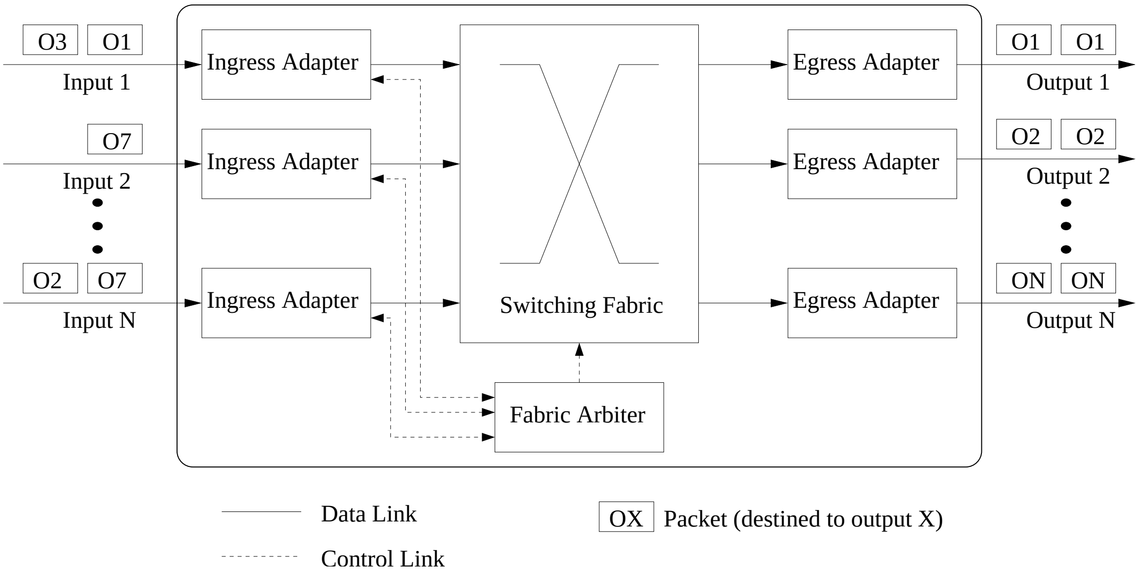


Figure 2.1. General architecture of a packet switch.

switch is *asynchronous*, on the contrary, the linecards and the fabric work on independent clock domains and transmission from different ingress adapters is not coordinated. In general synchronous switches internally operate on fixed-size cells, whereas asynchronous switches may natively support variable-size packets. Synchronous architectures are more popular because synchronicity simplifies many aspects of the design and implementation of the device. However, asynchronous switches have advantages as well, so they are being actively researched [4–6]. In the rest of this chapter we will implicitly refer to synchronous, cell-based switches.

2.3 Switching Fabric

2.3.1 Fabric properties

The switching fabric sets up connections between ingress and egress adapters. It is *non-blocking* if a connection between an idle input and an idle output can always be set-up, regardless of which other connections have already been established. This is a very desirable property, because it helps the switch in forwarding multiple packets concurrently, thus increasing throughput and reducing latency.

The fabric may run at a higher data rate than the linecards; in this case the ratio between the data rate of the fabric ports and that of the switch ports is called *speed-up*. For example, in a synchronous switch with speed-up two, at every time slot ingress/egress

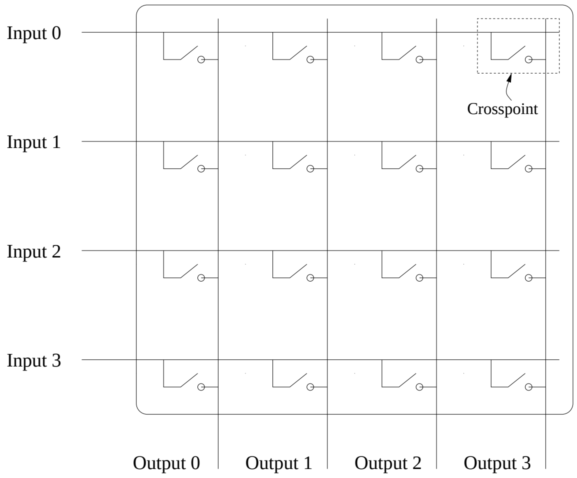


Figure 2.2. A 4×4 crossbar.

adapters can transmit/receive two cells to/from the fabric. When speed-up is used, the egress adapters can receive cells from the fabric at a higher rate than they can transmit on the output links, so they need buffers to temporarily store cells in excess. The term speed-up generally refers to the case in which both input and output fabric ports run faster than the switch ports; however, it is possible to have output speed-up only, i.e. to have only fabric output ports run at a higher data rate. Speed-up on the fabric inputs only is possible but has no practical use.

2.3.2 Crossbar

The crossbar is a very simple fabric that directly connects n inputs to m outputs, without intermediate stages. From a conceptual point of view, it is composed by $n + m$ lines, one for each input and one for each output, and $n \times m$ *crosspoints*, arranged as depicted in Figure 2.2. Input i is connected to output j if crosspoint (i,j) is *closed*.

Every output can be connected to only one input at a time, i.e. at most one crosspoint can be closed on a column. However, one input can be connected to multiple outputs at the same time by closing the corresponding crosspoints on the input row. In this case the signal at the input port is replicated to all the outputs for which the crosspoint is closed. The fabric has intrinsic support for *multicast* (one-to-many) communication. The crossbar is obviously non-blocking: an idle input (output) has all crosspoints its row (column) open, thus it is enough to close the crosspoint at the intersection to connect them.

The simplicity of the crossbar and its non-blocking property make it a very popular choice for packet switches. The main drawback is its intrinsic quadratic complexity, due to the presence of $n \times m$ crosspoints. Crossbars implemented on a single chip may also be limited by the amount of I/O signals that must be mapped to chip pins. However, it is possible to build a large multi-chip crossbar by properly interconnecting smaller single-chip ones [7]. The complexity in terms of gates remains quadratic.

2.4 Buffering Strategies

Due to traffic independence, the switch may receive in the same time slot multiple cells directed to the same output. In this case there is a *conflict* between inputs caused by *output contention*. It is not possible to forward one of the contending cells and discard all the other, because the drop rate would be unacceptable for any practical application. Therefore, the switch is endowed with internal buffers to store cells that cannot be transmitted immediately on the output link. The buffering strategy, mainly if the cells are buffered before being transferred through the switching fabric or after, is a major architectural trait and strongly influences performance, scalability and cost of a switch [8].

2.4.1 Output-queued (OQ)

In OQ switches all cells arriving at the fabric inputs are immediately transferred through the switching fabric and stored at the outputs. At every timeslot up to N cells directed to the same output can arrive, so the fabric must operate with speed-up $S = N$ and the memory bandwidth at each egress adapter must be equal to N times the line rate of the switch ports[1] (Figure 2.3).

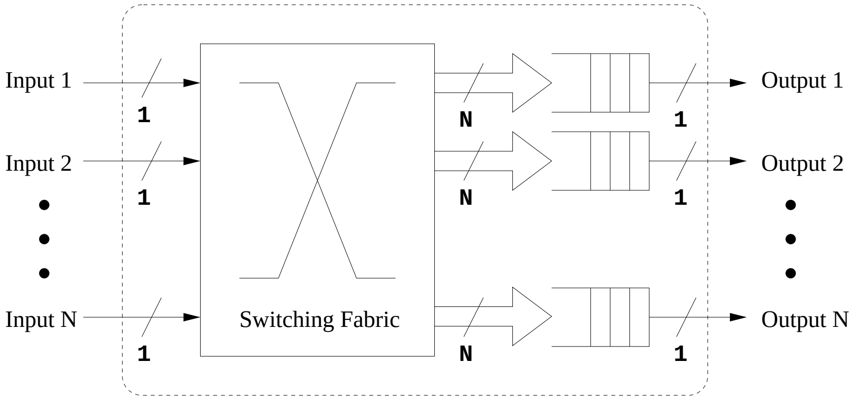


Figure 2.3. An Output-queued switch.

If multiple cells are buffered at an egress adapter, it is necessary to decide in which order they will be transmitted on the output link. This choice allows the switch to prioritize

[1] For simplicity we only consider memory *write* bandwidth.

different flows but does not have an impact on throughput. The OQ switch offers ideal performance, i.e. it achieves 100% throughput under any traffic pattern.

The problem with OQ switches is scalability: fabric speed-up and, above all, egress adapters memory bandwidth, grow linearly with N. As the bandwidth offered by commercial memories is on the same order of link rates, the OQ architecture is a suitable choice only for systems with a small number of ports or low link rates.

2.4.2 Input-queued (IQ)

In IQ switches the fabric transfers to the egress adapters only cells that can be transmitted immediately on the output links. Those that are blocked because of output contention are buffered on the ingress adapters (Figure 2.4).

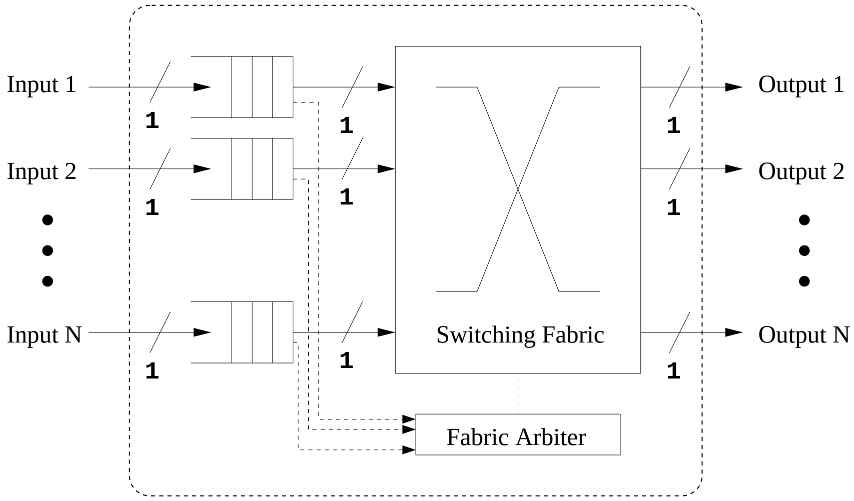


Figure 2.4. Input-queued switch.

This strategy has the following crucial consequences:

- buffers are not needed on the egress adapters, because at every timeslot the cell received from the switching fabric can be transmitted immediately on the output link[2];
- the switching fabric does not need speed-up, because it must be able to deliver at most one cell per timeslot to each egress adapter;
- the memory bandwidth of the buffers on the ingress adapters is equal to the switch ports line rate, irrespective of N, because at most one cell per timeslot arrives at each input;

[2] We neglect flow-control issues and assume that a cell can always be transmitted on an idle output link.

- a scheduler is required to decide which among multiple cells contending for the same output will be transferred; the fabric must be configured accordingly.

In the simplest case, arriving cells are stored in FIFO queues and each ingress adapter can only transmit the cell that is at the head of its queue. This constraint leads to a phenomenon called "Head-of-the-line (HOL) Blocking": a cell that is at the head of its input queue and cannot be transferred because of output contention blocks all the other cells in the same queue. Blocked cells may be destined to outputs for which no other input is contending, so the opportunity to transfer a cell is lost. HOL-blocking can severely degrade performance: for large values of N it limits switch throughput to about 58% under uniform i.i.d. traffic [8].

This level of performance is not acceptable, so in the past there have been many attempts to overcome the problem, in general by relaxing the FIFO constraint and allowing the scheduler to consider multiple cells from the same queue. In recent years increased CMOS densities have made feasible a new queueing architecture, called Virtual Output Queueing, that completely eliminates HOL blocking and allows IQ switches to achieve high performance.

2.4.3 IQ switches with Virtual Output Queueing (VOQ)

Virtual Output Queues (VOQs) are sets of independent FIFO queues, each of which is associated to a specific output [9]. In an IQ switch it is possible to avoid HOL-blocking by deploying a set of N VOQs on each ingress adapter (Figure 2.5). With VOQs, cells

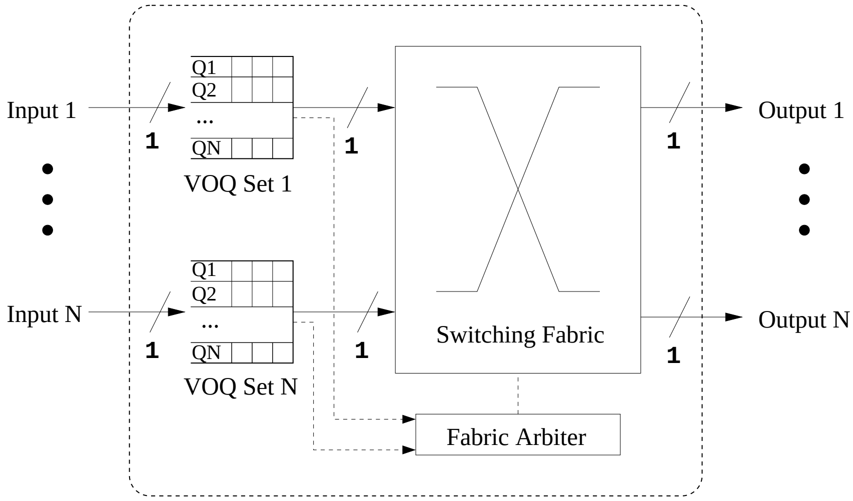


Figure 2.5. Input-queued switch with Virtual Output Queues.

destined to different outputs can be served in any order and do not interfere with each

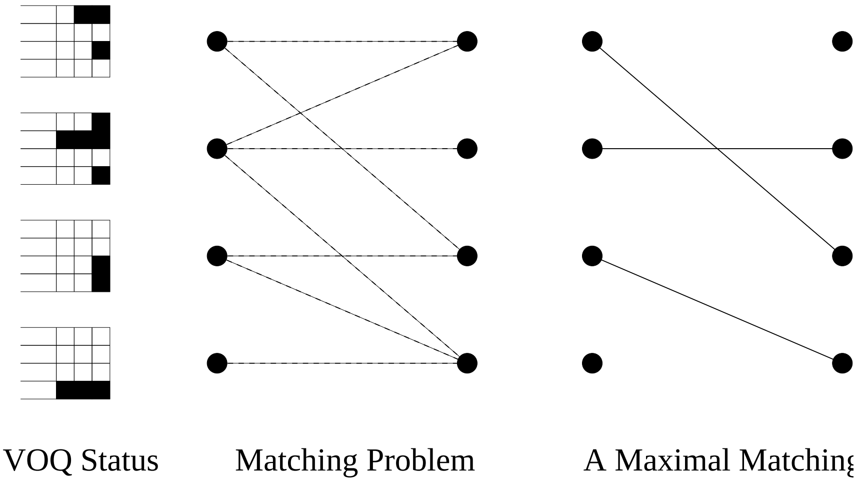


Figure 2.6. A Bipartite Graph Matching (BGM) problem.

other; cells destined to the same output, on the contrary, are served with a FIFO policy to preserve the ordering of packets belonging to the same flow.

At every timeslot the scheduler must decide which cells to transfer through the switching fabric, subject to the constraints that at most one cell can depart from an ingress adapter and at most one cell can be delivered to an egress adapter. The problem is equivalent to calculating a matching on a bipartite graph, as illustrated in Figure 2.6. Nodes on the left and right side represent fabric inputs and outputs respectively; dashed lines (edges) represent non-empty VOQs, i.e. cells that can be chosen for transfer. A *matching* is a set of edges such that each input is connected to at most one output and each output to at most one input.

A matching is *maximum size* if it contains the highest number of edges among all valid matchings; it is *maximal* if it is not possible to add new edges without removing previously inserted ones. For instance, the matching shown in Figure 2.6 is maximal but not maximum: no edges can be added, but it is easy to verify that there exists valid matchings with four edges. Edges can be assigned weights, such as the number of cells enqueued in the corresponding VOQ, or the time the cell at the head-of-the-line has been waiting for service. If weights are defined, the *Maximum Weight Matching (MWM)* is the one that maximizes the sum of the weights associated to the edges it contains.

IQ switches with VOQs can achieve 100% throughput under any i.i.d. traffic pattern, but only if very sophisticated scheduling algorithms are employed [10]. These algorithms are in general difficult to implement in fast hardware and too complex to be executed in a single timeslot. However, as we will discuss in Section 2.5, a number of heuristic matching algorithms that achieve satisfactory performance with reasonable complexity have been devised. Therefore input-queueing with VOQs is today the preferred architecture for the construction of large, high-performance packet switches. From this point on, when discussing IQ switches we will implicitly assume that VOQs are present.

2.4.4 Combined Input-Output-Queued (CIOQ) Switches

OQ and IQ switches represent two diametrically opposing points in the trade-off between speed-up and scheduling complexity. The former employ maximum speed-up but require no scheduling, the latter run without speed-up but need complex schedulers. CIOQ switches (with VOQs) represent an intermediate point: they buffer packets both at the inputs and at the outputs, employ moderate speed-up S $(1 \leq S \leq N)$ and use simpler schedulers.

Early simulation studies of CIOQ switches showed that, under a variety of switch sizes and traffic patterns, a small speed-up (between two and five) leads to performance levels close to those offered by OQ switches. These hints led a number of researchers to analytically investigate the maximum performance achievable by CIOQ switches. Among the many results that were published, these are particularly significant:

- With a speed-up $S = 2$ and proper scheduling algorithms, a CIOQ switch can exactly *emulate* an OQ switch, for any switch size and under any traffic pattern [11, 12]. “Emulating” means producing exactly the same cell departure process at the outputs given the same cell arrival process at the inputs.

- A CIOQ switch employing any maximal matching algorithm with a speed-up of two achieves 100% throughput under any traffic pattern, under the restriction that no input or output is oversubscribed and that the arrival process satisfies the strong law of large numbers [13].

These results prove that with moderate speed-up the performance of an IQ switch can be dramatically improved and that it can even reach the performance of an OQ switch if proper scheduling is used. A small fractional speed-up ($S < 2$) is also typically used to compensate for various forms of overhead, such as additional headers that must be internally prepended to cells and padding imposed by segmentation [14].

2.5 Scheduling Unicast Traffic in IQ Switches

2.5.1 Optimal Scheduling Algorithm

The optimal scheduling algorithms for an IQ switch, i.e. the one that maximizes throughput, is the Maximum Weight Matching (MWM), when queue lengths are used as weights [15]. McKeown et al. noted that, with this choice of the weights, specific traffic patterns can lead to permanent starvation of some queues [10]. However, they also proved that 100% throughput is still achieved for any i.i.d. traffic pattern if the ages of HOL cells are used as weights; in this case starvation cannot happen. The most efficient known algorithm for calculating the MWM of a bipartite graph converges in $O(N^3 \log N)$ time [16]. Despite polynomial complexity, this algorithm is not practical for high-performance packet switches, because it is difficult to implement in fast hardware and cannot be executed in the short duration of a timeslot. For this reason, a number of heuristic algorithms have been developed.

2.5.2 Parallel Iterative Matching Algorithms

Parallel iterative matching algorithms are the most popular class of heuristic matching algorithms. All inputs in parallel try to match themselves to one output by using a request-grant protocol. VOQ selection at the inputs and contention resolution at the outputs are performed by *arbiters* (also called *selectors*) using round-robin or random criteria. The process is iterated multiple times, until a maximal matching is obtained or the maximum number of iterations is reached. On average these algorithms converge in $\log_2 N$ iterations, but in the worst case they can take N.

PIM

PIM [17] (Parallel Iterative Matching) is one of the first parallel iterative matching algorithms that have been proposed. In every timeslot the following three *phases* are executed and possibly repeated multiple times:

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

2. *Grant:* every output that has been requested by at least one input *randomly* selects one to grant.

3. *Accept:* if an input receives more than one grant, it selects *randomly* one to accept.

The main disadvantage of PIM is that it does not perform well, as it achieves only 63% throughput with a single iteration under uniform i.i.d. traffic. Moreover, it employs random selection, which is difficult and expensive to perform at high speed and can cause unfairness under specific traffic patterns [18].

RRM

RRM (Round-Robin Matching) [18] addresses some of the drawbacks of PIM by replacing random selection with round-robin. The selection logic at each input and output is composed by a round-robin selector and a pointer. Pointers at the outputs are named *grant pointers*, whereas those at the inputs *accept pointers*.

Every iteration of RRM entails the following three phases:

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

2. *Grant:* every output that has been requested by at least one input selects one to grant in round-robin order, starting from the position indicated by the grant pointer. The pointer is advanced (modulo N) to one input beyond the one just granted.

3. *Accept:* if an input receives more than one grant it selects one to accept in round-robin order, starting from the position indicated by the accept pointer. The pointer is advanced (modulo N) to one output beyond the one just accepted.

The performance of RRM is very close to that of PIM, so still quite poor.

i-SLIP

i-SLIP [19] is a improvement of RRM that, with an apparently minor modification, achieves much higher performance. The three phases are modified as follows:

1. *Request:* same as RRM.

2. *Grant:* every output that has been requested by at least one input selects one to grant in round-robin order, starting from the position indicated by the pointer. The pointer is advanced (modulo N) to one input beyond the one just granted *if and only if the released grant is accepted in the accept phase*.

3. *Accept:* same as RRM.

Moreover, the grant and accept pointers are updated only in the first iteration; a detail that is crucial to prevent starvation of any VOQ under any traffic pattern.

i-SLIP performs extremely well: under uniform i.i.d. traffic it achieves 100% throughput with a single iteration, because it guarantees *desynchronization* of the grant pointers. When the switch is loaded at 100% and traffic is uniform i.i.d, all VOQs are backlogged. Assume that the grant pointers at multiple outputs point to the same input, i.e. they are *synchronized*. The input receives multiple grants, accepts one and moves the accept pointer. Thanks to the modification of the grant phase, only one of the grant pointers (the one corresponding to the grant that has been accepted) is moved and leaves the group. For the same reason, at most one new grant pointer can join the group. It is possible to prove that, after a transient period, all grant pointers point to different inputs, regardless of their initial position. A *maximum* matching is produced at every timeslot and 100% throughput is achieved. Desynchronization is preserved as long as all VOQs are non-empty, because all the released grants are accepted and so all the grant pointers move “in lockstep”.

Another important feature of i-SLIP is that it is fair and starvation free, i.e. it does not favor some flows over others and guarantees that a cell at the head of a VOQ will be served within finite time.

DRRM

DRRM [20] (Dual Round-Robin Matching) is a further variant of i-SLIP that achieves similar performance with one less phase and less information exchange between the input and the outputs.

The two phases performed in each iteration are:

1. *Request:* every unmatched input selects *one* unmatched output to request in round-robin order, starting from the position indicated by a *request pointer*. In the first iteration, the pointer is updated to one position beyond the input just requested (modulo N) if and only if a grant is received in the *grant* phase.

2. *Grant:* each output that has been requested by at least one input selects one to grant in round-robin order, starting from the position indicated by a *grant* pointer. In the first iteration the pointer is updated to one position past the input just granted (modulo N).

A grant phase is not required because each input requests only one output, so it can receive at most one grant, which is automatically accepted.

DRRM achieves 100% throughput under uniform i.i.d. traffic because in this situation request pointers (moved only if a grant is received) desynchronize.

Figure 2.7 shows the operation of the DRRM algorithm for a 4×4 switch. At the end of the first iteration all pointers (except R4 and G1) are moved forward by one position. As the matching is maximal, it is not necessary to perform additional iterations.

FIRM

FIRM [21] is an improvement of i-SLIP that achieves lower average latency by favoring FCFS order of arriving cells. It does so by introducing a minor modification in the pointer update rule of the grant phase of i-SLIP: *in the first iteration, if a grant is not accepted, the grant pointer is moved to the granted input.* The authors also show that this modification reduces the maximum waiting time for any request from $(N-1)^2 + N^2$ to N^2.

A similar modification has been proposed for DRRM in [22].

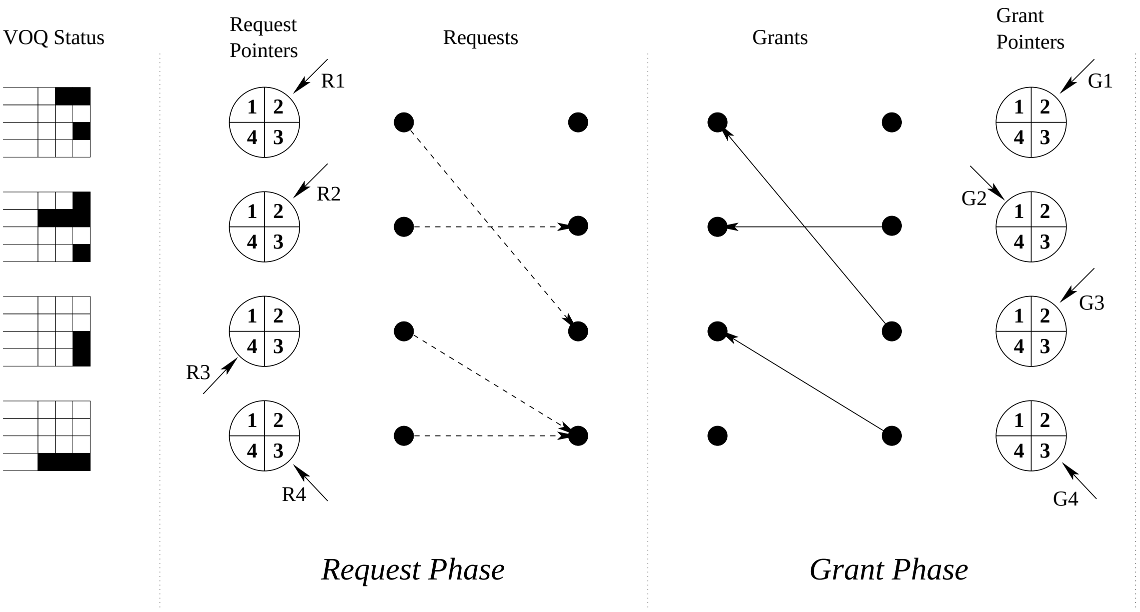


Figure 2.7. The behavior of the DRRM algorithm in a sample scenario.

Weighted Algorithms

As an attempt to approximate the behavior of MWM and improve performance under non-uniform traffic, heuristic iterative weighted algorithms have been developed. Among these are i-OCF (Oldest Cell First), i-LQF (Longest Queue First) and i-LPF (Longest Port First), proposed by Mekkittikul and McKeown [23].

2.5.3 Sequential Matching Algorithms

Sequential scheduling algorithms produce a maximal matching by letting each input add an edge at a time to an initially empty matching.

RPA [24] (Reservation with Pre-emption and Acknowledgement) and RRGS [25] (Round Robin Greedy Scheduler) are examples of sequential matching algorithms. An input receives a partial matching, adds an edge by selecting a free output and passes it on to the next input. Inputs considered first are favored, because they find most outputs still available. To avoid unfairness, the order in which inputs are considered is rotated at every timeslot. These algorithms always produce a maximal matching, are fair and can be pipelined to improve the matching rate. However, they require strong interaction among the inputs and introduce latency at low load when pipelined.

The Wavefront Arbiter [26] (WFA) is another popular sequential arbiter. The status of all the N^2 VOQs of the system is represented in a $N \times N$ request matrix R: $R_{i,j} = 1$

if input i has a cell destined to output j, 0 otherwise. Sets of VOQs that are positioned on a diagonal of the matrix are conflict-free, because they correspond to cells enqueued at different inputs and destined to different outputs. Hence it is possible to produce a matching by sequentially "sweeping" all the diagonals of the request matrix, excluding input and outputs that have already been matched. WFA is fast, simple and offers good performance; however, it suffers from some minor fairness and implementation issues [7].

2.6 Scheduling Multicast Traffic in IQ Switches

Traffic generated by a single source and directed to multiple destinations is called *multicast*. One-to-many communication is important for many applications (see Section 6.1) so switches must be able to efficiently replicate packets to multiple output ports.

In an IQ switch replication can be achieved simply by transmitting cells through the switching fabric multiple times, one for every egress adapter that must be reached. However, the crossbar has intrinsic multicasting capabilities and can replicate a cell to multiple outputs in a single timeslot. A scheduler that takes advantage of this feature can reduce the latency experienced by cells and the load on the fabric input ports, which are occupied for only one timeslot.

In this section we briefly introduce the problem of scheduling multicast traffic and present some of the most popular scheduling algorithms.

2.6.1 Definitions

The set of outputs a multicast cell is destined to is called the *fanout set* and its cardinality the *fanout* [3]. For clarity, we distinguish between the *input cell* that is transmitted to the switching fabric and the *output cells* that are generated by the replication process.

A scheduling discipline is termed *fanout splitting* if it allows partial service of an input cell, i.e. if the associated set of output cells can be transferred to the outputs over multiple timeslots. *No fanout splitting* disciplines, on the contrary, require all the output cells associated to an input cell to be delivered at the outputs in the same timeslot. Fanout splitting offers a clear advantage because it allows the fabric to deliver in every timeslot as many cells as possible to the outputs, at the price of a small increase of implementation complexity.

The *residue* is the set of all output cells that lose contention for output ports in a timeslot and have to be transmitted in subsequent timeslots.

[3] The term "fanout" is often used to refer also to the set itself.

2.6.2 Queueing

A multicast cell can be destined to any subset of the N outputs, so the number of possible fanout sets is $2^N - 1$. Even for moderate values of N it is not practically feasible to provide a dedicated queue to cells with the same fanout set, therefore HOL-blocking cannot be completely eliminated. Indeed, most architectures store cell arriving on an ingress adapter in a single queue and serve them in FIFO order.

To alleviate HOL-blocking, in [27] the authors propose a windowing scheme that allows the scheduler to access any cell in the first L positions of the queue. This scheme offers throughput improvements, but requires random-access queues, which are complex to implement. Moreover, it is clearly not effective under bursty traffic.

In [28] and [29] the benefits that can be gained by using a small number of FIFO queues at each ingress adapter are investigated. When multiple queues are present, it is necessary to define a queueing policy. Static queueing policies always enqueue cells with a given fanout in the same queue, whereas dynamic policies may enqueue them in different ones, depending on status parameters such as queue occupancy. Static policies lose effectiveness when few flows are active, because most of the available queues may remain empty, whereas dynamic policies lead to out-of-order delivery.

In [30] maximum switch performance is analyzed, under the hypothesis that a queue is provided for every possible fanout set. The results of this work have great theoretical interest, because they show that an IQ switch is not able to achieve 100% throughput under arbitrary traffic patterns, even if it employs this ideal queueing architecture and the optimal scheduling discipline.

2.6.3 Scheduling

The problem of scheduling multicast traffic in an input-queued switch has been addressed by a number of theoretical studies. In [31] and [32] the performance of various scheduling disciplines (such as random or oldest-cell-first) is analyzed under different assumptions. Work in [33] studies the optimal scheduling policy, obtaining it for switches of limited size (up to three inputs) and deriving some of its properties in the general case.

In [34] the authors take a more practical approach: they specifically target the design of efficient and implementable scheduling algorithms when FIFO queueing is used and fanout splitting allowed. They provide important insight on the problem and propose various solutions with different degrees of performance and complexity. An important observation is that at any timeslot, given a set of requests, all *non-idling* policies (those that serve as many outputs as possible) transmits cells to the same outputs and leave the same residue. What differentiates one policy from the other is residue distribution, i.e. the criteria with which the set of output cells that have lost contention is partitioned among the inputs. A *concentrating* policy assigns the residue to as few inputs as possible. Policies exhibiting this property serve in each timeslot as many HOL cells as possible, helping new cells to advance to the head of the queue. As new cells may be destined to idle outputs, throughput is increased. Actually a proof is given that for a $2 \times N$ switch a concentrating policy is optimal, but it cannot be extended to switches of arbitrary size.

The first proposed algorithm, called “Concentrate” implements a purely concentrating policy. However, the authors note that the algorithm suffers from fairness issues, as it can permanently starve queues, so they proceed with the design of TATRA, a concentrating algorithm with fairness guarantees. As TATRA is difficult to implement in hardware, they further propose the Weight Based Algorithm (WBA). WBA is a heuristic algorithm that approximates concentrating behavior by favoring cells with small fanout and guarantees fairness by giving priority to older cells. The algorithm works as follows:

1. At the beginning of every cell time each input calculates the *weight* of the cell at the head of its queue, based on the age of the cell (the older, the heavier) and the fanout (the larger, the lighter).

2. Each input submits a weighted request to all the outputs that it wishes to access.

3. Each output independently grants the input with the highest weight; ties are broken randomly.

In the specific implementation shown in the paper, the weight is calculated ad $W = \alpha A - \phi F$, where A is the age (expressed in number of timeslots), F is the fanout and α and ϕ are multiplication factors that allow tuning of the scheduler for performance or fairness. Large α implies that older cells are strongly favored, improving fairness, while large ϕ penalizes cells with large fanout, exalting the concentrating property and thus improving performance. Calculations show that a cell has to wait at the head of the queue for no longer than $(N(\phi/\alpha + 1) - 1)$ timeslots. WBA can be easily implemented in hardware, as reported in the paper.

Part I

A Switching Architecture for Shared-Memory Supercomputers

Chapter 3

Supercomputers and Interconnection Networks

In this chapter we present a brief overview of High-Performance Computing (HPC) systems, also called *supercomputers*. We first describe the main architectural traits of a supercomputer, including the organization of the computing nodes, the partitioning of the memory space and the programming model. We then focus on the *interconnection network* (sometimes simply called "the interconnect"), discuss its role in the system and analyze the main requirements. Finally, we introduce two fundamental classes of interconnection networks, highlight their most important features and show some sample topologies.

3.1 Supercomputing Systems

A *supercomputer* is "a computing system (hardware, system software and applications software) that provides close to the best currently achievable sustained performance on demanding computational problems" [35]. In the past the growth in demand for computing power has mainly been driven by scientific (weather forecasting, computational biology, plasma physics, etc.) and defense applications (cryptanalysis, stockpile stewardship, etc.). Nowadays business applications (automotive and aircraft design, geological analysis, modeling of financial markets, etc.) are also playing a role.

For almost two decades microprocessors have experienced a tremendous growth in performance, mainly due to technological improvements. Now the growth rate is slowing down, because of complicated issues such as power dissipation and difficulties in managing design complexity. Computer designers have traditionally tried to push the performance of computing systems by building parallel machines, in which multiple computing nodes work concurrently on portions of the same problem. In the near future we can expect parallelism to become the major source of performance improvement for all computing systems.

A large number of parallel computer architectures have been proposed over the years, varying considerably in terms of applications, programming model and intended system size. While it is difficult to provide a single, comprehensive taxonomy for this large and diverse set of architectures, some useful dichotomies for positioning and comparison of different systems have been established.

3.1.1 Shared-Memory vs. Message-Passing

In a shared-memory system all available memory can be accessed by all processors by means of a global address space. Processors exchange data and synchronize by accessing shared memory locations. Load/Store instructions issued by a processor are implicitly converted to Read/Write messages that the interconnection network delivers to the appropriate memory bank.

In a message-passing system, on the contrary, each processor has its own private memory space. Programmers explicitly exchange data and synchronization information among processors by invoking message passing primitives.

In general shared-memory systems are easier to program (at the operating system, compiler and application level) but more difficult to design than message-passing systems. On the other hand, the hardware simplicity of message-passing systems, especially the lack of complex cache-coherency issues, makes them much more scalable. For this reason, the majority of *Massively Parallel Processing* (MPP) systems, having thousands or even hundreds of thousands of processors, are message-passing machines.

3.1.2 UMA vs. NUMA

In a shared-memory machine memory can be logically placed in a single centralized location or distributed over the computing nodes, co-located with the processors. In the first case memory access time is *uniform*, i.e. it does not depend on which processor accesses which memory location. In the other, it is *non-uniform*, because a processor experiences lower access time when accessing a memory location in its local bank rather than in a remote one. Machines providing uniform or non-uniform memory access are classified as UMA and NUMA, respectively.

Typical UMA systems are SMP (Symmetric Multiprocessor) machines, in which a small number of processors (few tens at most) and a single bank of memory are connected by means of a simple interconnection (usually a shared bus), as shown in Figure 3.1. Examples of NUMA systems are DSM (Distributed Shared-Memory) machines, which comprise hundreds of computing nodes (composed by a processor and a memory bank) interconnected through a high-speed network (Figure 3.2).

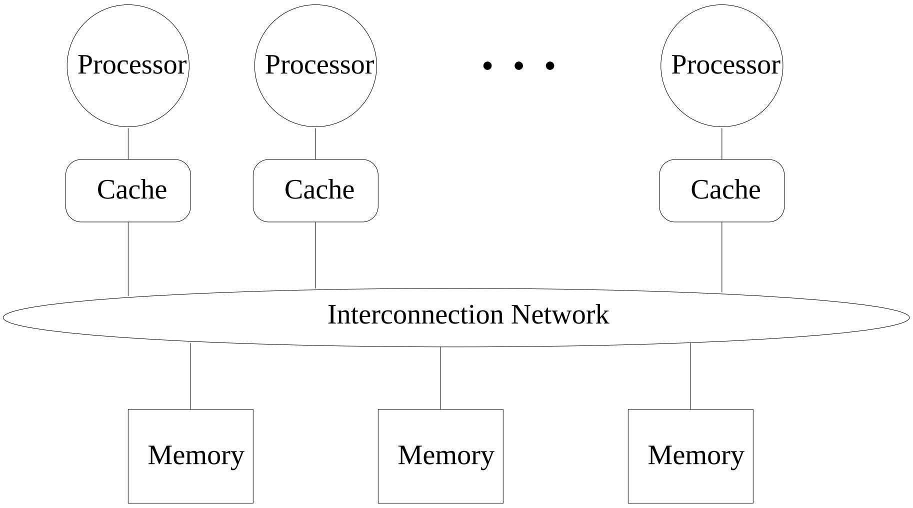


Figure 3.1. An SMP machine

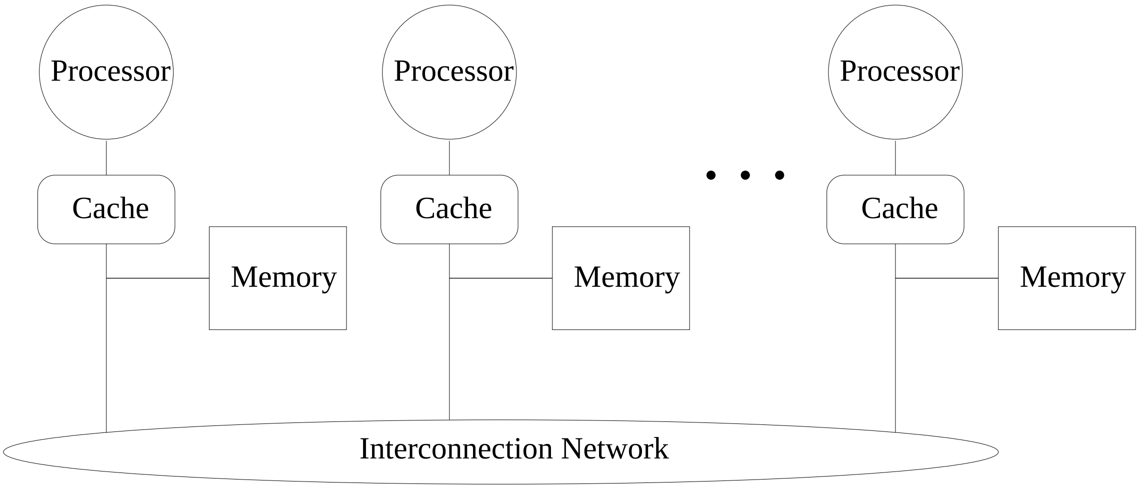


Figure 3.2. A DSM machine

3.1.3 Custom vs. Off-the-shelf

Various platforms use different blends of custom and commercial, off-the shelf (COTS) components. COTS components are designed for a broad range of applications and are produced in large quantities. They benefit from economies of scale and offer very good cost/performance ratios. On the other hand, their design is not optimized for supercomputing and they might perform poorly on some specific applications.

Microprocessors

The cost of designing and manufacturing a new processor has grown steadily over the years and nowadays only few companies can afford it. For this reason, most supercomputers today use commodity processors produced for the large-volume server and workstation markets.

Interconnects

The interconnection network, on the contrary, is more difficult to build with commodity components. The gap between the requirements of a local area network and those of a supercomputer interconnect is quite large. Although the bandwidth offered by Ethernet has increased by several orders of magnitude during its lifetime, its application domain is still limited by its inability to achieve low latency and guarantee lossless behavior.

New standard-based technologies, Infiniband [36] in particular, are trying to fill the gap and provide a unified network infrastructure for local area networking and parallel computing. Infiniband has a number of features specifically aimed at reducing network latency. It employs an improved node/network interface that allows the network adapter to connect directly to the memory controller of the node, bypassing the I/O bus. Moreover, it supports advanced communication paradigms, such as RDMA, that allow a node to move data directly in and out the memory space of another node. A carefully designed flow-control mechanism enables loss-free operation and allows prioritization of latency-sensitive messages. Altogether this characteristics make Infiniband a potential alternative to custom interconnection networks.

Clusters

Many of the largest supercomputers available today are *cluster*, i.e. collections of standard servers and workstations, loosely connected through standard LAN interconnects such as Gigabit Ethernet. As clusters are entirely composed by commodity components, they offer excellent cost/performance ratios. The use of standard interconnects promotes scalability, while the fact that each node has its own processor, memory and operating system provides significant advantages in terms of reliability and fault-tolerance [37].

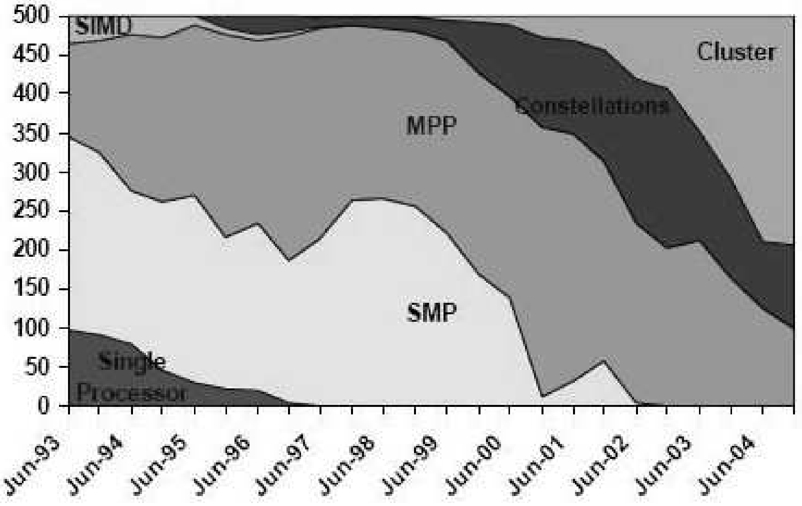


Figure 3.3. Time evolution of the architecture of the 500 most powerful supercomputers in the world (from `http://www.top500.org`).

Clusters performance is mainly limited by the latency introduced by the network, which makes them unfit for applications that require strong interaction among computing nodes. On the other hand, popular applications such as web servers and databases are particularly amenable to run on clusters, because they are characterized by a large number of independent threads that work in parallel, so they are not penalized by network latency. For example, in [38], the authors describe the Google cluster, built with standard PCs and comprising more than 6000 processors (as of December 2000).

Figure 3.3 shows the distribution of the 500 most powerful supercomputers in the world, based on their architecture. The current list is clearly dominated by clusters and constellations (clusters of SMP systems) that together account for 80% of the total. The remaining 20% is represented by custom MPP systems.

3.2 Interconnection Networks

The interconnection network is a critical component of a supercomputer, because it has a direct impact on performance and scalability. The variety of node architectures, programming models and application requirements has generated a proliferation of interconnection network designs, ranging from single shared busses used in SMP systems to complex, meshed fabrics with thousands of ports for MPP systems.

The key requirements of a supercomputer interconnect are:

- **Low latency** Latency is the time required for a packet to traverse the network.

It is the most important performance metric of an HPC interconnect, especially when considering shared-memory machines. In such systems communication is triggered by memory access instructions and the latency introduced by the network directly contributes to memory access time. Specialized processors use latency-hiding techniques, such as fetching data from memory in advance [39], fetching more than necessary and allowing multiple outstanding memory references. Despite the availability of these techniques, network latency remains the primary performance bottleneck for a number of applications [35].

- **High Throughput** Throughput is a measure of the *rate* at which the network can deliver data to the nodes. High throughput corresponds to high utilization of link bandwidth and is particularly important when the nodes have to exchange bulk sets of data. Latency-hiding techniques mentioned above tend to transfer large blocks of data, thus increasing throughput requirements.

- **Scalability** The network must be able to interconnect a large number of computing nodes. Moreover, as the number of nodes grows, the aggregate bandwidth of the network should increase proportionally and latency should remain low. Network scalability is essential to guarantee that the computing capacity of the system reaches the intended levels and improves as new nodes are added.

- **Reliability and Fault Tolerance** A supercomputer uses a large number of components and, as a consequence, the failure rate can be high. The network should be able to continue operation in presence of a limited number of faults. In particular, it should be able to exploit meshed connectivity and re-route messages over alternative paths in case of link or node failure.

Interconnection networks can be characterized in terms of *topology*, *routing* and *flow-control*. Topology describes the interconnection pattern among nodes, routing determines paths between pairs of non-adjacent nodes and flow-control defines mechanisms to regulate message transmission among nodes and prevent network overloading. Selecting the topology is usually the first step in designing the network, because routing and flow-control are heavily dependent on its characteristics. The choice of the topology is mainly driven by the constraints imposed by the available packaging technology [7].

In the remaining part of this section we briefly describe two important classes of interconnection networks and show some popular topologies. A comprehensive classification can be found in [40].

3.2.1 Direct networks

The distinguishing property of direct networks is that each node is directly connected to a small set of other nodes by means of bi-directional, point-to-point links. Communication between non-neighboring devices entails transmission through intermediate hops. Each node has an integrated *router* that handles communications, transmitting and receiving messages or relaying them to other nodes.

Popular topologies for direct networks are n-dimensional meshes, tori and hypercubes (Figure 3.4). The tree (Figure 3.5) is another important topology, because it efficiently supports one-to-many and many-to-one communication patterns, typical of synchronization and *collective* operations that require coordination of many computing nodes [40].

Direct networks scale very well in terms of bandwidth, so they have been used extensively in MPP systems. However, as the number of nodes increase, so does the distance between pairs, thus latency degrades.

Very large systems can use multiple networks optimized for specific tasks. For example, the IBM Blue Gene/L, capable of scaling up to 65535 computing nodes, uses a 3D-torus as a general-purpose interconnect and two specialized tree-like networks for synchronization and collectives [41, 42].

3.2.2 Indirect networks and MINs

Indirect networks interconnect computing nodes through intermediate nodes called *switches* (Chapter 2). Switches receive messages on input ports and forward them to the appropriate output ports, towards the final destination.

The complexity of a switch typically grows quadratically with the number of ports, so its scalability is limited to few hundred ports at most (Section 2.3). As a single switch cannot satisfy the requirements of large supercomputers, we must turn to *Multistage Interconnection Networks (MINs)*. MINs enable the construction of fabrics interconnecting thousands of nodes by employing several switches arranged in multiple *stages*. The number of stages and the interconnection pattern between the switches define the topology of the network.

MINs were originally studied for circuit-switched networks and later employed in packet-switched networks. Among the most popular topologies are Clos [43, 44], Butterflies [45] and Fat-trees [46] networks.

A MIN is *unidirectional* if data can flow on network links in a single directional, *bi-directional* if it can flow simultaneously in both directions. For computer interconnects, bi-directional MINs are usually preferred, because they offer shorter paths between nodes (messages traverse only as many stages as necessary) and better redundancy. Figures 3.6 and 3.7 show a bi-directional Butterfly and a Fat-Tree network (circles represent nodes and boxes represent switches).

MINs have very good scalability properties because the aggregate bandwidth grows as new switches are added to the network and latency remains low thanks to small number of stages. However, cost also increases rapidly, because more and more switch ports are used to connect to other switches rather than computing nodes.

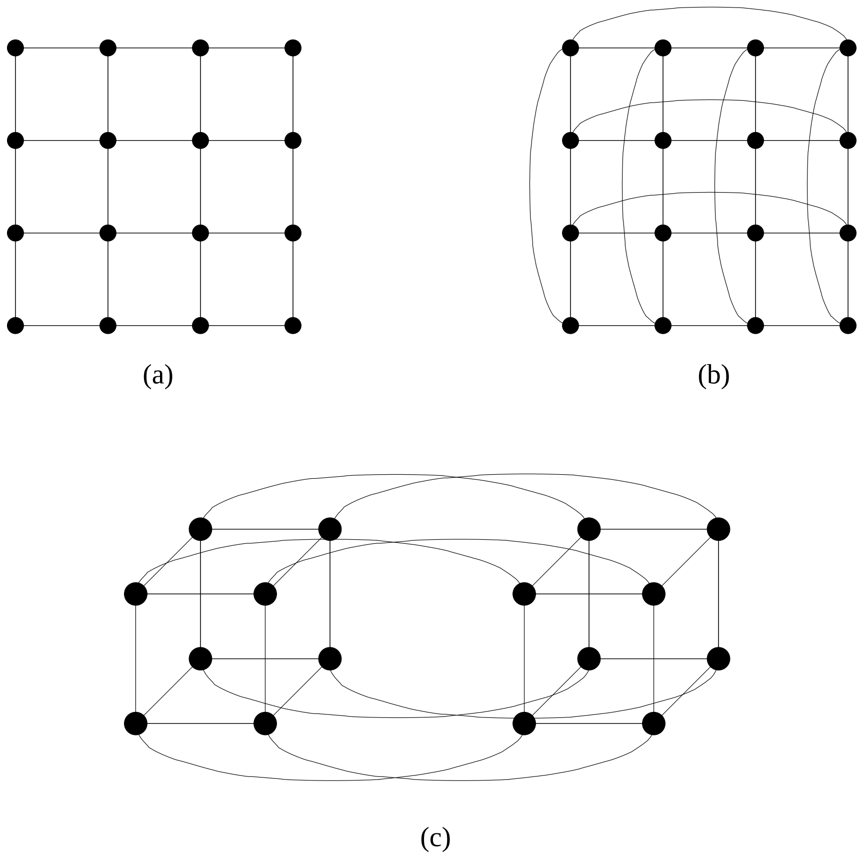


Figure 3.4. Direct network topologies: (a) 2D-mesh, (b) 2D-Torus, (c) Hypercube

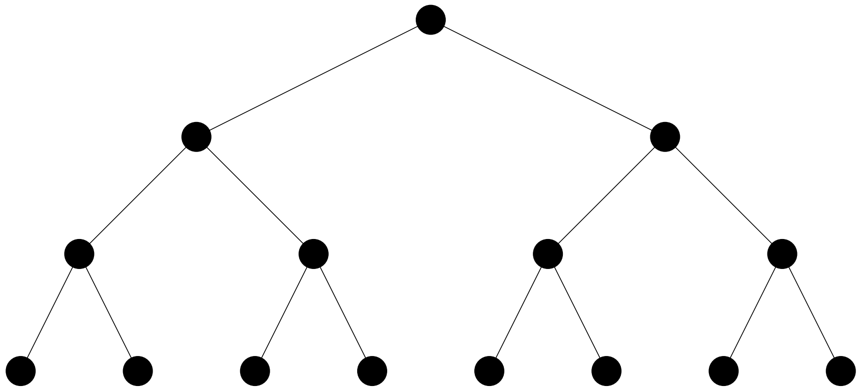


Figure 3.5. A 15-nodes binary tree topology

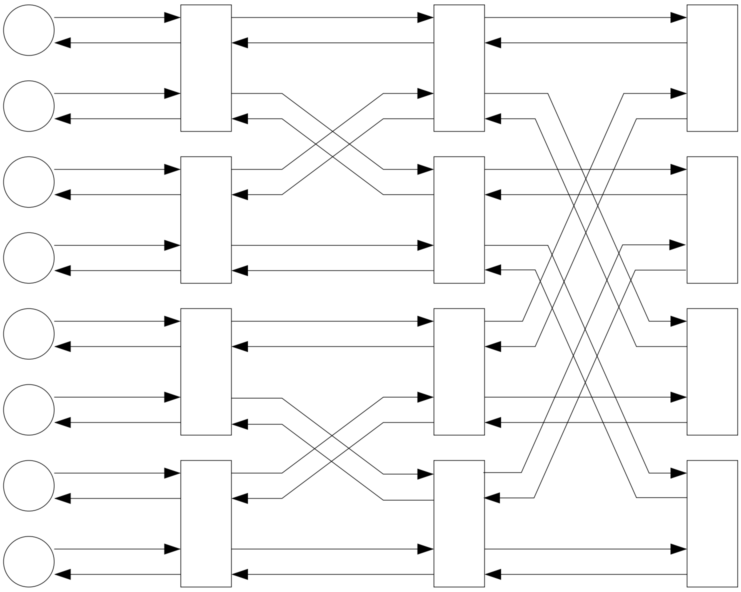


Figure 3.6. An 8-nodes bi-directional butterfly network

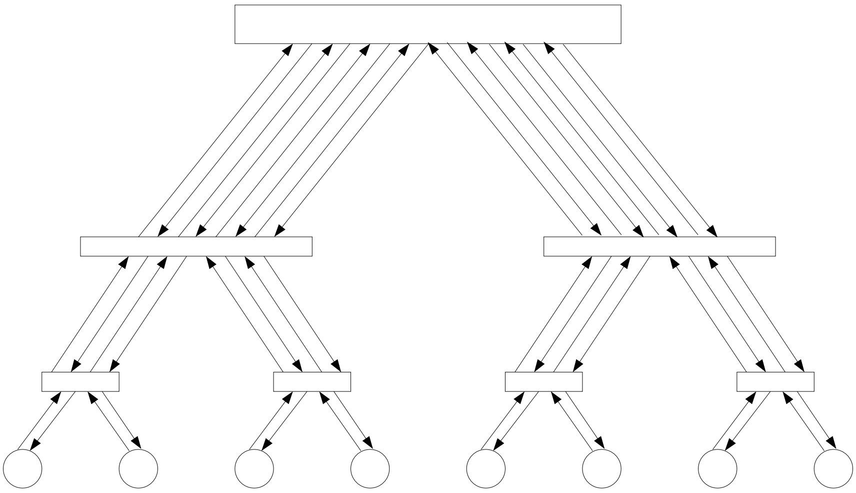


Figure 3.7. An 8-nodes fat-tree network

Chapter 4

The OSMOSIS Project

In this chapter we present the OSMOSIS project, aimed at developing a prototype of a switch for shared-memory supercomputers. We start with a digression on the role of electronics and optics in packet switching, to explain the rationale behind the choice of an electro-optical architecture and compare it briefly with other optical switching architectures. We then provide an overview of the system, discussing the set of requirements and the design of the the data- and control-path.

4.1 Electronics and optics in packet switching

4.1.1 Electronic switching

The performance of electronic packet switches has grown tremendously in the last fifteen years, driven by the exponentially-increasing bandwidth requirements of Internet traffic. Electronic switching is now a mature technology that has been employed in a number of other domains, including HPC systems. However, the tough latency and scalability requirements of HPC interconnects are pushing it to the limit and are exposing its weakest points.

The major problem that plagues electronic switching today is power consumption. As the line rates increase, it becomes more and more difficult to drive copper cables over acceptable distances. It is currently estimated that as much as 50% of the power consumed by a switch is actually spent on the cables [14]. The problem can be addressed by using optical fibers for transmission on the links and electronic components for buffering, switching and processing. This solution, however, is only partially satisfactory, because the O/E/O conversions required at the ingress and egress side of the switch consume power, increase the cost of the devices and introduce latency. It would be highly desirable to switch packets in the optical domain, avoiding conversion and reduce latency to the time-of-flight of signals in the fibers.

4.1.2 Optical devices

Optical devices have a number of unique features that distinguish them from electronic ones and make them extremely attractive. First, a single fiber link, thanks to DWDM techniques, can offer bandwidths on the order of terabits per second, several orders of magnitude larger than what is provided by electrical links. Second, optical links can span very long distances using limited power, so they are particularly fit for large and distributed computing systems, whose diameters can be in the order of tens or hundreds of meters. Last, and probably foremost, many optical devices are data-rate transparent, meaning that they have extremely large operational bandwidths and can operate on signals (split, combine, amplify, etc.) at constant power, regardless of the frequency at which they are modulated. In the electronic domain, on the contrary, devices can only operate in specific frequency ranges and power consumption is proportional to frequency. Thanks to these features, an all-optical data-path can scale in bandwidth by orders of magnitude, without increasing the physical size or the power consumption of the network elements.

The development of optical switches has mainly been limited by factors such as device cost, integrability and noise levels. Moreover, the lack of optical buffers and logic elements are two fundamental issues that haven't been addressed satisfactorily yet. However, it is a common opinion that economic and technological issues can be solved in short timeframes. If optical devices get market acceptance, their cost will decrease and the manufacturing process will improve, leading to higher quality and integration levels. As Moore's Law, that governs density as well as cost of electronic components, is slowing down, projections show that optical switches could be economically competitive by the end of the decade [47].

4.1.3 Optical switching architectures

Over the years many optical switching architectures have been proposed [48]. Many of them, however, are conceived for circuit-switching networks, so they exploit physical phenomena that enable switching times in the order of milliseconds.

For packet-switching networks, the switching time must be smaller than the duration of a minimum-size packet. Given the line-rates and packet sizes we are targeting, this translates to few nanoseconds. Optical Burst Switching (OBS) techniques mitigate this challenging requirements by fitting multiple packets in large containers and switching them together at once. These techniques are not suitable for supercomputing applications because packets must wait for a container to be full before being switched, so they experience additional latency.

Semiconductor Optical Amplifiers (SOAs) are the most promising technology for optical packet switching. They can be viewed as ON/OFF optical switching elements, with very low switching times (on the order of few nanoseconds), high extinction ratios and low noise. They are compact, consume low power and can be integrated into arrays [49].

SOAs can be used to build switching nodes ranging from simple 2×2 switches to large crossbars using broadcast-and-select networks [50].

Even with appropriate technology, the question remains on how to build a packet switch without large buffers and bit-level processing capabilities. A possible approach is to use electronics for buffering and control at the borders of the fabric, confining optics to the data-path. The basic philosophy behind such hybrid opto-electronic architectures is to use optics for what optics does best and electronics for what electronics does best [51].

An alternative, pursued by the Data Vortex project [52], is to eliminate the need for buffers altogether by using deflection routing and a very simple node structure that enables distributed control with minimal processing capabilities. The Data Vortex aims at fully exploiting the benefits of optical technologies and being a true all-optical switch. It has a number of desirable features that make it very attractive for supercomputing applications, first and foremost scalability. However, it also has some non-negligible drawbacks, mainly low throughput per port, out-of-order delivery and hard-to-predict latency.

4.2 The OSMOSIS System

4.2.1 Goals and requirements

OSMOSIS (Optical Shared-MemOry Supercomputer Interconnect System) is a research project jointly developed by IBM and Corning that aims at building an HPC switch with an all-optical data path and an optimized electronic control path [53].

The goals of the project are twofold: on one side, it aims at solving the technical challenges involved in building a demonstrator system that meets a set of ambitious requirements, on the other it aims at accelerating the cost reduction of all-optical switches, achieving denser integration levels of optical components and finding a high volume market for them, in addition to the low volume HPC market.

The specific requirements for the demonstrator are:

Port count	64 (single stage) – 2048 (multistage)
Line rate	40 Gb/s (scalable to 160 Gb/s)
Total (application to application) latency	$< 1\mu$s
Effective user bandwidth	$> 75\%$ of raw transmission bandwidth
Bit Error Rate (BER)	$< 10^{-21}$
Packet delivery	Reliable and in-order

In addition, efficient support for mutlicast and broadcast is a basic requirement, as they are particularly important for HPC applications [54]. All electronic control logic must be implemented using only FPGAs and commercial components, to gain flexibility and keep the cost of the demonstrator acceptable.

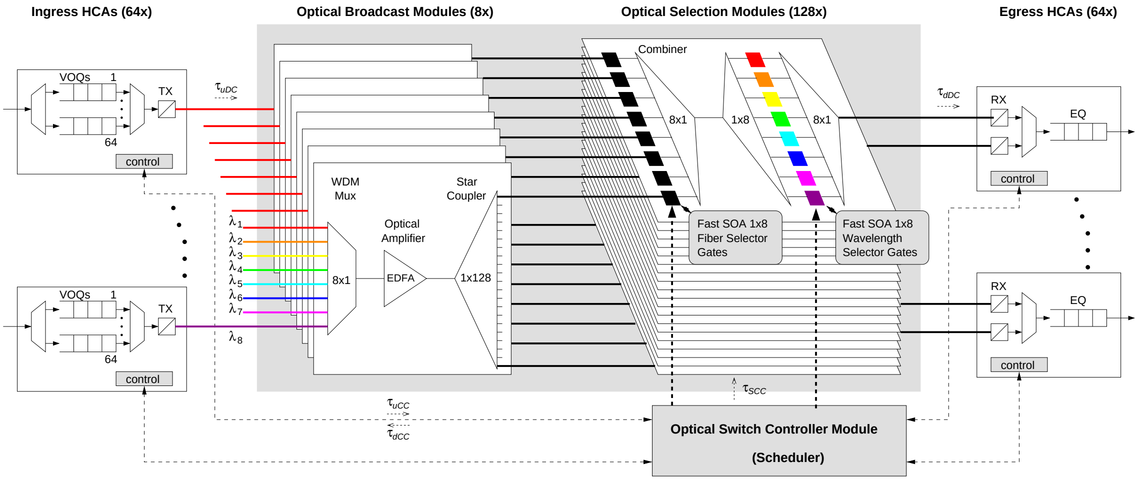


Figure 4.1. OSMOSIS system architecture.

4.2.2 System Overview

From an architectural point of view, OSMOSIS is a synchronous CIOQ switch (Chapter 2). This implies that the switch operates on fixed-size cells and the optical core, functionally equivalent to a crossbar, is reconfigured on a cell-by-cell basis. The cell size derived from the requirements set is 256 B [55]. In general shorter cells would be desirable, as they would offer lower latency and improve efficiency. However, this cell size is acceptable and well-suited for shared-memory supercomputing, as synchronization messages and cache-coherency transactions usually comprise a 100-300 B payload.

Figure 4.1 shows the high-level architecture of OSMOSIS in single-stage configuration. The system is composed by a set of *adapters* interconnected by an *optical core* controlled by a centralized *scheduler*.

Adapters have two separated but physically co-located parts, named *ingress* and *egress* adapter, that handle cells entering and exiting the switch respectively. They perform interfacing functions between the computing nodes and the interconnect, including E/O and O/E conversions, cells buffering and processing. Every ingress adapter comprises a full set of N VOQs, in which it stores cells based on their destination and a corresponding set of N reliable delivery queues (RDQs). Egress adapter host an *egress queue* where they buffer received cells before delivering them to the computing node or transmitting them to the next stage. Every adapter has a dedicated optical control link to the centralized scheduler, which carries the control channel protocols.

The scheduler is located on a separate card close to the optical switching core. At the beginning of every timeslot it receives requests to transmit from the ingress adapters, resolves conflicts and grants selected adapters, authorizing them to transmit.

4.2.3 Data path

Every adapter is assigned a specific wavelength λ_i, chosen among eight possible wavelengths. The 64 adapters are logically divided in eight groups, in such a way that all the adapters in the same group use a different wavelength.

The switching function is implemented with a *Broadcast & Select* architecture that combines eight-way space- and eight-way wavelength-division multiplexing to implement a 64-port fabric. The first stage of the optical core consists of eight *broadcast units* that receive the signals transmitted by a group of adapters (consisting of eight signals on eight different wavelengths), multiplex them on a single fiber, amplify the resulting WDM signal using an erbium-doped fiber amplifier (EDFA) and then split it to 128 waveguides[1]. The *select* stage comprises 128 planes, two per output port, each connected to all the eight broadcast units of the first stage. A group of eight SOAs performs fiber selection by blocking signals coming from all the broadcast units except one. Regardless of the selected fiber, the WDM signal is passed through a combiner to guide it to a common fiber and then de-multiplexed to separate the eight wavelengths on different fibers. A second group of SOAs is used to select a single wavelength and block the others. The signal is again guided to a common fiber by means of a combiner and finally reaches the egress adapter. Each egress adapter is connected to two select units, so it can independently receive two signals at the same time. Multicast and broadcast transmission can be achieved simply by having multiple planes select the same fiber/color pair.

The optical core employs a combination of Planar Lightwave Circuits (PLCs) and discrete components. Transmission from the ingress adapters must be synchronized in such a way that cells arrive at the optical crossbar at the same time, when SOAs have just been configured, and walk equal-length paths. This is achieved by using a global clock signal distributed to all the adapters and components whose length is matched to a fraction of the optical packet length.

Although all ports work at the same nominal bit-rate, egress adapters receive bit-streams generated by different serializers, with independent phases. Thus receivers must operate in burst-mode and, to keep cell overhead low, they must be able to recover bit-phase in a very short time. Moreover, channels use different wavelengths, so receivers must have wide-dynamic-range transimpedance.

4.2.4 Control path

The physical implementation and packaging constraints of OSMOSIS, and of large switches in general [14], lead to a distribution of switch components (adapters, optical core, scheduler) over multiple racks, interconnected by long cables. The latency introduced by these cables, together with the delay due to (de-)serialization and other contributions, add up

[1] 128-way splitting (rather than 64-way) is required to allow each egress adapter to receive up to two cells per timeslot. The details about the usage of the second receiver can be found in [54]

to several cell times and must be carefully taken into account in the design of the control path.

Further details about the architecture of the OSMOSIS scheduler and the design of the control plane can be found in [54].

Control Channel Protocol

As the scheduler is connected to the adapters through long cables, control messages have to be pipelined. The scheduler has delayed knowledge of the status of the VOQs at the ingress adapters, an issue that can seriously degrade performance. The problem is addressed by maintaining VOQ status information at the scheduler, updating it using an *incremental* protocol [56] and ensuring its consistency using a *census* mechanism [57].

The scheduler maintains N^2 counters, each representing the occupancy of a VOQ. When a new cell arrives at an adapter, the scheduler is notified by a control message and increments the corresponding counter. When the scheduler issues a grant, it decrements the counter, as the cell will be dequeued as soon as the grant arrives at the adapter.

The census mechanism is a distributed consistency protocol capable of detecting and correcting discrepancies between the information maintained at the scheduler and the known status of the VOQs on the adapters. It is triggered at regular intervals to ensure proper recovery from control channel transmission errors.

Scheduler

The scheduler must solve a bipartite-graph matching problem at every timeslot (Section 2.4.3). As the optimal scheduling algorithms is not implementable in fast hardware, the scheduler uses a heuristic iterative algorithm based on DRRM (Section 2.5.2 and [20]). Iterative algorithms are implemented using $2N$ programmable priority encoders, that perform 1-out-of-N selection [28]; as N increases, so does their space and time complexity. Moreover, they need to perform $\log_2 N$ iterations to produce a good matching and achieve high delay-throughput performance.

Given that N is large and that only FPGAs are at disposal, implementing the scheduling algorithm and performing the desired number of iterations entails a number of challenges. First, the scheduler must be distributed over multiple chips, as it doesn't fit on a single one. Distribution requires the usage of specific techniques to deal with delays and bandwidth limitations of in chip-to-chip communication. These techniques are part of the contributions of this work and are discussed in detail in Chapter 5 and [58]. Second, it is not possible to perform the desired number of iterations in the short duration of a timeslot. To overcome the problem, the scheduler employs a pipelining scheme, called FLPPR (Fast Low-latency Parallel Pipelined aRbitration) [59]. The most important feature of FLPPR, that distinguishes it from previously proposed pipelining schemes [60,61] is that new requests are allowed to enter at any stage of the pipeline, reducing the minimum latency to a single timeslot. FLPPR also achieves better performance at high load under non-uniform and bursty traffic.

The scheduler also implements a novel scheme to achieve fair and efficient integrated scheduling of unicast and multicast traffic. The scheme is described in Chapter 6 and [62].

Reliable Delivery

The optical data path is engineered to achieve a raw bit-error rate of 10^{-10}. A custom forward error-correcting code (FEC) is employed to reduce it to 10^{-17}. Cells that cannot be corrected by the FEC must be retransmitted, hence each adapter has a set of reliable-delivery queues (RDQs) where packets are stored until the egress adapter acknowledges reception. ACKs are transmitted on the control channel and a Go-Back-N retransmission policy is used. This policy is simple to implement and is consistent with the bursty nature of optical link errors. As the error rate provided by the FEC is already fairly low, retransmission is rarely required and the inefficiencies of Go-Back-N are not an issue.

Flow Control

To prevent overflow of the egress buffers, the system employ an on-off flow control loop between egress adapters and the scheduler, which is embedded in the upstream control channel messages. If a specific egress buffer is close to saturation, the scheduler no longer considers any request for the corresponding output. When occupancy decreases below a pre-determined threshold, the permission is reinstated.

4.2.5 Multistage scalability

One of the main objectives of the OSMOSIS project is to provide a system that can scale to thousands of nodes. In single-stage configuration this is not practically feasible due to the quadratic complexity of the scheduler and the optical core. A viable solution is to scale the number of ports is to use a multistage network. Based on the considerations exposed in Section 3.2.2, the Fat-Tree topology has been selected. Using 96 64×64 switches the fabric scales to 2048 ports with full bi-sectional bandwidth (Figure 4.2).

Having an end-to-end all-optical data path, without intermediate electronic buffers, would be very attractive, because it would reduce cost, power consumption and latency. However, it would require a centralized scheduler capable of configuring all the switches in the network simultaneously. The complexity of such scheduler would be unbearable given the timing constraints. Moreover, the delay introduced by the control channel would become much larger. As the adapter must send a request and wait for a grant before transmitting a cell, much of the latency advantage would be defeated [51].

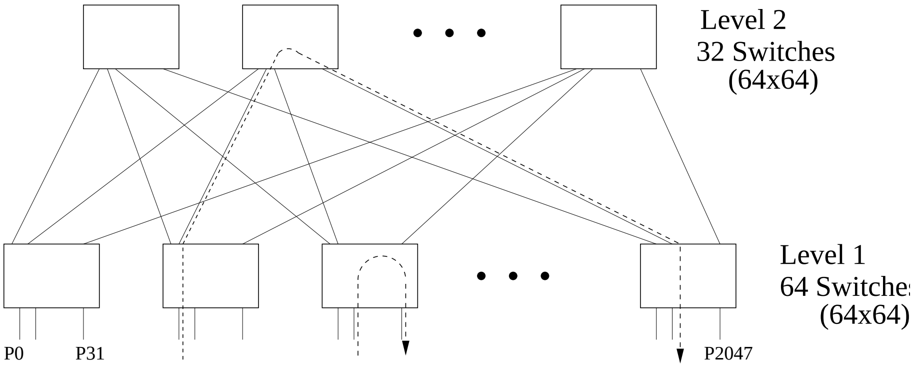


Figure 4.2. Multistage Fat-Tree configuration with 2048 ports and full bi-sectional bandwidth. All links are bi-directional. A single- and a multi-hop paths are shown.

Chapter 5

Distributed Implementation of Crossbar Schedulers

Despite extremely high-density CMOS technologies, as the number of ports N grows, the complexity of crossbar schedulers based on parallel iterative algorithms such as i-SLIP and DRRM quickly exceeds single-chip implementation limits. The implementation is limited by power density, gate count, pin count, I/O bandwidth and wiring, due to the high degree of connectivity between the input and output selectors [14]. In this chapter we present a set of techniques that enable distributed (multi-chip) implementations of iterative matching algorithms, enabling the construction of schedulers for large switches, while achieving a level of performance that is close to that of a monolithic (single-chip) implementation.

The practical motivation for this effort is the design and implementation of the OSMOSIS arbiter, as described in Chapter 4. Sizing experiments show that the scheduler logic must be distributed over multiple devices, which introduces a number of new challenges. Most importantly, the physical distances among chips introduces latencies that exceed the timing requirements and the separation of logical units prevents shared access to status information.

We consider four levels of distribution, from monolithic to fully distributed, and present a number of techniques to mitigate the effects of specific distribution levels. The performance results obtained via simulation show that, using these methods, a distributed scheduler can achieve performance close to that of a monolithic one, even with large internal latencies.

5.1 Iterative Matching Algorithms

Iterative matching algorithms such as i-SLIP [19], FIRM [21], and DRRM [22] are widely used owing to the key advantages they offer:

1. High performance: More precisely, they guarantee 100% throughput under uniform uncorrelated traffic with a single iteration. Additional iterations significantly reduce the mean latency.

2. Fairness: They ensure that under any traffic pattern any nonempty VOQ receives service within finite time.

3. Practicality: Although, a total of $2N$ selectors (one per input and one per output) is required, these selectors operate independently and in parallel. Thus, high matching rates can be achieved. Moreover, the selectors are feasible to implement in fast hardware [28].

5.1.1 Two- vs. three-phase algorithms

Iterative matching algorithms can be classified into two- and three-phase algorithms according to the number of steps per iteration. In three-phase algorithms, there are *request*, *grant*, and *accept* steps in every iteration. In the request phase, every input sends a request to *every* output it has at least one cell for. In the grant phase, every output independently selects one request to grant. As these decisions are independent, multiple outputs may grant the same input. Therefore, in the third phase, every input selects one grant to accept. Two-phase algorithms, on the other hand, comprise only a request and a grant phase. In the request phase, every input sends a request to *one* output for which it has at least one cell. In the grant phase, every output independently selects a request to grant. Because every input can receive one grant at maximum, there is no need for an accept phase, i.e., every grant is automatically accepted. i-SLIP and FIRM are three-phase algorithms, whereas DRRM is a two-phase one.

Input and output selection is based on a prioritized round-robin mechanism, i.e., the input (output) selector chooses the first eligible output (input) starting from the position indicated by a *pointer*. The pointer update policy is a crucial characteristic of each algorithm and must be chosen carefully to guarantee performance and fairness. The update policies employed by these algorithms share a common trait: once a connection (corresponding to a VOQ) becomes highest priority, it will be given precedence over the other competing ones until it is established. In i-SLIP this is achieved by having an output grant the same input (in the first iteration) until the grant is accepted. In DRRM, on the contrary, an input will keep requesting the same output (in the first iteration) until it receives a grant. This feature guarantees fairness and leads to *pointer desynchronization* [18], i.e., it assures that under heavy traffic (when all the VOQs are nonempty) each output grants a different input (i-SLIP) or each input requests a different output (DRRM). When this happens, there are no conflicts and a maximum-size matching is produced in every timeslot, leading to 100% throughput.

N	device utilization			performance		#iterations	
	slices	%	#nets	$f_{\max}$ (MHz)	$t_{\min}$ (ns)	I_a	I_t
iSLIP (in- and output selectors)							
4	266	0.60	2,075	203.9	4.90	10.4	2
8	1,071	2.43	8,008	119.1	8.39	6.1	3
16	4,544	10.3	33,770	79.9	12.51	4.1	4
32	15,046	34.1	114,987	57.9	17.27	3.0	5
48	34,652	78.6	264,174	42.4	23.58	2.2	5.6
52	41,437	93.8	316,856	44.1	22.66	2.3	5.7
64	does not fit FPGA device			–	–	–	6

Table 5.1. Sizing in Xilinx Virtex-II-Pro (speed grade -6), from [55].

5.1.2 Sizing experiments

This study is motivated by the implementation of the 64×64 crossbar scheduler for OSMOSIS. One of the challenges in this project is to implement a scheduler of this complexity in FPGA technology, which is used mainly for reasons of cost and flexibility.

Our sizing results, shown in Table 5.1 (also previously reported in [55]), demonstrate that a monolithic implementation does not fit in the targeted FPGA device, which is the biggest and fastest FPGA available from Xilinx at the time of implementation, namely, the “xc2vp100-6ff1704,” a Virtex-II-Pro series FPGA providing 8 M system gates (100 K logic cells)[1] and 1040 users I/Os. The table lists the device utilization in the number of slices, percentage and number of nets, the scheduler performance in terms of maximum clock frequency $f_{\max}$ and minimum clock period $t_{\min}$, and the number of achievable iterations I_a vs. the target $I_t = \log_2(N)$. The numbers refer to the unconstrained placement and routing of the request-grant-accept phases of i-SLIP, based on the implementation described in [28], considering only the core of the algorithm, without the I/O interfaces required to convey the external requests and grants to/from the scheduler device.

As the largest scheduler feasible in a single Virtex-II-Pro xc2vp100 is somewhere in the range of 52×52, we cannot use a monolithic matching algorithm for our centralized crossbar scheduler. Clearly, these hold for three- as well as two-phase algorithms, as the latter are not inherently less complex in terms of silicon area.

Additional experiments show that it is possible to place *the N output selectors* of a 64×64 DRRM scheduler, together with the I/O logic and the status and configuration registers on a single chip. This result is important because it will be used for the design of our specific scheduler.

[1] Virtex logic cell = (1) 4-input LUT + (1) flip-flop + carry logic. Virtex slice = 2 logic cells.

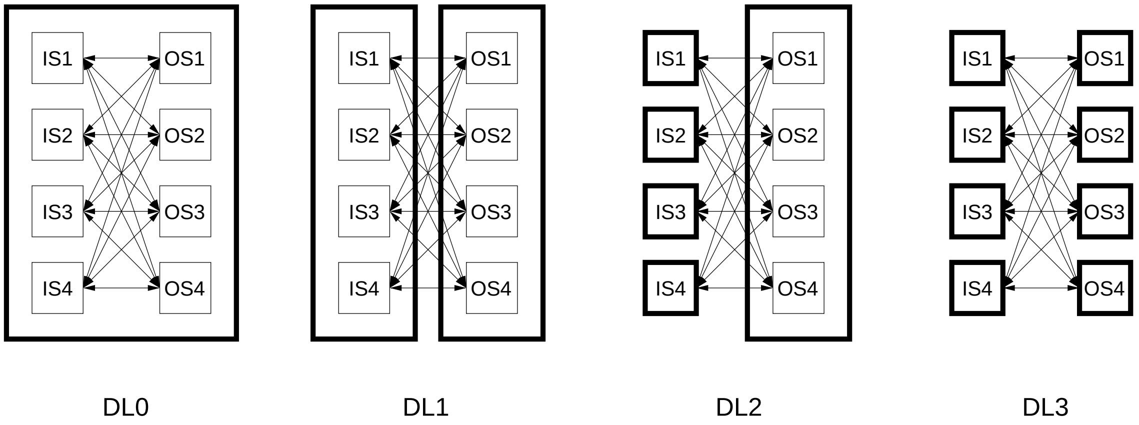


Figure 5.1. Schematic representation of the four levels of distribution. The bold lines represent the device (chip) boundaries. IS = input selector, OS = output selector.

5.2 Distribution Challenges

The sizing results above call for a distributed implementation, which entails partitioning the selectors over multiple physical devices. In a monolithic implementation, the selectors are tightly coupled and decisions taken at the inputs are known to the outputs (and vice versa) within the same timeslot. In a distributed implementation, this is no longer true. The delays caused by wires, (de-)serialization and pin-sharing can add up to several timeslots. This means that critical information needed to update pointers or issue new requests is not available in a timely fashion. As mentioned previously, pointer update is crucial for iterative algorithms, because it is the key to performance and fairness, so care must be taken not to disrupt it.

For ease of reference, we introduce and discuss four levels of distribution (DL0 through DL3), as illustrated in Figure 5.1:

DL0 Monolithic implementation: All input and output selectors are implemented in a single device. The implicit assumption is that the result of every iteration is known globally before the next iteration is executed.

DL1 Separate the input from the output selectors, creating two groups of N selectors each, enabling distribution over two devices.

DL2 Additionally separate the input selectors from each other, enabling distribution over $N + 1$ devices.

DL3 Additionally separate the output selectors from each other. This level represents full distribution and enables distribution over $2N$ devices.

5.2.1 Monolithic DRRM implementation

In order to clarify the issues that arise when selectors are distributed and explain our solutions, we refer to an implementation of the DRRM algorithm. We consider the “enhanced” version of DRRM [22], which achieves lower mean latency via a modified pointer update rule similar to that used in FIRM. The techniques that we present are applicable to other two- and three-phase, pointer-based algorithms as well.

DRRM computes a matching in every timeslot in a sequence of iterations. The following steps are performed in every iteration (initially all inputs and outputs are unmatched):

- Step 1: Request. Each unmatched input sends a request to an unmatched output corresponding to the first nonempty VOQ in round-robin order, starting from the current position of the request pointer. In the first iteration the pointer is updated to point to the output just selected. The pointer is further updated to one position beyond the output selected (modulo N) if and only if the request is granted in step 2 of the first iteration.

- Step 2: Grant. If an output receives one or more requests, it chooses the one that appears next in a fixed round-robin order starting from the current position of the grant pointer. The output notifies each requesting input whether or not its request was granted. The pointer is updated to one position beyond the input granted in the first iteration, modulo N. If there are no requests, the pointer remains where it is.

To facilitate the discussions that follows, Listing 5.1 shows a piece of C++ code that implements the DRRM matching algorithm in a monolithic fashion. N represents the number of ports and I the number of iterations. The arrays `imatch[]` and `omatch[]` store the port number that each input and output are matched to, respectively. They are initialized to the value -1 (i.e., unmatched) at the start of every timeslot. `reqPtr[]` and `grtPtr[]` are the round-robin request and grant pointers, respectively. The two-dimensional `requests[][]` array stores the number of requests for every VOQ. For the time being we shall assume that $I = 1$. The input selection (request) is performed is lines 5–18, whereas the output selection (grant) takes place in lines 20–36. Lines 12–13 implement the enhanced request pointer update policy.

Listing 5.1. C++ implementation of the DRRM matching algorithm.

```
1 void DRRM::schedule() {
2   int i, x, inp, outp, inpReq[N];
3   for (i = 0; i < I; i++) {
4     // request
5     for (inp = 0; inp < N; inp++) {
6       inpReq[inp] = −1;
7       if (imatch[inp] == −1) {
8         for (x = 0; x < N; x++) {
 9            outp = (reqPtr[inp]+x) % N;
10            if (omatch[outp] == -1 && requests[inp][outp] > 0) {
11              inpReq[inp] = outp;
12              if (i == 0) // Enhanced DRRM
13                reqPtr[inp] = outp;
14              break;
15            }
16          }
17        }
18      }
19      // grant
20      for (outp = 0; outp < N; outp++) {
21        if (omatch[outp] == -1) {
22          for (x = 0; x < N; x++) {
23            inp = (grtPtr[outp]+x) % N;
24            if (imatch[inp] == -1 && inpReq[inp] == outp) {
25              imatch[inp] = outp;
26              omatch[outp] = inp;
27              if (i == 0) {
28                reqPtr[inp] = (outp+1) % N;
29                grtPtr[outp] = (inp+1) % N;
30              }
31              break;
32            }
33          }
34        }
35      }
36    }
37  }
```

5.2.2 Separating Input Selectors from Output Selectors

Physically separating the input and output selectors (i.e. moving from DL0 to DL1) introduces a non-negligible round-trip time (RTT) between them, as illustrated in Figure 5.2. Assuming that this RTT is larger than the timeslot duration, there are two major implications, which we explain with the help of Listing 5.1.

The request decision of a given input i depends on the position of the request pointer `reqPtr[i]` and is stored temporarily in `inpReq[i]` (line 11). The requests made are then considered in the grant loop (line 24). In a distributed implementation, things are different. First, the request information is delayed by RTT/2[2]. Moreover, as the request pointers are physically located at the input side, the pointer update (line 28) cannot be performed immediately after the grant; this update occurs after RTT/2, i.e., when the

[2] For ease of discussion we assume that the RTT is symmetric, i.e. the up- and down-stream latencies are equal.

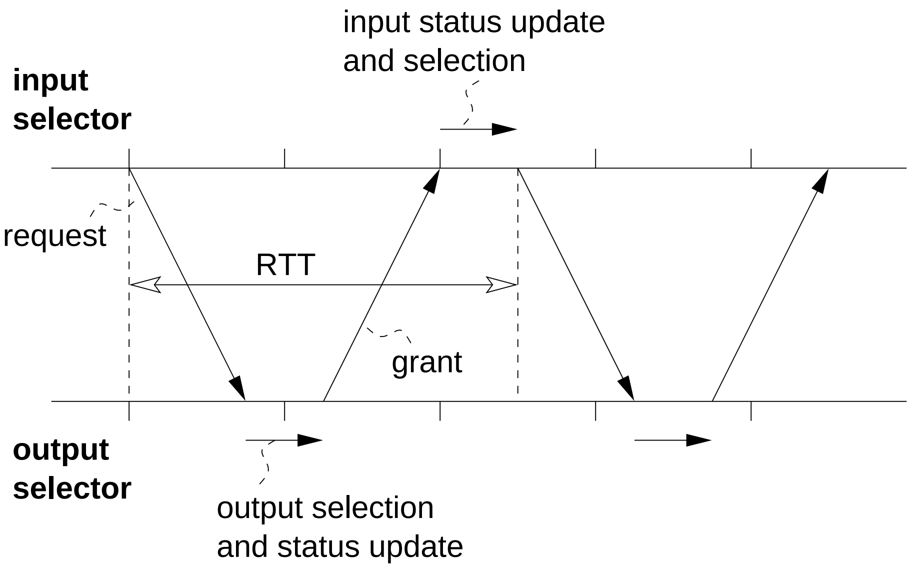


Figure 5.2. Round-trip time between input and output selectors.

grant arrives at the input selector. This has a further important consequence: unlike in a monolithic implementation, the requests to be issued in the next timeslot are based on pointer positions that are not updated according to the most recent grants. This breaks the round-robin desynchronization mechanism, leading to throughput limitations well below 100%.

Another consequence of the delayed availability of grant information is that the request selector is not able to accurately know for how many of the cells waiting in a given VOQ grants are already underway. This affects its request decisions: clearly, issuing requests for VOQs which are soon going to be empty is a waste of resources. Section 5.3.1 introduces pointer update policies to address the desynchronization issue, while Section 5.3.2 deals with delayed grants.

5.2.3 Achieving further distribution levels

The issues just identified appear when moving from DL0 to DL1. When moving further, from DL1 to DL2, distribution of input schedulers prevent them from sharing information. This is not a problem, because by design they work independently and base their decision on local information only. Hence, the techniques presented in Sections 5.3 and 5.4, can be applied to achieve DL2 as well.

DL3 prevents output schedulers from sharing status information. This does not inhibit use of techniques presented Section 5.3 but impedes multiple iterations, as better explained in Section 5.4.

5.3 Distributed Implementation

Our objective is to enable physical separation of the input and output selectors with an arbitrary RTT latency between them, while preserving performance and fairness.

For the moment we assume that during each timeslot only one iteration is performed. Performing multiple iterations poses additional challenges and will be discussed in Section 5.4.

5.3.1 Pointer Update Approaches

The key to achieving performance and fairness in a distributed implementation is to conserve the pointer desynchronization property. We have developed two techniques to do so: the *multi-pointer* approach, based on the duplication of status information, and the *pointer-cursor* approach, based on heuristic pointer updating.

Multiple Pointers

In the first approach, each input and output selector maintains a distinct pointer for every timeslot of the input-output round-trip. These pointers are labeled $R_i(t)$ and $G_j(t)$ for the request and grant pointers, respectively, with t being the temporal index. By *pointer set* we denote the set of all pointers $R_i(t)$ and $G_j(t)$ corresponding to a specific index t, so there are a total of RTT pointer sets.

The traditional pointer update rules are used: request pointers are only updated at the time a grant arrives (which happens one RTT after issuing the corresponding request), whereas grant pointers are updated immediately after issuing a grant, because issued grants are automatically accepted.

In a given timeslot t each input i issues requests using pointer $R_i(t \mod \text{RTT})$. When a request issued using a pointer with temporal index t is granted, the corresponding grant pointer $G_j(t)$ with the same index t is updated.

At timeslot $t_0 + \text{RTT} - 1$ the grant decision for requests submitted at timeslot t_0 will arrive, so the pointers $R_i(0)$ can be updated and used again in timeslot $t_0 + \text{RTT}$. The output selectors use a different pointer at every timeslot in the same way.

This pointer update policy implies that all pointer sets evolve independently and that each request pointer is never reused before being updated according to the result of its previous request. Therefore, it preserves the important features of the matching algorithm regardless of the value of RTT. In particular, pointers belonging to each set will eventually desynchronize, resulting in 100% throughput. Fairness is preserved as well, as each input will request the same output at least once every RTT time slots, until it is granted.

This scheme requires RTT pointer status registers (each $\log_2 N$ bits wide) per selector. However, the combinatorial selection logic does not have to be duplicated. Instead, every selector employs a multiplexer to select one of the registers depending on the temporal index. Also needed is a counter (modulo RTT) indicating the current pointer set to be used.

Pointer-Cursor Approach

The fact that the number of registers required at each input and output selector is proportional to RTT constitutes a drawback of the multi-pointer approach. Here we describe an alternative solution that offers slightly lower performance but employs only *two* registers per selector, regardless of RTT. We refer to the first set of registers simply as *pointers* and the second set as *cursors*; they are used in different timeslots and updated in different ways. Each input maintains a counter K, which is incremented modulo RTT at the end of every timeslot.

Each input uses the following policy to determine which output to request: If $K = 0$, the input selector makes its selection using the pointer. Otherwise, it uses the cursor and advances it to one position beyond the output selected modulo N *without* waiting for the result. If, at the end of a timeslot, the input receives a grant that was produced using a (grant) pointer, the input selector updates its (request) pointer to one position beyond the granted output modulo N *and* copies the value of the pointer to the cursor.

Every output selector operates as follows: If it receives requests that were produced using (request) pointers, it issues a grant using its (grant) pointer, else it issues a grant using the cursor. In either case, pointer or cursor are updated to one beyond the input granted modulo N.

To know whether the requests (grants) received were produced using pointers or cursors, a bit can be added to the protocol or, alternatively, a counter can be used at each output (input), as it is known that first of each group of RTT requests (grants) are issued using pointers, the remaining using cursors. All selectors must be synchronized to use the pointers in the same timeslot.

The idea behind this solution is that we can have a “slow”, but rigorous matching algorithm (using pointers), overlapped with a simple round-robin algorithm (using cursors). Every request-grant cycle of the slow matching algorithm takes RTT timeslots. However, the pointers are strictly updated according to the algorithm rules, hence they will eventually desynchronize and guarantee fairness. Once desynchronization of pointers has been achieved, the copy operation propagates it to cursors. As a matter of fact, the cursors start from the positions of the pointers (which are desynchronized, hence point to different outputs) and afterwards, being all moved by one position at every timeslot, will remain desynchronized. If not all the VOQs are nonempty, the round-robin policy that is used to update cursors is not optimal, as it might lead cursors to synchronize again. However, as soon as a request-grant cycle using pointers is completed, the situation will be corrected by aligning cursors to pointers, and desynchronization is regained.

Although this solution guarantees 100% throughput when the switch is uniformly loaded at 100%, performance at intermediate loads decreases as RTT increases, because cursors are updated less frequently and “sub-optimal” cursor positioning (caused by empty VOQs) take longer to be corrected. If RTT is particularly large, it is possible to increase the number of pointers and align cursors more frequently. For instance, if two pointers are used instead of one, cursors can be aligned every RTT/2 timeslots. In the extreme case, there are RTT sets of pointers, which falls back to the multi-pointer solution.

5.3.2 Pending Request Counters

The VOQ status registers reside close to the input selectors. `requests[i][j]` is incremented whenever a new cell arrives for VOQ(i,j) and decremented whenever a grant for VOQ(i,j) is issued. The RTT introduced by the distribution implies that when an input selector submits a request, it has to wait RTT time slots before knowing whether it was granted or not. In the meanwhile, this cell is considered as unscheduled, so the input selector can submit further requests. If the number of submitted requests exceeds the number of enqueued cells, it may happen that a slot is reserved for a VOQ that is currently empty. In general, this is undesirable because grants that arrive for an empty VOQ are wasted, while another cell may have used this timeslot.[3]

To avoid the problem of issuing too many requests for a given VOQ, we introduce *pending request counters* (PRC, labeled P_{ij}) per VOQ plus a request history per input selector. The request counter P_{ij} is incremented when input i issues a request for output j. The request history $H_i(t)$ for input selector i is an array with RTT entries, where entry $H_i(t)$ indicates the output that was requested t timeslots ago. In every timeslot, input selector i decrements P_{ij} for which $j = H_i(\text{RTT} - 1)$. As a result, P_{ij} keeps track of the number of requests per VOQ for which the results are still pending.

The input selectors use these counters to filter their requests. Any VOQ for which the pending request counter P_{ij} exceeds or equals the current VOQ occupancy is not eligible to issue a new request. This prevents grants from being wasted and therefore improves performance.

This enhancement, while not strictly necessary for either of the solutions we propose, is beneficial in the presence of large RTT and light loads or when traffic is heavily unbalanced and different VOQs have significantly varying occupancy. Section 5.5 demonstrates the performance improvement obtained by the PRCs.

[3] On the other hand, there is a possibility that, although the VOQ was empty at the time the output selector issued the grant, a new arrival occurs in the meanwhile. In that case, this arrival will benefit from a reduced scheduling latency, as it receives a grant in less than one RTT.

5.4 Performing Multiple Iterations

In a monolithic implementation, performing multiple iterations per timeslot significantly improves performance by allowing more edges to be added in case multiple inputs requested the same output (or multiple outputs granted the same input in the case of i-SLIP). In our distributed implementation the effectiveness of subsequent iterations is lower, as explained below.

In Listing 5.1, the matched ports are indicated by the `imatch[]` and `omatch[]` arrays. These are updated in lines 25–26 when a new edge is added. In the next iteration, these updated values are taken into account in lines 7 and 10 to produce requests for that iteration. In our distributed implementation, these updates occur at the output side, so the input side does not learn of them for another RTT/2 timeslots. The input selectors, which have to choose *at the beginning of the timeslot* which output to request in each iteration, do not know which outputs will be matched at the end of each iteration and should therefore be disregarded. As a result, we can have *wasted requests*.

Note that in all iterations except the first one, outputs must disregard requests from inputs that have already been matched. This assumes that the output selectors have shared access to this information, which is true as long as they are implemented in a single device (up to DL2), but is not the case when they are also separated (DL3).

We address the issue of wasted requests by adding a separate pointer `flywheel[inp]` to every input selector. In the first iteration, a selection is made using the round-robin pointer `reqPtr[inp]`. The `flywheel[inp]` is updated to one beyond the output just requested, modulo N. In subsequent iterations, the input selector is operated using the `flywheel[inp]` rather than the `reqPtr[inp]`. After every selection, the `flywheel[inp]` is updated to one beyond the output just requested, modulo N. This way, we make sure that the input selector requests as many different outputs as possible across the iterations, although there is no guarantee that the outputs requested are still available. Each input selector also keeps track of which outputs it has already requested in the current timeslot and avoids requesting the same output more than once, as this would be useless.

PRC-based request filtering, as described above, ensures that the number of wasted grants is minimized. On the other hand, overly conservative filtering can be detrimental: once the filtering condition is met, a new request can only be submitted when the the result for the first in-flight one is received. This can introduce gaps in the request pipeline and therefore cause unnecessary delays. Furthermore, requests for subsequent iterations are increasingly less likely to be successful. As a result, our findings show that it is counterproductive to include requests beyond the first iteration in the PRCs and request history. Therefore, the PRC and request history operation (update and filter) apply only to requests in the *first* iteration. This choice, besides improving performance, simplifies the implementation of the input selectors.

Listing 5.2 shows a C++ implementation of the input selector for the multi-pointer approach. This procedure is executed once every timeslot and generates one request for every iteration. `reqPtr[]` is the request pointer and `reqFlywheel` is the request flywheel pointer. The pointer is updated according to the enhanced DRRM rule, but only in the first iteration. The flywheel is always updated to one beyond the output just requested. The `requested[]` flags keep track of which outputs have already been requested in the current timeslot. `ptr_set_id` is the pointer set ID corresponding to the temporal index, which is incremented by one (modulo RTT) in every time slot. `requests[]` keeps track of the number of requests per output, whereas `pending_requests[]` represents the pending request counter. When a request is made in the first iteration, the corresponding `pending_requests[outp]` counter is incremented, and a corresponding entry is made in the `request_history[ptr_set_id]` array.

Listing 5.2. C++ implementation of the input selector for the multi-pointer approach.

```
int j, iter, outp, t;
bool requested[N] = {false};
for (iter = 0; iter < I; iter++) {
  for (t = 0; t < N; t++) {
    if (iter == 0) // round-robin pointer
      outp = (reqPtr[ptr_set_id] + t) % N;
    else // flywheel
      outp = (reqFlywheel + t) % N;
    if (requests[outp] > 0) {
      if (requested[outp])
        continue;
      if (iter == 0 && pending_requests[outp] >= requests[outp])
        continue;
      req.outp = outp;
      req.iter = iter;
      req.ptr_set_id = ptr_set_id;
      reqFlywheel = (outp+1) % N;
      if (iter == 0) // EDRRM
        reqPtr[ptr_set_id] = outp;
      requested[outp] = true;
      if (iter == 0) {
        pending_requests[outp]++;
        request_history[ptr_set_id].outp = outp;
      }
      break;
    } // if
  } // for t
} // for iter
```

5.5 Simulation Results

We built a software model of the proposed architecture with the OMNeT++ [63] simulation environment and the Akaroa2 [64] parallel simulation tool. We simulated this model to obtain its performance characteristic, focusing specifically, on mean throughput (measured at the egress across all ports) and mean packet latency (measured from source to sink).

In our experiments, we study a switch with $N = 16$ ports using the distributed EDRRM architecture according to DL2. We vary RTT and measure the performance of the multi-pointer as well as the pointer-cursor approach. We also vary the number of iterations per timeslot. Section 5.5.1 presents results based on uniform i.i.d. Bernoulli arrivals, whereas Section 5.5.2 presents results based on bursty and nonuniform arrivals.

5.5.1 Uniform Bernoulli Traffic

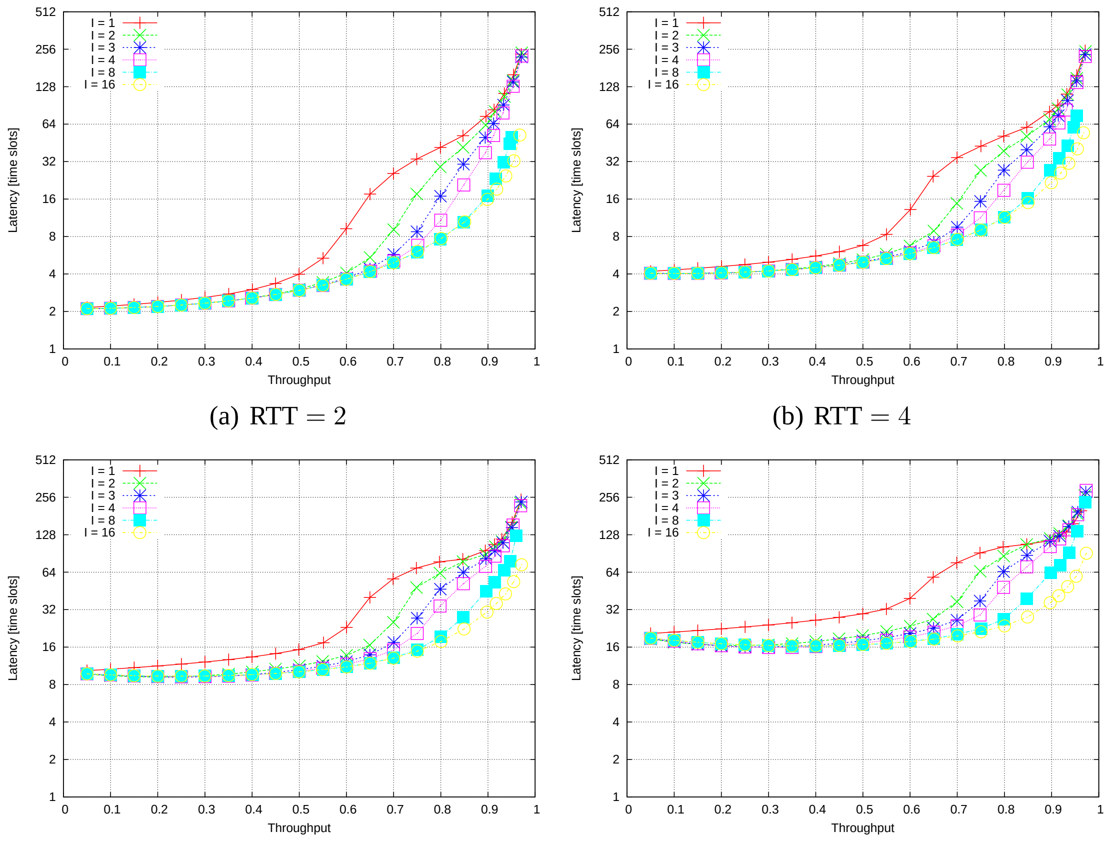
Figure 5.3 shows the results for the multi-pointer approach and Figure 5.4 shows those for the pointer-cursor approach. Both figures comprises subfigures for RTT = 2, 4, 10, and 20 timeslots. Each subfigure shows curves for I = 1, 2, 4, 8, and 16 iterations per time slot. Note that the minimum latency at very light load equals RTT. For reference, results using a monolithic DRRM implementation are also included, adjusted to take into account the constant latency component of the distributed implementation. These results lead to the following observations:

- The achievable maximum throughput exceeds 98% in all simulations, i.e., both with the the multi-pointer approach and the pointer-cursor approach, and for all values of RTT and I.

- The mean latency decreases significantly as the number of iterations increases. When $I = N = 16$, the performance of the distributed implementation is almost identical to that of the monolithic implementation with four iterations. Using as many iterations as there are ports overcomes the issue of wasted requests, as there is an opportunity to request every output in every time slot. However, it does not overcome the issue of uncertainty due to pending requests.

- When RTT is large, there is a load region in which the mean latency decreases as the load increases. This effect is caused by excess grants that, instead of going to waste on an empty VOQ, find a new arrival in their VOQ; these cells experience a latency smaller than RTT.

- The multi-pointer approach achieves lower latency than the pointer-cursor approach, especially at high utilization. The latency difference increases with RTT. This behavior is expected: The MP approach achieves faster pointer desynchronization because pointer updates occur in every timeslot, as opposed to once per RTT.



(a) RTT $= 2$

(b) RTT $= 4$

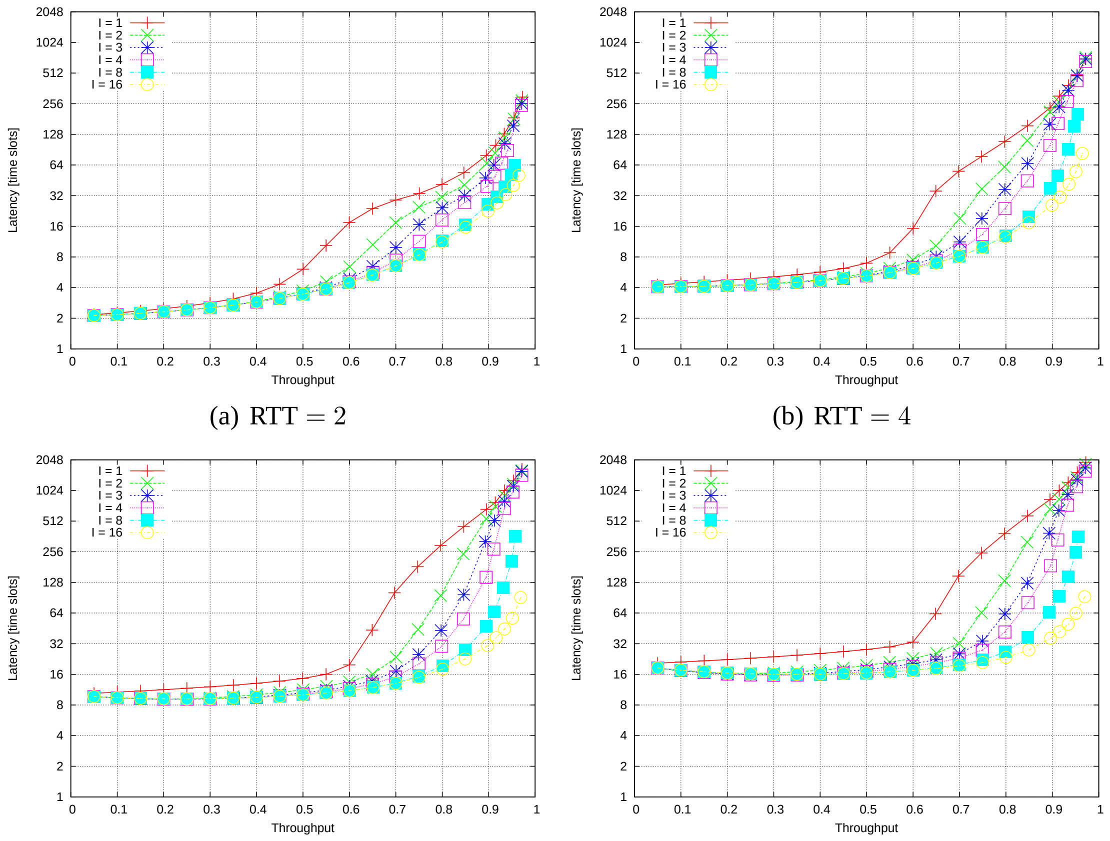
(c) RTT $= 10$

(d) RTT $= 20$

Figure 5.3. Delay vs. throughput curves for the multi-pointer approach with varying RTT.

Figure 5.5c shows the results for the multi-pointer approach with I = 1 and 4, and demonstrates the impact of RTT. Here, the mean latency is normalized with respect to RTT. These curves clearly show that, in a limited load range, the mean latency drops below RTT timeslots when RTT is large and $I = 4$.

Figure 5.6 compares the performance of the multi-pointer approach *with* (Figure 5.6b) and *without* (Figure 5.6a) pending request counters with $N = 16$ and RTT $= 4$. These graphs clearly show that use of PRCs achieves drastically lower latency throughout the load range. Considering the case $I = 1$, the main difference is in the load range from 10% to 70%; beyond 70% there is no noticeable latency difference. The reason is that, with heavy loads, the rate of wasted grants will be low, as most VOQs will be backlogged; therefore, the negative effect of excess requests is not noticed. At low to medium loads, on the other hand, many of the excess requests will result in wasted grants; every wasted grant potentially is a wasted opportunity to transmit another cell, which therefore incurs a longer latency. As a result, the mean latency increases. With $I = 16$, performance is



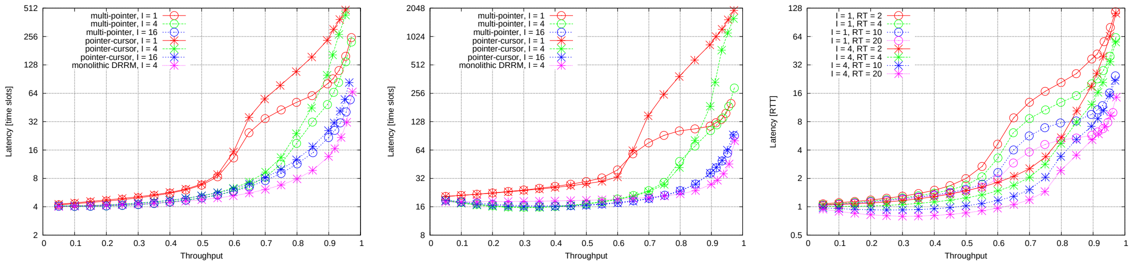
(a) RTT $= 2$

(b) RTT $= 4$

(c) RTT $= 10$

(d) RTT $= 20$

Figure 5.4. Delay vs. throughput curves for the pointer-cursor approach with varying RTT.



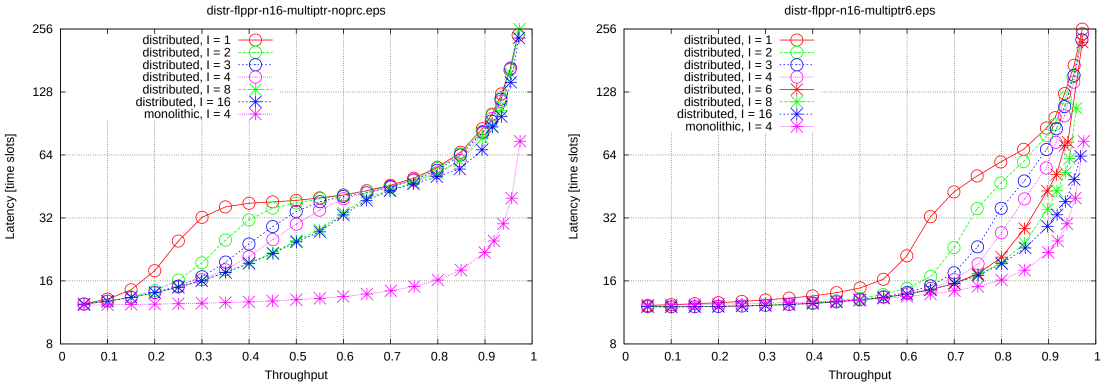
(a) Delay vs. throughput comparison of both approaches, RTT $= 4$

(b) Delay vs. throughput comparison of both approaches, RTT $= 20$

(c) Normalized delay vs. throughput curves for the multi-pointer approach with varying RTT.

Figure 5.5. Comparative figures.

close to ideal when using PRCs.



(a) Without PRCs

(b) With PRCs

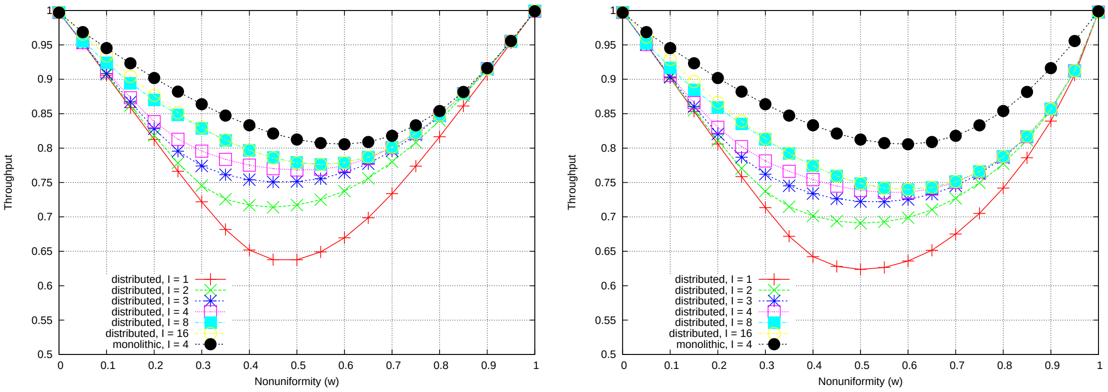
Figure 5.6. Delay vs. throughput curves for the multi-pointer approach with RTT $= 4$, comparing performance with and without pending request counters.

5.5.2 Bursty and Nonuniform Traffic

To study the performance under nonuniform traffic, we adopt a destination distribution characterized by a non-uniformity parameter w [65], where $w = 0$ corresponds to uniform traffic and $w = 1$ to fully unbalanced, contention-free traffic: $\lambda_{ij} = \lambda\left(w + \frac{1-w}{N}\right)$ if $i = j$, $\lambda\frac{1-w}{N}$ otherwise. Here, λ_{ij} represents the traffic intensity from input i to output j, $0 \leq i{,}j < N$; λ is the aggregate offered load, and w the non-uniformity factor. Note that no input or output is oversubscribed and that traffic is admissible as long as $\lambda \leq 1$. We vary the value of w from 0 to 1 and measure the throughput achieved at an offered load of 100%.

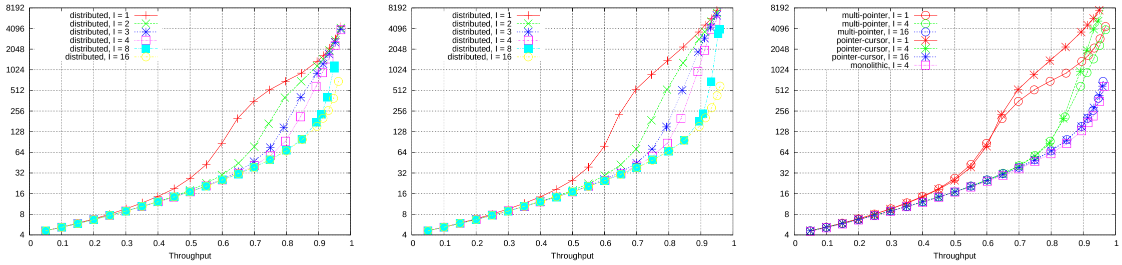
Figures 5.7(a,b) show the results for $N = 16$, RTT $= 4$, and Bernoulli arrivals for I ranging from 1 to 16. Also included for reference is a curve for monolithic DRRM with $I = 4$. All curves exhibit the behavior of dipping to significantly less than 100% throughput as w moves away from the extremes. However, increasing I increases the throughput. The multi-pointer approach is able to reduce the gap with the reference to below four percentage points when $I \geq 8$. Overall, we again observe that the multi-pointer approach performance better than the pointer-cursor approach, with a difference in throughput that is generally less than five percentage points.

We also evaluate the performance using bursty traffic with geometrically distributed burst sizes with average burst size of 10 cells. Figure 5.8 shows the results. Here, we first observe that the maximum throughput again exceeds 98% in all cases. Moreover, the latency differences with the reference curves are even smaller than with Bernoulli traffic and the difference between the multi-pointer and pointer-cursor approaches is also significantly smaller. Hence, the proposed distributed implementation is able to closely approximate a monolithic implementation in terms of performance for correlated as well as uncorrelated arrivals.



(a) Multi-pointer (b) Pointer-cursor

Figure 5.7. Throughput vs. non-uniformity curves for both approaches with $N = 16$ and RTT $= 4$.



(a) Multi-pointer (b) Pointer-cursor (c) Comparison

Figure 5.8. Delay vs. throughput curves using bursty traffic for both approaches with $N = 16$ and RTT $= 4$.

Chapter 6

Fair Integrated Scheduling of Unicast and Multicast Traffic in Input-Queued Switches

In this chapter we present a scheme to concurrently schedule unicast and multicast traffic in an input-queued switch. It aims at providing high performance under any mix of the two traffic types as well as avoiding starvation of any connection. The key idea is to schedule the two traffic types independently and in parallel and then arbitrate among them for access to the switching fabric. Unicast and multicast matching are combined in a single integrated one. The edges that are excluded from the integrated matching are guaranteed to receive service at a later time, thus preventing starvation. We use simulation to evaluate the performance of a system employing the proposed scheme and show that, despite its simplicity, it achieves the intended goals. We also design an enhanced remainder-service policy to achieve better integration and further improve performance.

This work was performed in the context of the OSMOSIS project (Chapter 4), but is generally applicable to input-queued crossbar-based synchronous switches.

6.1 Motivation

In environments where packet switches are used (TCP/IP networks, Storage Area Networks, supercomputer interconnects) the vast majority of traffic consists of unicast (point-to-point) connections. However, in all these contexts, support for multicast (point-to-multipoint) traffic is essential. On the Internet, multicast enables applications such as audio- and video-conferencing, multimedia content distribution (radio, TV) and remote collaboration; in SANs, it is required to replicate data among multiple sites or to distribute content to multiple servers; in supercomputing architectures it is essential to implement cache coherency protocols and support collective operations [40]. Ideally, a network switch should be able to achieve high performance under any mix of the two traffic types.

Multicast packets can be treated as unicast simply by sending a separate copy of the packet to each of the intended destinations; conversely, unicast packets can be considered as multicast packets with only one destination and treated without any differentiation. These trivial solutions allow the switch to handle both types of traffic concurrently but are far from optimal and generally lead to poor performance.

Another issue to address when both unicast and multicast are present is fairness. A traffic type must not be allowed to monopolize switch resources; however, it is also important to guarantee that *all* connections of a given traffic type receive service. When both conditions are met, we say that the switch scheduler is *fair*.

In this work we propose a novel method for integrated scheduling of unicast and multicast traffic. It leads to high utilization of switch resources under any traffic mix, guarantees fairness, and exhibits a number of other desirable properties.

Although the problem of supporting unicast and multicast concurrently is clearly important, not much attention has been devoted to it in the past. The problem has been thoroughly studied from a theoretical point of view in [66] and its hardness has been assessed. These authors also propose an integration scheme that consists of scheduling multicast first and using the remaining resources for unicast. This scheme, which we call "sequential" predictably leads to high performance because it uses the switch resources very efficiently. The multicast scheduler has all the resources at its disposal and can produce its best matching. The unicast scheduler, on the contrary, is constrained by the remaining resources but, thanks to the VOQs, it can fully exploit them and increase the size of the total matching. The main disadvantage of this scheme is that it easily leads to starvation of unicast traffic. A single input loading the switch with broadcast traffic would suffice to prevent unicast from getting any service at all. In [67] the authors propose a refinement of the sequential scheme, in which at some timeslots the unicast scheduler runs first while in other the multicast scheduler is given priority. The choice of which scheduler runs first in a given interval, however, is based on a parameter that depends on the traffic patterns, in particular on the ratio of multicast to unicast traffic, and that must be determined a-priori to guarantee high performance. Smiljanić showed that a practical approach to achieve integrated scheduling is to treat multicast traffic as unicast, but distributing the burden of cell replication over multiple ports [68]. The main problem with this scheme is that it potentially introduces very high latency, so it is not suitable for our applications. The problem was also considered in [69], but the proposed solution is mainly targeted to shared-memory switches.

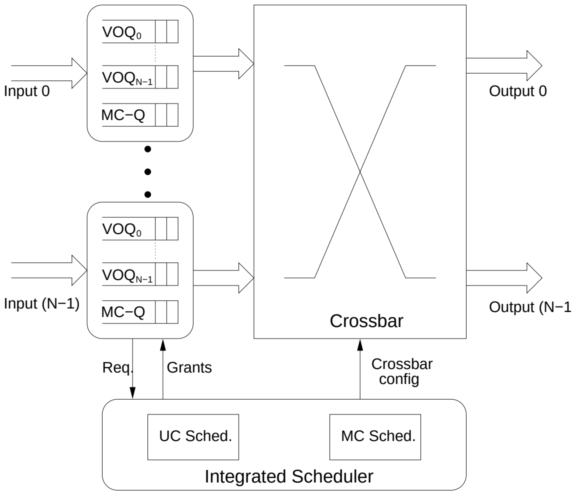


Figure 6.1. Reference architecture

6.2 Fair Integrated Scheduling

Our integration scheme is conceived for a synchronous, IQ, crossbar-based, $N \times N$ switch (Figure 6.1). The scheme is independent from the queueing structure adopted for unicast or multicast traffic, but for concreteness we refer to the most common situation in which each switch input maintains N VOQs for unicast and a single FIFO queue for multicast.

6.2.1 Reference architecture

At every timeslot, contentions among the cells of a single traffic type are resolved separately by specialized schedulers. The unicast scheduler receives requests from the inputs for non-empty VOQs and produces a one-to-one matching between the inputs and the outputs. The multicast scheduler examines the fanout of the cells that are at the HOL of the multicast queues and produces a one-to-many matching. Fanout splitting is allowed: during a timeslot a multicast cell can receive partial service, being transmitted only to a subset of its destinations.

As the two schedulers run in parallel and independently, the matchings they produce in general are overlapping, meaning that they have conflicting edges. To obtain a consistent configuration for the crossbar, the two matchings must be combined into a single one. An *integration block* decides which unicast and multicast edges will be part of the integrated matching. The set of edges that are excluded from the integrated matching is called the *remainder*.

The *request filter* is a block capable of reserving a subset of the switch inputs and outputs by dropping the corresponding unicast and multicast requests. Reservations at

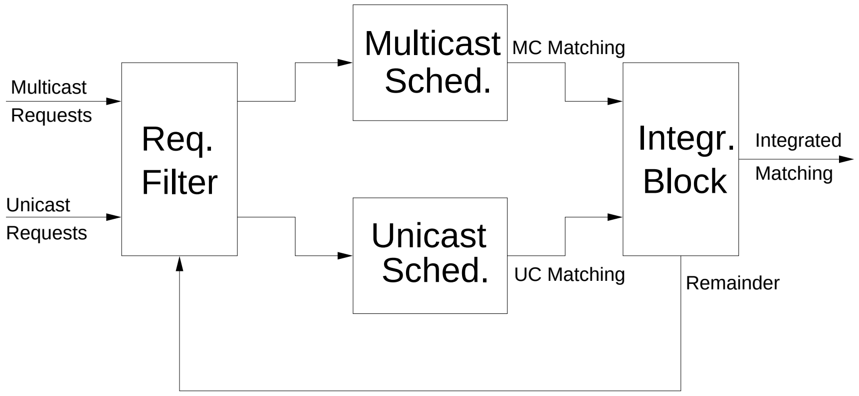


Figure 6.2. The FILM integration scheme

any timeslot may be made on the basis of information provided by a number of sources, including current requests and the integration block.

Employing two different schedulers that run in parallel provides important advantages. The designer is free to choose the algorithms that better fit his or her needs. The system can easily be partitioned over multiple chips. The minimum timeslot duration is determined by the scheduling time of the slowest scheduler, whereas, if the schedulers ran in sequence, it would be limited by the sum of the two.[1] Moreover, it avoids the additional latency naturally introduced by sequential schemes.

A block diagram of this scheme, called "FILM" (FILter & Merge), is shown in Figure 6.2.

6.2.2 Achieving fairness

In the FILM scheme each connection goes through two points of contention: first it competes with the other connections belonging to the same traffic type, then with those of the other traffic type. To achieve fairness we must make sure that every connection regularly has a chance to win both contentions.

A scheduling algorithm is starvation-free if it guarantees that no queue is allowed to remain unserved indefinitely. As this is a fundamental property, many algorithms exhibit it. Unicast algorithms such as i-SLIP [19] and DRRM [20] prevent starvation by using pointers that keep track of which VOQs have been served most recently. Multicast algorithms, on the other hand, often take into account the age of a cell or the order in which cells at different inputs have advanced to the HOL of their queues (e.g. WBA and TATRA [34], respectively). We require both schedulers to employ starvation-free algorithms to be sure that all connections eventually get past the first contention point.

[1] We assume that the delay contributed by the additional blocks is much lower than the scheduling times. As we will see in Section6.5 devoted to implementation complexity, this assumption is likely to hold.

Connections that have been selected by their schedulers still remain unserved if the integration block excludes them from the integrated matching. The scheduler is unaware of the fact that granted service has in fact been withdrawn, so fairness is no longer guaranteed. A solution to this problem is to make sure that all edges that are part of the remainder actually receive service, albeit in a later timeslot.

6.2.3 Integration policy

The performance of multicast scheduling algorithms varies considerably, as demonstrated in [34]. This is due to the fact that the single FIFO queuing architecture causes HOL blocking, therefore the algorithms must carefully choose which inputs to serve in order to mitigate its effects. For example, it is shown that “concentrating the residue” at every timeslot (which roughly means providing full service to as many inputs as possible) greatly helps in draining the queues fast. Hence, special care should be taken when manipulating multicast matchings to avoid compromising the effectiveness of the choices made by the scheduler.

Unicast scheduling, on the contrary, is less sensitive to withdrawal of resources because the VOQs provide the scheduler with a wide choice of connections to serve. Moreover it is important to note that if unicast and multicast contend for an input, only one edge is lost if multicast wins, whereas multiple edges might be removed if it loses.

Following these considerations, we opt for an integration policy that gives strict priority to multicast over unicast. Hence, the algorithm implemented in the integration block can be formulated as follows:

1. Start with an empty matching,
2. add all multicast edges,
3. add all non-conflicting unicast edges.

As a consequence, the remainder always contains only unicast edges.

6.2.4 Remainder-service policy

As noted above, if a remainder is produced in a timeslot, it is important to ensure that all the edges it contains are eventually served. This can be done according to different policies, the simplest one being to serve all of them in the next timeslot. As these edges are part of a matching, they do not conflict with each other. In addition, the resources they claim are known and can be reserved to avoid further contention.

At every timeslot, new unicast and multicast requests are issued. The request filter drops all those that involve inputs and outputs needed to serve the remainder produced in the preceding timeslot and submits the others to the corresponding scheduler. Accordingly, the integration block issues grants for the edges in the remainder and for those in the current matching. A new remainder is produced and fed back to the request filter for the next timeslot.

An important property of the scheme is that, as a consequence of filtering unicast requests, the remainders produced in two consecutive timeslots are disjoint, i.e., have no inputs or outputs in common. This is crucial for fairness because it assures that all switch resources eventually become available for scheduling. Reserving resources for the remainder does not persistently preclude access to any input or output.

We expect this combination of integration and remainder-service policy to achieve good link utilization. The resources allocated to the remainder are fully utilized and those remaining can be assigned either to unicast or multicast. The integration block preserves the matching produced by the multicast scheduler, but tries to enlarge it by adding unicast edges.

6.3 Simulation Results

We have studied the performance of a system employing the FILM scheme by simulation. In particular, we observed the total throughput as well as the individual throughputs of unicast and multicast traffic as the fraction of multicast traffic (MCF) grows from 0 (unicast only) to 1 (multicast only). Ideally, the throughput achieved by each traffic type should be equal to the corresponding share of the output load and the total should be 100%.

The simulated system is an 8×8 switch with infinite buffers at the inputs. The unicast scheduler uses i-SLIP with three iterations and the multicast scheduler uses WBA. Simulations run for 1 million cell times and results are collected after a quarter of the total simulation time has elapsed.

Cells are generated according to an i.i.d. Bernoulli process, i.e. every input port receives a cell with probability ρ, equal to the input load. Each cell has a probability P of being a multicast cell. The fanout of multicast cells is uniformly distributed between 2 and 8. Traffic is uniform, i.e. all outputs have the same probability of being the destination of a unicast cell or of belonging to the fanout of a multicast cell. Note that, under these conditions and with this choice of scheduling algorithms, when the switch is loaded only with multicast traffic, the maximum throughput it can achieve is approximately 0.93 [34], whereas it is 1.0 when only unicast is present.

The total load on the switch is $\rho(P\overline{F} + (1 - P))$ where $\overline{F}$ is the average fanout. In our case, $\overline{F} = 5$ and P and ρ are varied to obtain the desired multicast load on the switch while keeping the total load equal to 1.

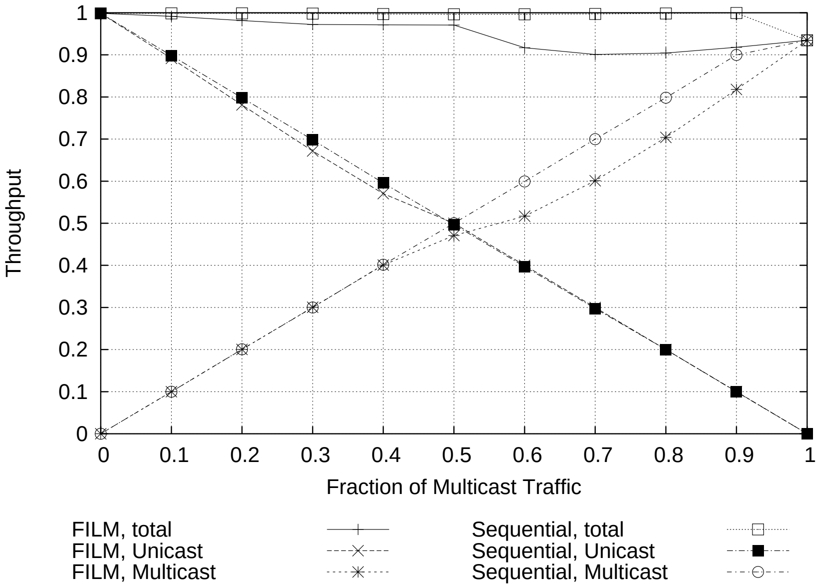


Figure 6.3. Performance of the FILM integration scheme

Figure 6.3 shows the throughput achieved by FILM with the integration and remainder-service policies described in the preceding section. The performance of the sequential scheme, which is close to ideal, is also shown for reference.

The total throughput achieved by our scheme is always higher than 0.9. Unicast throughput shows very little degradation (on the order of a few percent) when it is the predominant traffic type, whereas it achieves ideal performance when multicast is predominant. However, multicast throughput progressively decreases with respect to output load as MCF grows from 0.4 to 1.0. The worst case is MCF $= 0.7$, when multicast throughput is 0.6 instead of 0.7. This also corresponds to the point at which the overall throughput is at its minimum (0.9).

Figure 6.4 shows the delay experienced by unicast and multicast cells as a function of the throughput when MCF $= 0.5$, i.e., when each traffic type is responsible for half of the output load. The unicast curve is bounded for any value of the total throughput, whereas the multicast curve saturates when it approaches 1.0.

6.4 Enhanced Remainder-Service Policy

Although the scheme presented above provides overall good performance and is quite simple, it has a drawback: it penalizes multicast traffic most, especially when it is predominant.

Multicast performance is limited because at every timeslot, some switch resources are used to discharge the remainder. Although it is essential to eventually serve all edges

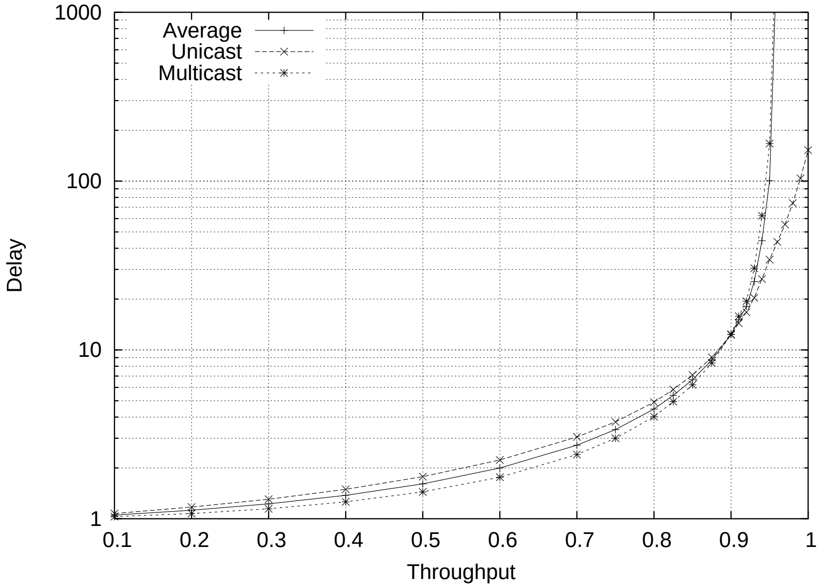


Figure 6.4. Delay vs. throughput curves for the FILM integration scheme

that are not selected in the merge, it is not necessary to do so immediately. Thanks to the disjoint remainder property, it is possible to *accumulate* the remainders produced in consecutive timeslots and serve the individual edges when the conditions are most favorable. The remainder-service policy identifies which edges should be served at every timeslot and filters the corresponding multicast requests. Unicast requests, in contrast, are always filtered using all the accumulated edges to obtain disjoint remainders.

A good policy should be able to serve the edges in the remainder rapidly and at the same time cause as little disruption as possible to the flow of multicast cells. We propose an *enhanced* policy that serves a remainder edge if it uses

1. an input not requested by multicast OR
2. an output not requested by multicast OR
3. an input that discharged a multicast cell in the preceding timeslot.

The first two rules obviously aim at improving integration: if it is possible to use a resource that would otherwise remain idle, it is desirable to do so. In this case the cost of serving a remainder edge is to make one output (first rule) or one input (second rule) unavailable to multicast.

The third rule instead stems from the general observations on multicast scheduling found in [34]. The scheduler tends to favor cells that contend with few others. Cells that have just advanced to the HOL still have their full, usually large, fanout and cause many conflicts. They are unlikely to receive much consideration, so postponing their scheduling

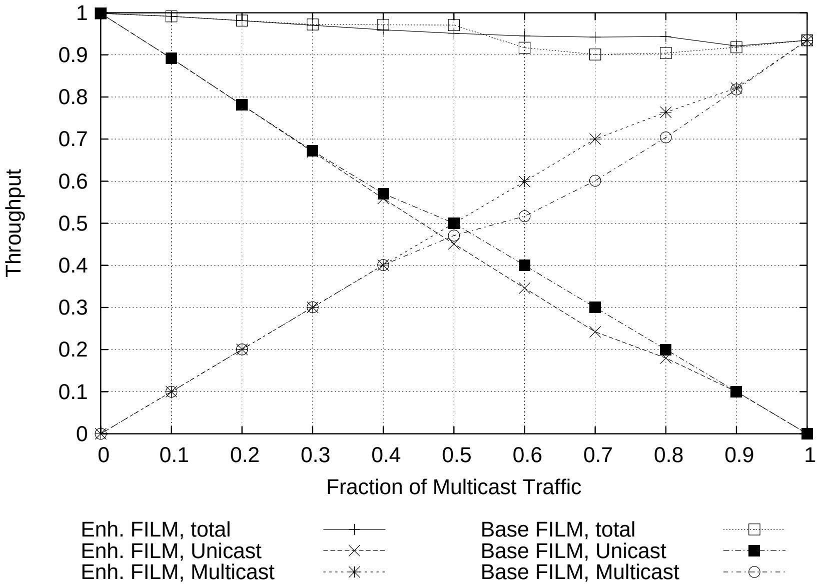


Figure 6.5. Performance of FILM with enhanced remainder-service policy

should not significantly affect the quality of the matching. This rule is particularly important because it enables fairness: the multicast scheduler guarantees that the HOL cell at any input will be served in finite time; consequently, the inputs becomes available to serve remainder edges. Many algorithms (such as TATRA, WBA and mRRM [70]) ensure that at least one multicast cell is fully discharged at every timeslot.

Figure 6.5 shows the performance of FILM when the enhanced policy is used, under the same conditions as in Section 6.3. The benefits on multicast traffic are evident: throughput is increased when $(0.4 < \text{MCF} < 0.9)$ and closely tracks the output load up to MCF $= 0.7$. Unicast, on the other hand, shows a moderate decrease in the same range. In the worst case (MCF $= 0.7$), the difference with respect to the output load is slightly lower than 0.06. Overall throughput is noticeably increased when multicast predominates, whereas it shows little degradation when both traffic types are equally active.

Figure 6.6 shows the delay vs. throughput curve for this situation. Multicast experiences very low delay, seeming to be almost insensitive to the presence of unicast. Unicast delay instead saturates when the total throughput is approximately 0.95.

6.5 Implementation Complexity

In this section we discuss some implementation aspects of the FILM scheme in order to get an idea of its complexity.

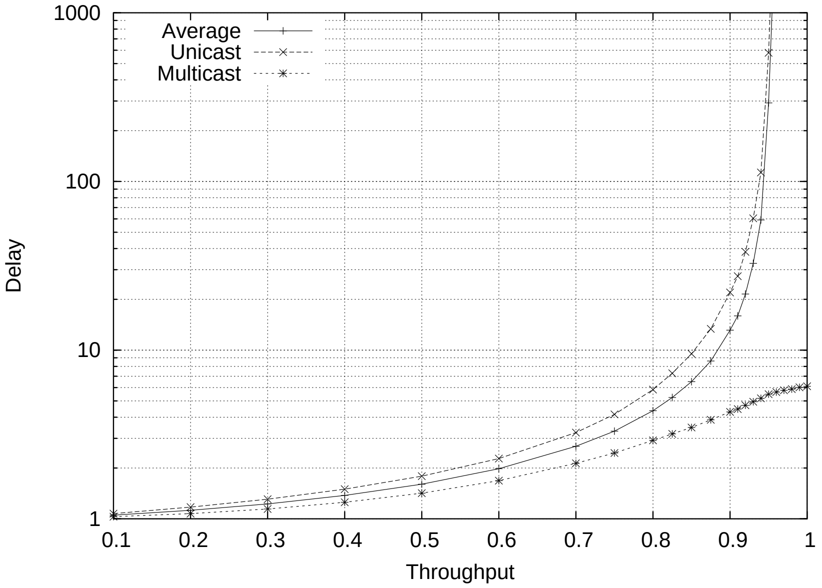


Figure 6.6. Delay vs. throughput curves of the enhanced FILM integration scheme

6.5.1 Integration policy

As the integration policy always prioritizes multicast over unicast, its implementation is quite straightforward. From the output of the multicast scheduler, it is immediately known which inputs and which outputs are used by the multicast matching. This information ($2N$ bits) in turn determines whether an edge in the unicast matching is to be interpreted as part of the integrated matching or of the remainder. In the former case, grants are released immediately, in the latter the information is buffered for subsequent timeslots. The remainder can be stored using N registers, each $(\log_2 N + 1)$ bits wide.

6.5.2 Base remainder-service policy

The request filter needs to know which inputs and outputs are used by the remainder edges in order to drop the corresponding requests. This information is available at the integration block and can be carried to the request filter with a channel $2N$ bits wide. Filtering a request for an input-output pair simply translates to ANDing it with the negated values of the corresponding signals.

6.5.3 Enhanced remainder-service policy

When the enhanced policy is used, the request filter needs more information and performs more complex operations. It needs to know exactly which edges are in the remainder, not only which inputs and outputs are taken. This means that $N(\log_2 N + 1)$ bits must be transferred from the integration block. The information about which inputs discharged a multicast cell in the preceding timeslot consists of N bits and can be maintained by the queues managers. Finally, the information about which inputs and which outputs are being requested by multicast ($2N$ bits) is readily available as it can be derived from the requests themselves.

Unicast requests are filtered using all edges in the remainder as in the previous case, whereas multicast requests are filtered depending on which remainder edges are served. This information is produced at the request filter block by ORing the signals corresponding to the three conditions that grant service to an edge. The integration block also needs to know which edges are served, as it has to issue the appropriate grants. As the remainder edges are part of a matching, only N bits need to be transferred from the request filter to the integration block.

As a final remark, we wish to highlight that all the operations described above can be performed in parallel and implemented using combinational logic only.

Chapter 7

Conclusions – Part I

In the first part of this thesis we have discussed the design of packet switches for supercomputer interconnection networks.

We have started with an overview of supercomputing systems, illustrating how various factors such as the node architecture, the partitioning of the memory space and the programming model influence the requirements of the interconnection network. We have explained why optical switching has the potential to be the best technology to satisfy the demanding requirements of supercomputers and which factors limit its deployment.

We have then introduced OSMOSIS, a research project jointly developed by IBM and Corning that aims at building a demonstrator interconnect for HPC applications. The building block of the interconnect is a hybrid switch with an all-optical data-path and an electronic control-path. The switch is designed to meet an ambitious set of requirements that include very low latency, high throughput, high port count, high line rate, scalability to thousands of ports and efficient support of multicast traffic. The design of the control-path is further complicated by the need to use only FPGAs.

In the context of the OSMOSIS project we have the developed techniques that enable the construction of multi-chip crossbar schedulers, which constitute the first contribution of this part of the thesis. These techniques overcome the area, pin-count and power density constraints of single-chip schedulers and thus allow scheduling of much larger crossbars than previously possible. The distribution techniques we have proposed can be applied to various parallel iterative matching algorithms, such as DRRM and SLIP, and preserve their throughput and fairness properties. The first, named *multi-pointer*, is based on duplication of status information and the performance it provides is almost insensitive to the RTT between chips. The second one is called *pointer-cursor*, is based on heuristic update of status information and has constant complexity; performance, however, degrades as the RTT grows. Simulation results show that high performance levels are maintained under uniform i.i.d. traffic, even when the scheduler is distributed over $2N$ chips separated by distances equivalent to several time slots. Moreover, with proper distribution level and number of iteration, satisfactory performance is achieved also under non-uniform and bursty traffic.

We then devoted our attention to the problem of scheduling concurrently unicast and multicast traffic in an input-queued switch. We have developed a novel integration scheme, named FILM, capable of scheduling the two traffic types fairly and efficiently, without a-priori knowledge of traffic characteristics. The scheme first schedules the two traffic types separately and then arbitrates among the results for access to the switching fabric. A integration block combines the matchings produced by the two schedulers, producing an integrated matching and a remainder. The remainder contains edges that cannot be served in the current time slot, but are guaranteed to receive service in a subsequent one. The first remainder service policy we have proposed is extremely simple and performs well, but tends to penalize multicast. The second one is more sophisticated and is able to minimize the interference with the flow of multicast cells. It leads to a very high overall performance and an almost ideal treatment of multicast traffic, at the cost of some additional complexity.

Although the work described in this part of the thesis has been performed to specifically address the challenges posed by the design of the OSMOSIS scheduler, we believe that it is valuable in all the contexts in which high-performance packet switches are used.

Part II

A Switching Architecture for Storage Area Networks

Chapter 8

Introduction to Storage Area Networks

Nowadays servers are at the center of the enterprise information system. They run mission-critical applications, such as enterprise resource planning (ERP), supply chain management and customer relationship management (CRM). With the advent of the Internet and e-business, servers are used to do on-line transaction processing (OLTP) and provide services to millions of users. They must be able to access data in the storage subsystem quickly and reliably. Failure to do so directly translates to significant costs and loss of revenue.

In this chapter we describe the evolution of the I/O interface between servers and storage devices, pointing out the limits of directly-attached storage and how storage networking overcomes them. We then introduce Fibre Channel as the preferred network technology to implement a SAN and outline its most important characteristics.

8.1 Limits of directly-attached storage

Computing nodes have been traditionally connected to their storage resources (disks, tapes, CD libraries, etc.) by means of a fixed, dedicated channel, such as the SCSI parallel bus.

In recent years this paradigm, called "Directly-attached Storage" (DAS) has become inadequate. As servers grow in number and request additional capacity, several different problems arise. The most important are:

- **Scalability:** the number of devices that can be attached to a disk controller is limited to few tens. Even with multiple controllers in the same server, the total available capacity might be insufficient.

- **Performance:** as the physical media is shared, adding devices results in more arbitration overhead and less bandwidth being available to each of them.

- **Distance limitations:** parallel buses are limited in length to few tens of meters by electrical issues, such as skew. Skew is a phenomenon typical of parallel transmission. Electrical pulses transmitted on different line of the parallel bus do not reach the target device at exactly the same time. If the delay between the first and the last arriving pulse is comparable to the time slot occupied by the pulse itself, the receiver cannot correctly decode the transmitted bit string.

- **Availability:** devices attached to the bus cannot be added or removed without putting the whole string off-line. This causes downtime every time the storage subsystem needs to be reconfigured.

- **Data protection:** each server must be equipped with proper devices (for example, tape drives) to backup its data. With hundreds or even thousands of servers, this is costly and quickly becomes an administrative burden, as each server must be backed up separately. If backup operations are performed through the LAN, the performance of the corporate network might be severely impacted for long time frames.

- **Efficiency:** disk space not used by a server cannot be relocated to another one. The administrators may need to buy and install additional storage devices even if free space exists on those already available.

A close look to the problems listed above suggests that many of them are intrinsic to the DAS model and cannot be solved simply with technological enhancements.

8.2 Storage Area Networks

Storage Area Networks (SANs) have emerged as the key solution to address the performance, scalability, reliability and maintainability issues posed by DAS. The SAN is a dedicated network infrastructure that provides meshed, any-to-any connectivity between servers and storage devices.

The introduction of networking concepts and technologies as a replacement of a single, direct connection, redefines the relationship between servers and storage devices and enables the design of new information systems, as depicted in Figure 8.1. Storage resources are now a separated and well-delimited component of the system and servers become the front-end towards the users.

This novel organization of storage resources enables the implementation of new paradigms, providing several benefits [71]:

- **Storage consolidation:** as servers are no-longer directly connected to disks, all the disks can be physically relocated in one or more disk arrays. Disk arrays are devices able to host tens or hundreds of disks. By using the management interface of the

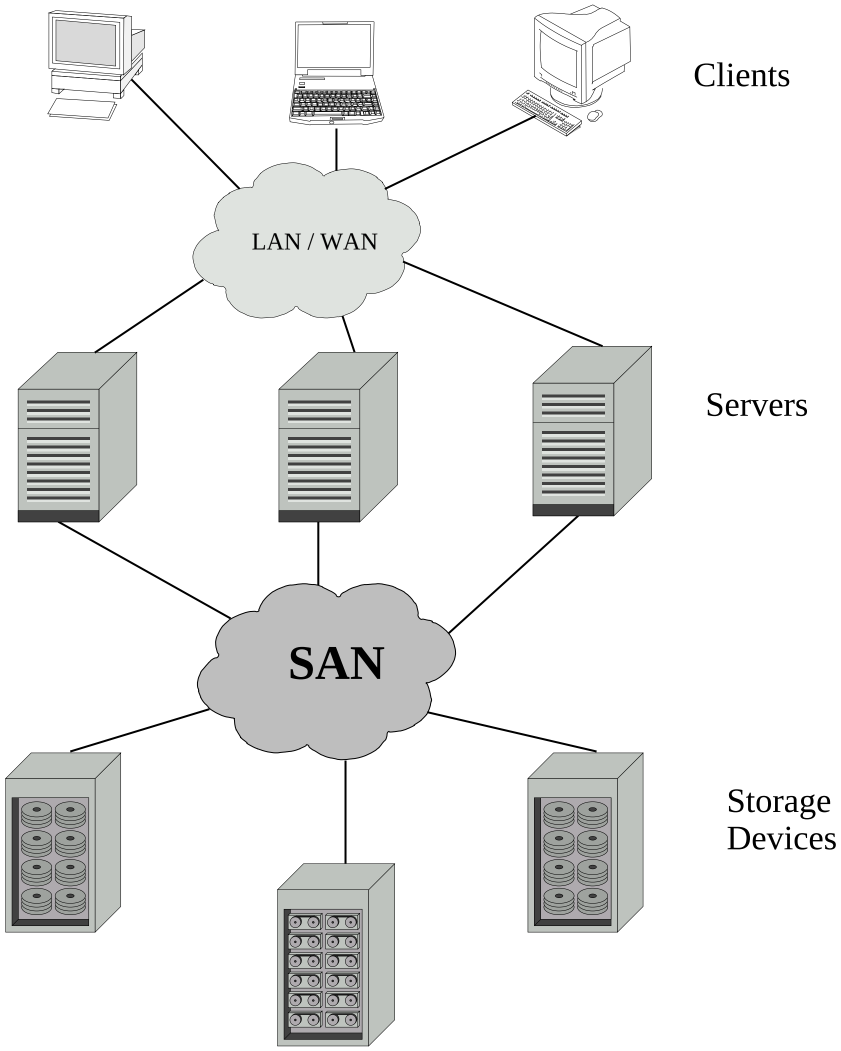


Figure 8.1. An information system employing a SAN

disk array, the storage administrator can allocate to each server a proper fraction of the total capacity. Additional space can be provided without disruption by adding disks to the array and reconfiguring it. Storage consolidation can take place even across multiple disk arrays.

- **Remote replication and disaster recovery:** data can be protected from disk faults by using a technique called “mirroring”. A pool of physical disks of equal capacity is combined in a single, virtual disk of the same capacity. Data written to the virtual disk is physically stored on all the disks in the pool. If any of the disks in the pool fail, data is immediately available on the others and the server can continue its operations without disruption. As a SAN can connect devices located tens of Kilometers away, data can be replicated on remote sites, providing protection even in case of disasters, such as natural calamities or terrorist attacks.

- **Server clustering:** a cluster is a set of servers working concurrently on the same set of data. Clustering provides higher performance (as the servers work in parallel) and higher reliability (if one of the server fails, it simply goes out of the cluster). Although complex issues exist at the operating system and application level (inter-process communication, concurrent data access, etc.) a SAN effectively promotes clustering because it allows easy sharing of common data.

- **LAN-free, server-free backup:** data stored in multiple disk arrays can be backed up directly to large, shared tape drives, without traversing the LAN and without involving the servers. All operations are scheduled and managed from a single, central location.

- **Storage resources management:** the ability to have a consistent and unified view of all the storage devices greatly simplifies monitoring and allocation of resources, as well as provisioning and planning.

In general, the deployment of a SAN enables *virtualization*, i.e. the capability to provide to computing nodes a logical view of available storage resources that is independent of the physical location and the specific characteristics of the devices.

8.3 Networking Technologies for SANs

SANs are networks in all respects and present all the features typical of networking technologies. The most important characteristics inherited from the networking world are:

- serial transport, to ship data over long distances at high rates

- data packetization, to achieve high link efficiency and fair sharing of network resources

- addressing schemes that support very large device population

- routing capabilities, to provide multiple, redundant paths between source and destination devices

- a layered architecture, to support the transport of different protocols at the upper layers and the usage of different interfaces at the lower ones.

SANs can be built using different networking technologies, however, it it is important to remember that servers, operating systems and applications still expect from the storage interface a "channel-like" behavior, i.e. high-speed, low-latency, error-free communications. Networking technologies used to implement SANs must therefore be carefully chosen and deployed in order to satisfy these strict requirements.

Today the preferred networking technology for SANs is Fibre Channel, although different solutions such as iSCSI (based on TCP/IP and Ethernet [72]) or Infiniband [36] have been proposed.

8.3.1 Fibre Channel

Fibre Channel is a multi-purpose, standard-based networking technology, specifically designed for computing environments. Its design is based on the assumption that the transport media (copper cable or optical fiber) is reliable, hence error recovery mechanisms are reduced to a minimum and are mostly left to upper layer protocols. Data are fragmented and encapsulated in network protocols with minimum overhead, in order to achieve high efficiency. Intermediate nodes guarantee that frames will not be discarded, duplicated or delivered out-of-order under any circumstances. A simple, credit-based mechanism is used for flow and congestion control. These characteristics of the data-path make a full hardware-based implementation feasible. Incoming frames can be processed by end nodes at very high speed and do not incur the latency induced by large reassembly and reordering buffers.

8.3.2 Credit-based flow control

Flow control mechanisms are used to regulate the rate at which a transmitter sends frames, in order to achieve efficient bandwidth utilization without overwhelming the receiver. These mechanism represent one of the most important characteristics of a networking technology and have a very strong influence on the design of network devices.

In Fibre Channel networks flow control mechanisms are based on the concept of *credit*. A credit represents the ability of a receiver to accommodate one frame. The receiver grants to the transmitter an initial number of credits, typically proportional to the size of its buffers. The transmitter is authorized to send one frame for each credit it has received; after that it has to stop until it receives more. As soon as the receiver has finished processing an incoming frame (for instance, it has passed it to upper layers) it can free the resources that were used by that frame and grant a new credit.

Fibre Channel provides two levels of flow control: "buffer-to-buffer" and "end-to-end". Buffer-to-buffer flow control takes place between pairs of adjacent ports, such as a link between a node and a switch or between two switches. It operates on all the packets traversing the link, without the capability to discriminate among multiple flows. End-to-end flow control, on the contrary, operates only between end-nodes and is performed per

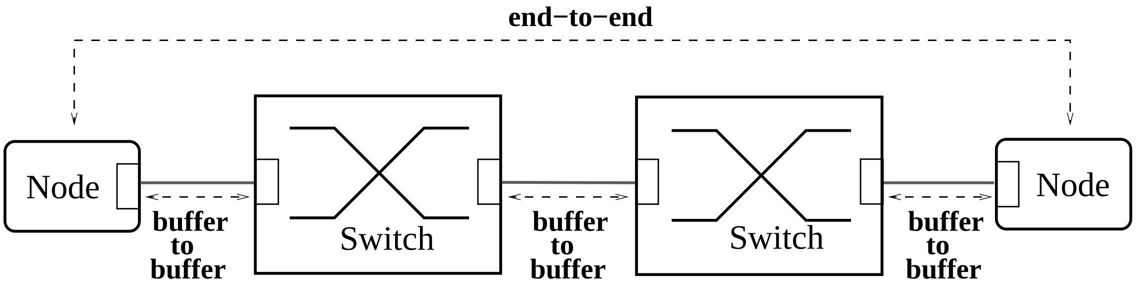


Figure 8.2. Flow control levels

flow, i.e. if a node is receiving multiple flows, it controls each of them them separately. The two levels are illustrated in Figure 8.2.

Credit-based flow control mechanisms guarantee that a device accepts incoming frames only if it has the resources to service them. Switches can use such mechanisms to regulate incoming traffic, but once they have accepted a frame, they are not allowed to drop it.

Chapter 9

The Switching Architecture

In this chapter we present the architecture of a CIOQ switch specifically designed for Fibre Channel SANs. The main differentiation points with respect to traditional LAN switches and Internet routers are the asynchronous design, the addition of a centralized arbiter and the employment of a number of buffer management techniques that guarantee loss-free operation. We first present the base architecture, considering unicast traffic only, and then discuss extensions to support multicast.

9.1 System Overview

The logical architecture of the system is depicted in Figure 9.1. It is composed by a set of N_{LC} linecards interconnected by a crossbar-based switching fabric. Each linecard hosts P_{LC} input/output ports and has two links to the switching fabric, named *uplink* and *downlink*. Packets enter the linecard through input ports and are multiplexed on the uplink. They traverse the fabric and are transmitted to the proper output linecard on the downlink. After demultiplexing, they finally reach the destination output port. The bandwidth of the uplink and the downlink is equal to the sum of the bandwidths of the input/output ports hosted on a linecard, so they are not oversubscribed and do not constitute a bottleneck.

The system is fully asynchronous, i.e. the linecards and the switching fabric run on independent clock domains. This feature provides several benefits in terms of simplicity, cost and scalability [4]. First, it prevents the necessity to maintain and distribute a global clock signal, a task that can be problematic, especially if modules are distributed among multiple racks. Second, it enables native support for variable-length packets, eliminating the need for segmentation and reassembly buffers. Finally, it allows simplified arbitration of the switching fabric, without employing complex scheduling algorithms. These advantages, however, come at a price: buffers are needed at the fabric inputs and outputs, and moderate speed-up is required to achieve good performance.

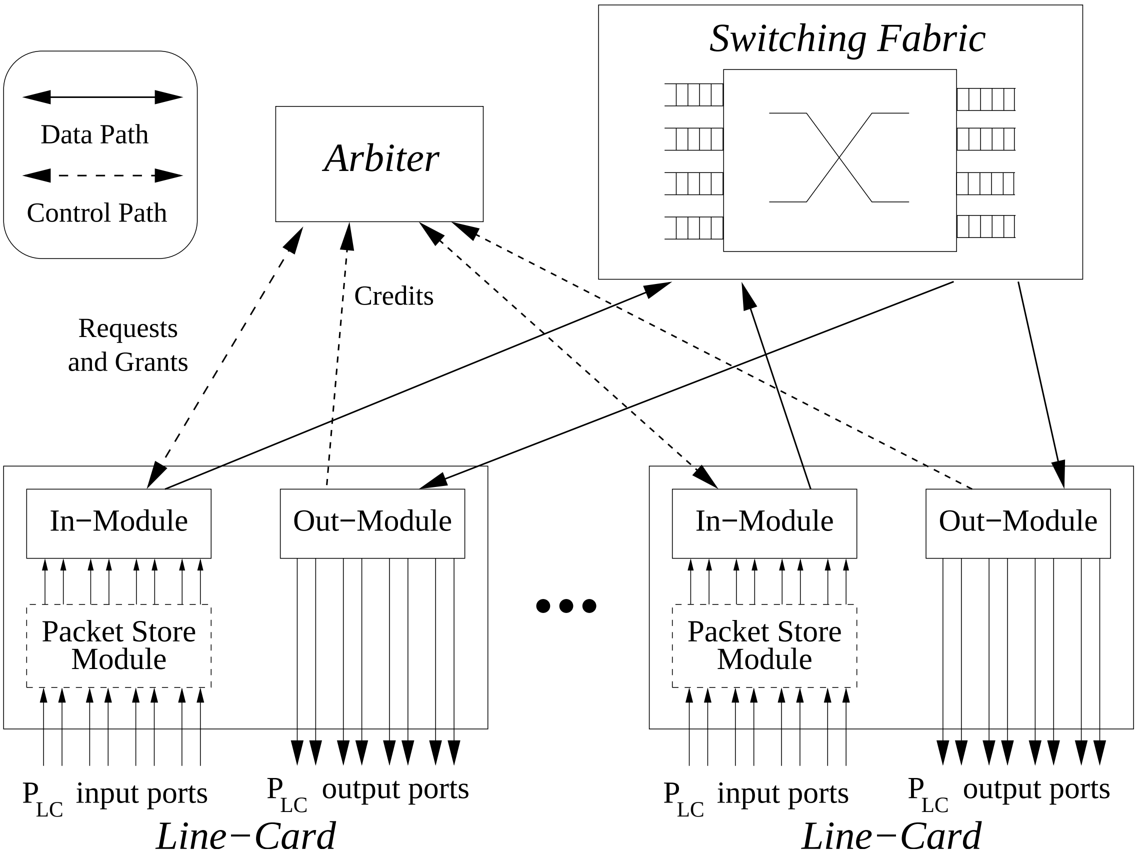


Figure 9.1. Logical architecture of the switch, showing two linecards

To achieve lossless operation, all buffers are endowed with a *backpressure* mechanism, that allows them to block transmission from previous stages when they are close to saturation. The signal is activated when occupancy grows above a certain threshold and is deactivated when it decreases below another threshold.

This form of flow-control is sufficient to prevent buffer overflow, but is too coarse, as it does not discriminate among multiple flows traversing the switch. Therefore the system employs an additional centralized mechanism that guarantees fairness among multiple flows and enhances performance.

9.2 Data Path

9.2.1 Linecards

The Packet-Store Module (PSM) is a large buffer that stores packets entering on input ports[1]. Memory is divided in slots of equal size, dimensioned to contain a maximum-size packet[2]; if a smaller packet is stored, the remaining part of the slot is unusable. Memory segmentation reduces usage efficiency but simplifies the implementation of buffer management schemes. Total capacity is statically partitioned among input ports. If the space assigned to a specific port is completely used, the port uses buffer-to-buffer flow control (Section 8.3.2) to inhibit transmission from the connected node. Buffer space is logically organized to provide a separate set of VOQs to each input port. Ports select independently and in parallel packets to be transferred to the In-module using a round-robin policy.

The In-module is a fast and small random-access memory, that acts as a high-speed interface towards the switching fabric. It contains a small number of fixed-size slots, organized as a single set of VOQs. Access to the In-module memory is regulated by a buffer management mechanism that prevents input ports from monopolizing available space. Moreover, if a VOQ in the In-module grows beyond a specific size, no more packets from the corresponding destination are accepted from the PSM VOQs. When all In-module slots are occupied, a backpressure signal blocks any packet transfer from the PSM.

The Out-module receives from the switching fabric the aggregate flow of packets directed to the linecard and demultiplexes it based on the destination port. Memory is segmented in fixed-size slots and structured as a set of P_{LC} queues, one for each output port on the linecard. Queues cannot overflow because space is pre-allocated using the credit-based flow-control mechanism described in Section 9.3.1. The logical layout of a linecard, highlighting queueing stages, is shown in Figure 9.2.

[1] In a practical implementation, PSM functionalities would be spread among multiple chips, each serving a subset of the input ports.

[2] MTU is 2 KB for Fibre Channel devices.

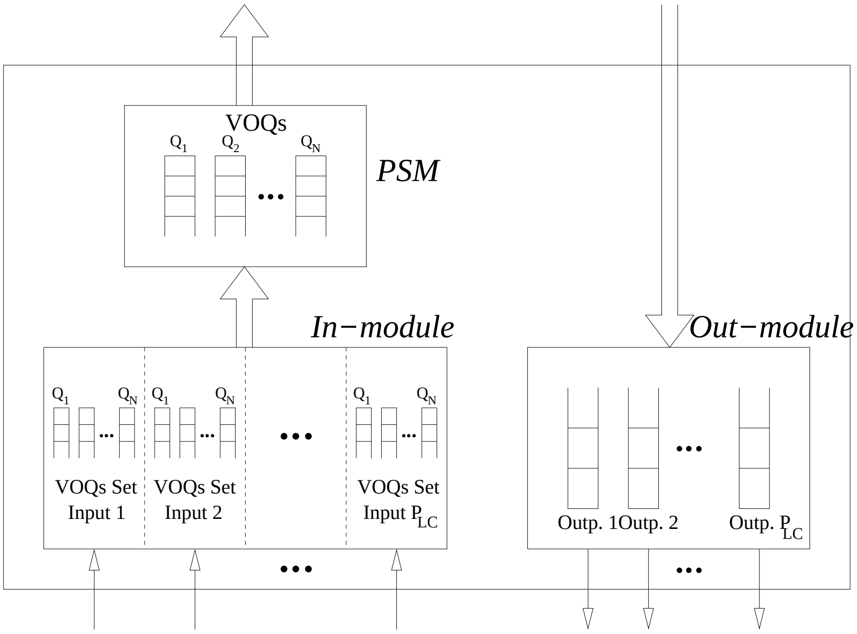


Figure 9.2. Logical layout of a linecard

9.2.2 Switching fabric

The switching fabric is based on a $N_{LC} \times N_{LC}$ crossbar. As the fabric runs asynchronously with respect to the linecards, buffers are required at the input and output *fabric ports*. The size of these buffers is determined by the Round-Trip Time (RTT) between the linecards and the fabric. In principle, if the RTT is negligible, a single MTU-sized buffer suffices. In practice it is necessary to have more, to take into account store-and-forward delay and other sources of overhead that contribute to the RTT. Moreover, it is desirable to support multi-rack configurations, in which the RTT is large due to propagation delay. The actual size of these buffers is chosen to be on the order of few tens of MTUs per port, depending on the constraints imposed by chip technology. Packets are stored contiguously in memory (i.e. there is no segmentation) to maximize efficiency. Buffer overflow is prevented using a backpressure signal that blocks transmission from the linecard when space is exhausted.

As the available amount of memory is small and the clock frequency high, it is not possible to implement VOQs at the fabric input ports. Each input buffer is organized as a single FIFO queue, hence the fabric suffers from HOL-blocking. This phenomenon can severely impact throughput [8] but its effects can be partially mitigated by providing a moderate speed-up K.

Buffers at the fabric output ports receive packets at K times the rate of fabric input ports, so they must be able to store at least K MTUs. In order to sustain temporary overload conditions, their size is chosen to be K times the size of the input buffers.

At the head of multiple fabric input queues there might be packets directed to the same linecard. Each crossbar output can receive packets from only one crossbar input at a time, hence arbitration is necessary to resolve the contention. This task is performed by simple *fabric arbiters*, one for each crossbar output, that select a crossbar input among the contending ones in a round-robin fashion. Fabric arbiters work independently and in parallel and do not need to perform multiple iterations. Note that this is much simpler than solving a bipartite-graph matching problem.

If persistent overload conditions cause a fabric output queue to fill up, the corresponding fabric arbiter does not allow any new transmission until queue occupancy goes below a given threshold. This forms of backpressure prevents packet losses inside the fabric.

9.3 Control Path

In a Fibre Channel network, nodes make explicit use of buffer-to-buffer flow-control to regulate incoming traffic (Section 8.3.2). A switch output port can be blocked by an adjacent node that, due to congestion, is not able to accommodate new packets and stops releasing credits. In this situation the switch stores packets in its internal buffers. If congestion persists, buffers eventually fill up and the switch blocks incoming flows directed to the congested node. It is important to make sure that blocked flows do not interfere with others that are traversing the switch.

Interference is potentially caused by the sharing of switch resources among multiple ports. In particular, the switching fabric handles aggregates of flows that come from or are directed to the same linecard and has no notion of input and output ports. It cannot selectively block flows directed to a specific output port without blocking at the same time those directed to other ports on the same linecard.

9.3.1 Internal flow-control

To isolate congested flows, the switch uses a mechanism that operates at the (input port, output port) granularity level. This mechanism, named “internal flow-control” is managed by the central arbiter, that acts as a “bridge”, effectively extending across the switching fabric the buffer-to-buffer-flow control performed at the input and output ports.

Internal flow-control operates in the following phases, as depicted in Figure 9.3:

1. The Out-module sends a *credit* to the central arbiter to signal a free, MTU-sized slot in a port queue.

2. The In-module sends to the central arbiter a *request* to transmit to the switching fabric a packet destined to a specific output port.

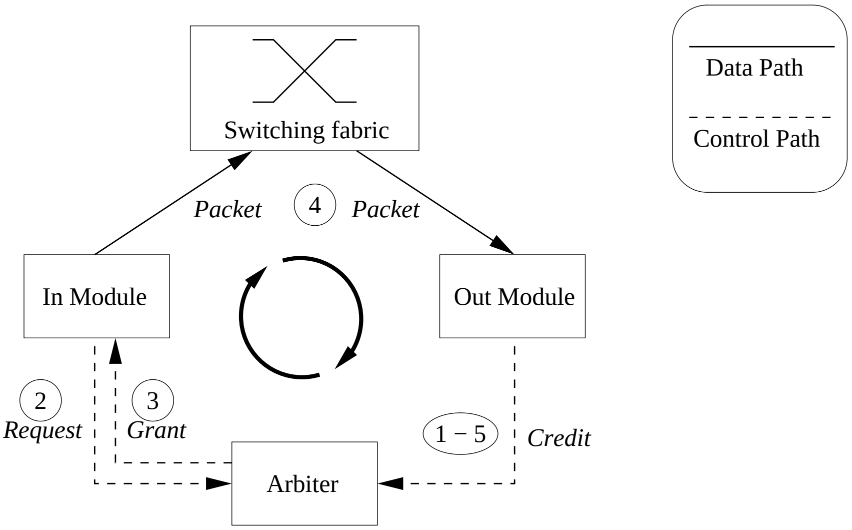


Figure 9.3. Credit-based flow control loop managed by the central arbiter

3. When the central arbiter finds an output port for which a credit is available and an ungranted request exists, it sends a *grant* to the requesting In-module.
4. The In-module transmits the packet, which crosses the switching fabric and arrives to the destination linecard, where it is stored in the Out-module.
5. When the packets is finally transmitted on the output link, the Out-module frees the buffer and returns the credit to the arbiter.

Thanks to this mechanism, a packet enters the switching fabric only if there is space to store it in the destination output port's queue. Queues cannot overflow, therefore no backpressure is required from the Out-module to the switching fabric. If an output port is blocked, the queue fills up, the arbiter runs out of credits and In-modules do not receive grants to transmit to that port. Buffer space inside cannot be monopolized by packets waiting to be transferred to the blocked port, because the buffer management technique described in section 9.2.1 limits the number of packets directed to a specific destination that can be present in the In-module at the same time.

The term "credit" has been used to refer to control messages used both by buffer-to-buffer and internal flow-control. While the two mechanisms are clearly distinct, the semantic of the term is the same in both contexts. A credit represents the capability of the receiver (whether it is a network node or a switch output port queue) to store a frame of maximum size or less, so the terminology we have introduced is consistent. From this point on, however, our discussion will focus on the internal operations of the switch, so we will always implicitly refer to internal flow-control and its control messages.

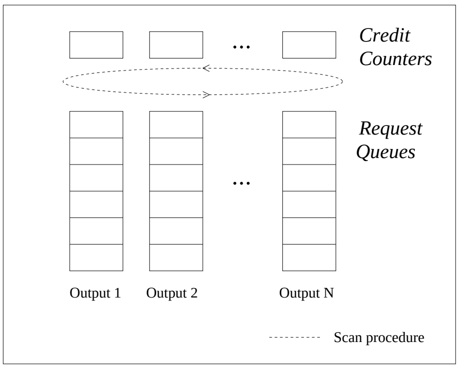


Figure 9.4. Logical architecture of the central arbiter

9.3.2 The central arbiter

The central arbiter explicitly authorizes the transmission of every packet, so it must operate at very high rate and provide fair treatment to input ports.

The internal structure of the central arbiter is depicted in Figure 9.4. For each output port there is a credit counter and a request queue. The counter is incremented every time a credit for the output port is received. Requests for the output port are are stored, in FIFO order, in the request queue. The arbiter continuously scans, in round-robin order, the credit vector and the requests queues, looking for output ports that have both an available credit and an ungranted request. When a match is found, the credit counter is decremented, the request dequeued and the grant sent to the requesting In-module.

The maximum number of requests pending for an output port corresponds to the maximum number of packets directed to that output port that can be present in the In-modules at the same time. This number is limited by the buffer management policy employed at each In-module. The request queues are dimensioned to be able to host the maximum number of pending requests, so their occupancy doesn't have to be controlled.

9.4 Extension to Support Multicast Traffic

The architecture we have presented can be suitably extended to support multicast traffic. This part of the design is not finalized yet, so at some point we will consider multiple solutions, whereas at others we will neglect some issues, specifically those related to the partitioning of system resources (internal and external links, buffers, etc.) between unicast and multicast.

The delivery of a multicast packet entails two phases:

1. the packet must be replicated to all the linecards that host one or more destination ports,

2. on each linecard, the packet must be replicated to all the intended output ports.

To minimize the load on the uplink, a single copy of each multicast packet is transmitted from the linecard to the switching fabric, and replication to multiple linecards is performed in the switching fabric. Conversely, to reduce the load on the downlink, a single copy of the packet is sent to each intended linecard, and replication to multiple output ports is performed independently on each linecard.

9.4.1 Linecards

Linecard buffers can simply be augmented with dedicated space for multicast packets. The PSM provides to each input port a separate portion of memory, organized as a FIFO queue. This choice naturally leads to HOL blocking, but is dictated by the fact that it is not practically feasible to sort incoming packets based on their fanout set [30]. When an input port fully occupies its share of the PSM memory, transmission from the adjacent device is inhibited using buffer-to-buffer flow-control.

The In-module collects packets from the PSM queues in round-robin order and stores them in a single FIFO queue. Each input port has a limit on the amount of In-module memory that it can occupy. When this limit is reached, the In-module stops servicing the input port's PSM queue, until occupancy decreases below a certain level. This mechanism guarantees that individual inputs cannot monopolize the In-module memory. Both the PSM and the In-module memory is segmented to simplify buffer management.

The Out-module stores multicast packets received from the switching fabric in a buffer organized as a single FIFO queue. When a packet reaches the head of the Out-module queue, it is replicated to all the output ports it is destined to and dequeued. The replication process is instantaneous and does not delay packet transmission on output links. If internal flow-control is used, memory must be segmented, because it is not possible to know in advance the size of a packet that will be received, so an MTU-sized slot must be set aside. On the contrary, if memory is not pre-allocated, packets can be stored contiguously, to make more efficient usage of available space. In this case backpressure towards the fabric output queues is necessary to avoid overflow.

9.4.2 Switching fabric

Multicast packets entering the switching fabric are stored in fabric input queues, together with unicast packets or in a separated space. When a packet reaches the head of its queue, it must be replicated to multiple linecards. The crossbar is equipped with internal multicasting capability, meaning that it can replicate a packet to multiple outputs at the same time with no extra cost. By using this feature it is possible to reduce packet delays and fabric input queues occupancy; however, doing so requires multiple outputs to be free at the same time. Waiting to gain access to all the intended outputs before transmitting a packet can be counterproductive, because it forces outputs that have already granted access to stay idle while the others become free.

To exploit the benefits of crossbar replication without compromising efficient usage of output ports, fabric inputs transmit packets in multiple phases:

1. the input requests all the outputs included in the fanout set of the packet at the head of the fabric queue, and starts a timer T_o;

2. it sends “in a single shot” the packet to all the outputs that have immediately granted access;

3. afterwards, the input individually sends to each remaining output a copy of the packet as soon as it grants access;

4. when the timer T_o expires, the packet is dequeued and dropped, even if not all the intended destinations have been reached.

This final drop decision can be optionally skipped for packets that need more reliable delivery (T_o set to ∞).

9.4.3 Central arbiter

In section 9.3 we have illustrated the benefits achieved by controlling individually unicast flows. The same result is more difficult to obtain for multicast, because the number of possible flows traversing the switch grows exponentially (rather than quadratically) with N. This implies that no switch resource can be assigned per-flow. In particular, both on the ingress and egress side of linecards packets are stored in a single FIFO queue, regardless of their fanout. As a consequence, internal flow-control cannot provide differentiated treatment to multicast flows.

How to effectively isolate congesting multicast flows, using a reasonable number of queues and an implementable arbiter, remains an open issue.

Chapter 10

Performance Under Unicast Traffic

In this chapter we study by means of simulation the performance of the switch presented in Chapter 9. To obtain and discuss our simulation results we refer to a specific implementation, with a realistic choice of system parameters. We refer to a 256×256 system offering 512 Gbps of aggregate bandwidth [6].

The goal is to understand the effects of the flow control and backpressure mechanisms under different traffic patterns. We are particularly interested in observing system performance as the number of available credits per output port varies.

10.1 Simulation model

The simulator we have developed models all the system components described in Chapter 9, together with flow-control, backpressure and buffer management techniques. It explicitly takes into account the transmission times of packets on output links and transmission times of control messages (credits, requests and grants) on control links. Transmission times are only due to store-and-forward delays, as propagation delays are not considered.

In our experiments we assume that output ports absorb traffic at line-rate, i.e. they do not receive blocking signal from downstream devices (end-nodes or other switches).

The simulator samples system evolution at regular intervals, called "timeslots". The duration of a timeslot T_s is equal to the length of the shortest event; all events are assumed to take an integer number of timeslots.

10.2 Simulation settings

Table 10.1 summaries the settings adopted in the simulation.

Parameter	**Symbol**	**Value**
Input - Output ports		
Input-Output ports per linecard	P_{LC}	16
Linecards in the system	N_{LC}	16
Total number Input-Output ports	N	256
Link speeds for data and signaling		
Input - Output ports (data path)	V_P	2 Gbps
Linecard ↔ crossbar (data path)	V_X	32 Gbps
Linecard ↔ central arbiter (control path)	—	2 Gbps
Packet size		
Minimum packet dimension	—	64 bytes
Maximum packet dimension	MTU	2048 bytes
In-module & Out-module		
Total PSM buffer size	—	4000 MTU
In-module shared buffer size	—	100 MTU
Number of credits per output	X	*variable*
Switching Fabric		
Internal speed-up	K	2
Input fabric buffer size	—	20 KBytes
Output fabric buffer size	—	40 KBytes

Table 10.1. Summary of the main architecture parameters

10.3 Traffic model

Open-loop source models, such as Bernoulli or on-off/bursty, traditionally used to analyze the performance of lossy packet-switching systems, are not suitable for our study. A Fibre Channel source receives control information from the network through buffer-to-buffer flow-control (Section 8.3.2). When the source is blocked by flow control, it stops transmitting on the link and starts accumulating packets in the output link queue. When it is allowed to restart transmission, it has a burst of packets waiting to be sent. However, if a source injects traffic at line-rate and packets are uncorrelated, the transmitter queue can be neglected. In this situation the source transmits packets back-to-back anyway, so accumulated packets would not make any difference.

Traffic matrix $\tilde{\Lambda} = [\tilde{\lambda}_{ij}]$ represents the rate at which source i *generates* packets directed to output j, whereas $\Lambda = [\lambda_{ij}]$ represents the rate at which packets actually enter the switch; obviously $\lambda_{ij} \leq \tilde{\lambda}_{ij} \forall i,j$. As we assumed that sources transmit at line-rate,

$\sum_j \tilde{\lambda}_{ij} = 1$ and admissibility conditions further require that $\sum_i \tilde{\lambda}_{ij} \leq 1$.

In stationary conditions λ_{ij} also represents the rate at which traffic from input i to output j exits the switch, so we can define throughput for port j as $\sum_i \lambda_{ij}$.

We consider three distributions of the packet size:

- minimum size (64 bytes) only,
- maximum size (2048 bytes) only,
- uniform between 64 and 2048 bytes, with 64 bytes increment

10.4 Diagonal traffic

In this scenario each input port only transmits to itself: $\tilde{\lambda}_{ij} = 1$ if $i = j, 0$ otherwise. This traffic pattern allows us to observe system behavior when there is no contention in the switching fabric and system dynamics are dominated by the flow control mechanisms.

As depicted in Figure 9.3, internal flow control effectively represent a closed-loop control systems. The control loop delay (i.e. the time that elapses between the moment the arbiter consumes a credit and the moment the Out-module returns it) is non-negligible and includes store-and-forward delay of control messages, the transmission time of a packet on an output link as well as additional delays introduced by system components.

A minimum number of credits is required to compensate for the control loop delay and achieve line-rate. Consider for example the case in which a single flow is traversing the switch. If only 1 credit were available, an output port, after finishing the transmission of that packet, would have to stay idle waiting for the next one to be transferred from the In-module to the Out-module.

Let T_{LOOP} be the control loop delay and T_{PK} the transmission time of a minimum-size packet on an output link: then in a period of time equal to T_{LOOP} the switching fabric must be able to transfer at least $\lceil T_{LOOP}/T_{PK} \rceil$ packets. T_{LOOP} is the sum of three terms:

- the time required to send a credit from the Out-module to arbiter and a grant from the arbiter to the In-module (T_{CR});
- the time required to transmit a packet through the switching fabric, from the In-module to the Out-module (T_{SW});
- the transmission time of a packet on an output link (T_{PK}).

Note that the first term is constant, whereas the other two depend on the packet size.

We assume T_{CR} is equal to 64 ns, broken down as:

- 16 ns of processing time in the Out-module,

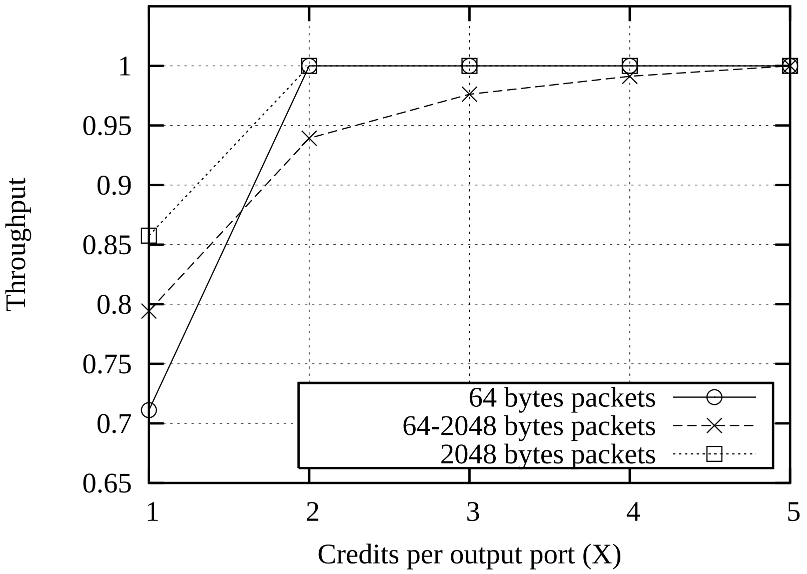


Figure 10.1. Throughput under diagonal traffic as a function of number of credits per output port

- 16 ns of transmission time to the arbiter,
- 16 ns of processing time in the arbiter (this term is constant because no other input is contending for the same output) and
- 16 ns of transmission time to the In-module.

T_{SW} in turn is the sum of three terms, that account for packet transmission on the uplink, through the switching fabric and on the downlink. Overall, considering crossbar speed-up K, $T_{SW} = (2 + 1/K)L_{PK}/V_X$. The transmission time on an output link is simply $T_{PKT} = L_{PK}/V_P$.

Having evaluated control loop delay, we can calculate the minimum number of credits needed to achieve 100% throughput. Moreover, we can calculate the maximum throughput for $X = 1$ and fixed-size packets, because in these conditions $1/T_{LOOP}$ represents the arrival rate of packets at the Out-module:

$$T_{LOOP} = T_{PK} + T_{CR} + T_{SW} \tag{10.1}$$

$$\lambda = \frac{T_{PK}}{T_{LOOP}} = \frac{T_{PK}}{T_{PK} + T_{CR} + T_{SW}} \tag{10.2}$$

Figure 10.1 shows average throughput for different values of X and different packet-size distributions.

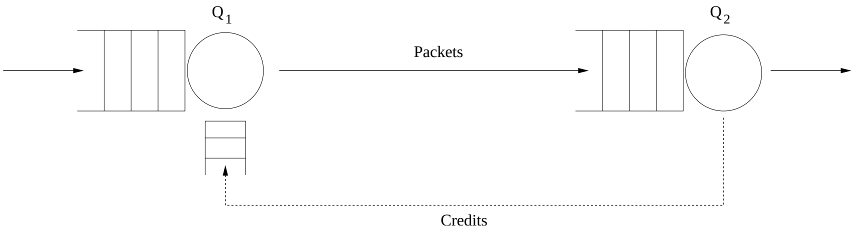


Figure 10.2. Simplified queueing model to study performance under variable-size packets.

10.4.1 Small packets

This scenario is the most critical because small packets lead to small transmission times, whereas control path overhead is constant.

With 64 bytes packets, $T_{PK} = 256$ ns , $T_{SW} = 40$ ns and $T_{LOOP} = 360$ ns . From the above formula we expect throughput with $X = 1$ to be equal to 71% and to reach 100% with $X = 2$. Figure 10.1 confirms the results.

10.4.2 Large packets

In this scenario $T_{PK} = 8192$ ns , $T_{SW} = 1280$ ns and $T_{LOOP} = 9536$ ns . For $X = 1$ throughput is 85.9% and for $X = 2$ it is 100%.

10.4.3 Variable-size packets

Referring again to Figure 10.1, we observe that with variable-size packets 5 credits are required to achieve 100% throughput.

This throughput reduction is due to the fact that variable-sized packets may introduce a mismatch between the rate at which packets arrive at the In-module and the rate at which credits are released at the Out-module, forcing some packets to wait at the In-module.

Consider the simplified model of the flow-control mechanism shown in Figure 10.2. Queue Q_2 models the Out-module transmitting on the output link, while queue Q_1 models the In-module transmitting to the switching fabric. Assume that only two classes of packets are present: large, of size L and small, of size $\alpha L, \quad \alpha < 1$. Q_2 serves packets at rate R_2 bits/s, so it is capable of servicing R_2/L large packets/s or $R_2/(\alpha L)$ small packets/s. Whenever it has finished servicing a packet, it releases a credit to queue Q_1. Q_1 serves packets at rate $R1 \gg R2$ bit/s, however, it can start packet service only if it has a credit in its buffer. Therefore, the rate at which it can serve *packets* is equal to the rate at which it receives credits. We ignore control-path latencies, so credits released at queue Q_2 are immediately available to queue Q_1.

If we look at instantaneous packet service rates at Q_1 and Q_2, four cases are possible:

1. Small packets at the heads of both Q_1 and Q_2: Q_2 services packets and releases credits at rate $R_2/(\alpha L)$. The rate is sufficient for Q_1 to serve enqueued packets, so no backlog accumulates and maximum throughput is achieved.

2. Large packets at the heads of both Q_1 and Q_2: this case is similar to the previous one, the only difference being the credit release rate equal to R_2/L. Maximum throughput is achieved again.

3. Large packets at the head of Q_1 and small packets at the head of Q_2: Q_2 releases credits at rate $R_2/(\alpha L)$, which is larger than the rate needed by Q_1 to service incoming packets, so maximum throughput is achieved.

4. Small packets at the head of Q_1 and large packets at the head of Q_2: in this case Q_2 releases credits at rate R_2/L, which is smaller than the rate needed by Q_1 to service incoming packets. Basically, large packets with a long transmission time on an output link are holding credits needed by small packets at the In-module. The rate at which new packets are enqueued at Q_1 is $R_2/L(1/\alpha - 1)$.

The situation described in case number 4 leads to throughput reduction, unless the credit buffer is large enough to accumulate the excess credits issued in case number 3 and provide compensation. This is again in accordance with the results shown in Figure 10.1.

The conclusion that we can draw using this simple model is that under any monomodal distribution of packet size, to achieve maximum throughput is only necessary to compensate for the control-loop delay, whereas under variable size packets, more credits are needed to compensate for temporary mismatch between the rates at which packets enter and exit the switch.

10.5 Uniform traffic

In this scenario packets entering the input ports are destined to all output ports with equal probability ($\tilde{\lambda}_{ij} = 1/N \,\forall\, i,j$), so there is significant contention for system resources.

The main contention points are in the In-module, where packets must wait for a credit in order to access the switching fabric, and at the crossbar inputs, where packets from a linecard compete with packets from other linecards to reach a crossbar output port.

If a packet is blocked in the In-module because it lacks the credit needed to proceed, we say that it is experiencing *starvation*; if, instead, it is blocked because the In-module is receiving backpressure from the congested fabric input queue, we say that the packet is experiencing backpressure.

Both circumstances can degrade final throughput, so our aim is to understand under what conditions they occur. We start with a qualitative discussion of the impact of X and then we proceed with a more systematic analysis of simulation results.

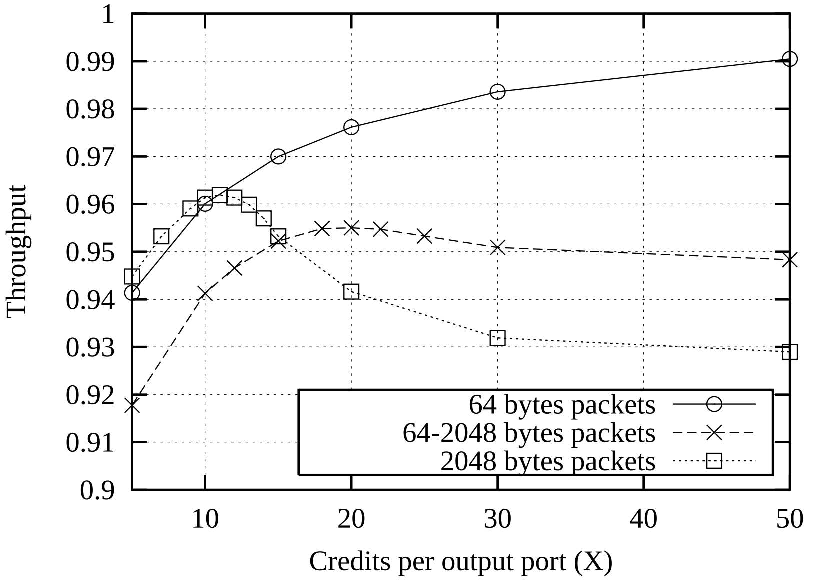


Figure 10.3. Throughput under uniform traffic and different packet size distribution

10.5.1 The effects of internal flow-control

Understanding the effects of internal flow-control under a generic traffic pattern and choosing the optimal value X is quite complex, because many implications must be taken into account.

As discussed in the previous section, X must be large enough to to cover for the control-delay loop, which becomes larger when there is contention in the switching fabric. Moreover, the larger X, the longer the switch can sustain temporary overload of specific output ports, without blocking packets at the In-module due to starvation.

However if X is small, it is less likely that fabric input queues saturate, leading to backpressure. In general backpressure is more harmful than starvation because it affects all active flows entering on a linecard, whether they are congested or not. Besides, a small X helps in reducing HOL-blocking, because it shapes traffic entering the switching fabric. In the switching fabric there can be up to X packets directed to a specific output port, therefore there can be up to $P_{LC}X$ packets directed to a specific linecard. The lower this number, the lower the probability that two packets destined to the same linecard arrive at the head of different fabric input queues and collide.

Figure 10.3 shows system throughput for different values of X under uniform traffic, with fixed- and variable-size packets. The first thing to note is that for all packet-size distributions the number of credits required to achieve maximal performance is in the order of 10's, so significantly higher than under diagonal traffic. This is due to the contention in the switching fabric, that increases the control-loop delay.

Cre-dits	**Thru-put**	**Pkt. block prob.**		**Av. fabric FIFOs occ.** (Bytes)	
		St.	Bp.	Inputs	Outputs
5	0.941	0.059	0	90.88 (0.44%)	283.52 (0.69%)
10	0.960	0.040	0	96.00 (0.47%)	460.8 (1.1%)
15	0.970	0.030	0	98.56 (0.48%)	646.4 (1.6%)
20	0.976	0.024	0	100.48 (0.49%)	972.8 (2.4%)
30	0.984	0.016	0	103.04 (0.50%)	1273.6 (3.1%)
50	0.990	0.010	0	104.96 (0.51%)	2227.2 (5.4%)

Table 10.2. Stationary results under uniform traffic and 64 bytes packets

10.5.2 Small packets

Table 10.2 reports additional details, namely average fabric queue occupancy and average packet blocking probability, due to starvation (“St.”) or backpressure (“Bp.”). If an In-module is receiving backpressure from the fabric input queue, this is accounted as backpressure, regardless of the actual availability of credits. Starvation only represent the case in which the packet is blocked by lack of credits and backpressure is not active.

Thanks to the small size of the packets and the fact that fabric memory is not segmented, queues occupancy remains low and backpressure towards the In-module is never activated. Starvation is the only cause of throughput degradation, and decreases as the number of available credits grows. However, as X grows more packets are present in the switching fabric at the same time and control-loop delay grows as well, so the benefits provided by additional credits are progressively reduced.

10.5.3 Large packets

With large packets, the situation is significantly different: throughput increase until $X = 10$, then it starts decreasing and keeps going down as X grows. By looking at Table 10.3 we realize that this is mainly due to backpressure, which becomes the only cause of throughput degradation for $X > 10$. Fabric input queues have limited size and can only host about ten packets each. With more credits available, they quickly fill up and backpressure blocks packets at the In-modules. Fabric output queues occupancy grows until $X = 15$, then it starts decreasing, even if fabric input queues occupancy keeps growing. This is due to the fact that as X grows the shaping effect of internal flow-control is reduced and packets experience higher contention in the switching fabric.

Cre-dits	**Thru-put**	**Pkt. block prob.**		**Av. fabric FIFOs occ.** (Bytes)	
		St.	Bp.	Inputs	Outputs
5	0.945	0.055	0	1542 (7.5%)	8058 (20%)
10	0.961	0.030	0.009	3220 (16%)	14185 (35%)
15	0.953	0	0.047	7000 (34%)	16235 (40%)
20	0.942	0	0.058	7522 (37%)	15468 (38%)
30	0.932	0	0.068	7922 (39%)	14929 (36%)
50	0.929	0	0.071	8032 (39%)	14833 (36%)

Table 10.3. Stationary results under uniform traffic and 2048 bytes packets

Cre-dits	**Thru-put**	**Pkt. block prob.**		**Av. fabric FIFOs occ.** (Bytes)	
		St.	Bp.	Inputs	Outputs
5	0.918	0.082	0	1210 (5.9%)	6541 (16%)
10	0.941	0.056	0.003	1794 (8.8%)	10448 (26%)
15	0.952	0.033	0.014	3373 (16%)	12824 (31%)
20	0.955	0.013	0.031	5222 (25%)	14271 (35%)
25	0.953	0.003	0.044	6326 (31%)	14585 (36%)
30	0.951	0	0.049	6700 (33%)	14455 (35%)
50	0.948	0	0.052	6948 (34%)	14238 (35%)

Table 10.4. Stationary results under uniform traffic and variable-size packets

10.5.4 Variable-size packets

Table 10.4 show that the values of system metrics in this scenario are intermediate with respect to the two cases presented above. For $X < 15$ throughput with variable sized packets is lower than with fixed size packets (either small or large) because of the mismatch between credit release rate and packet arrival rate, as explained in Section 10.4. As X grows we see starvation gradually disappearing but backpressure showing up, with the net result that for any X (except very small or very large values) both are present at the same time. The final curve is the result of the combined effects of all the phenomena described before. Overall throughout reaches its maximum for $X = 20$, where it is equal to 95.3%, and then decreases very gently for larger values.

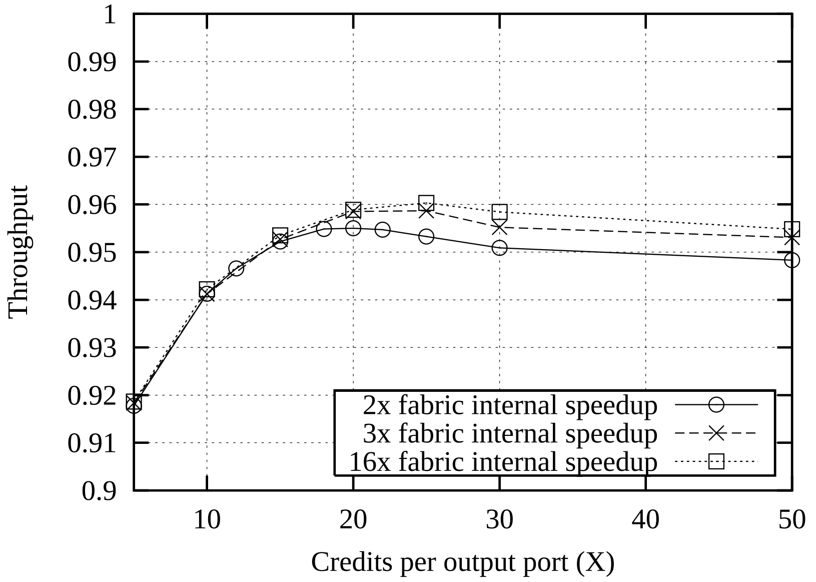


Figure 10.4. Throughput with increased fabric speed-up under uniform traffic and variable-size packets

10.6 Improving system performance

System performance is satisfactory in all the scenarios tested so far, throughput being greater than 92% under any packet size distribution for $X > 5$. Starting from the results obtained previously we try to understand on which parameters it is worth acting to further improve performance.

10.6.1 Increased internal speed-up in the switching fabric

We first try to vary crossbar speed-up to understand how much HOL-blocking penalizes performance. Results are reported in Figure 10.4 and Table 10.5 for variable-size packets. Note that the case $K = 16$ corresponds to an output-queued switch, in which HOL-blocking and output contention are completely eliminated.

We see that $K = 3$ brings small improvement, whereas $K = 16$ almost no improvement. We can deduce that HOL-blocking in the switching fabric has a very small impact on system performance. The reason is that with very high speed-up and small X system performance is limited by starvation, whereas with large X fabric output queues fill up rapidly and activate backpressure towards the fabric input ports. If a fabric output port is saturated, it cannot accept any new packet and speed-up becomes useless.

A comparison of Table 10.5 and Table 10.4 confirms that fabric output queues occupancy increases, whereas fabric input queues occupancy decreases. This leads to a

Cre-dits	**Thru-put**	**Pkt. block prob.**		**Av. fabric FIFOs occ.** (Bytes)	
		St.	Bp.	Inputs	Outputs
5	0.919	0.081	0	111.1 (0.5%)	6746 (16%)
10	0.942	0.055	0.003	275.0 (1.3%)	10838 (26%)
20	0.959	0.013	0.028	1971 (9.6%)	15413 (38%)
25	0.960	0.001	0.039	2802 (14%)	16497 (40%)
30	0.958	0	0.042	3331 (16%)	16742 (41%)
50	0.955	0	0.045	3690 (18%)	16901 (41%)

Table 10.5. Stationary results under uniform traffic and variable-size packets, for $K = 16$

small reduction of backpressure but leaves blocking probability due to starvation almost unchanged.

10.6.2 Extended memory size in the switching fabric

In this scenario a large amount of memory is placed at the fabric inputs and outputs, to avoid backpressure. The maximum amount of packets destined to a linecard inside the switching fabric at any point in time is $X \times P_{LC}$. In the worst case they are all maximum-size packets and occupy $X \times P_{LC} \times L_{max} = X \times 16 \times 2048 = X \times 32768$ bytes. We choose this value for the fabric output ports and set the size of fabric input ports to one half of it.

Cre-dits	**Thru-put**	**Pkt. block prob.**		**Av. fabric FIFOs occ.** (Bytes)	
		St.	Bp.	Inputs	Outputs
5	0.918	0.082	0	1211 (1.5%)	6552 (4%)
10	0.942	0.058	0	1340 (0.8%)	10726 (3.3%)
15	0.955	0.045	0	1401 (0.6%)	14342 (2.9%)
20	0.963	0.037	0	1450 (0.4%)	17772 (2.7%)
25	0.969	0.031	0	1489 (0.4%)	21294 (2.6%)
30	0.973	0.027	0	1506 (0.3%)	24255 (2.5%)
50	0.983	0.017	0	1561 (0.2%)	36866 (2.3%)

Table 10.6. Stationary results with extended memory under uniform traffic and variable-size packets

Figure 10.5 shows the throughput achieved with the original memory size and with

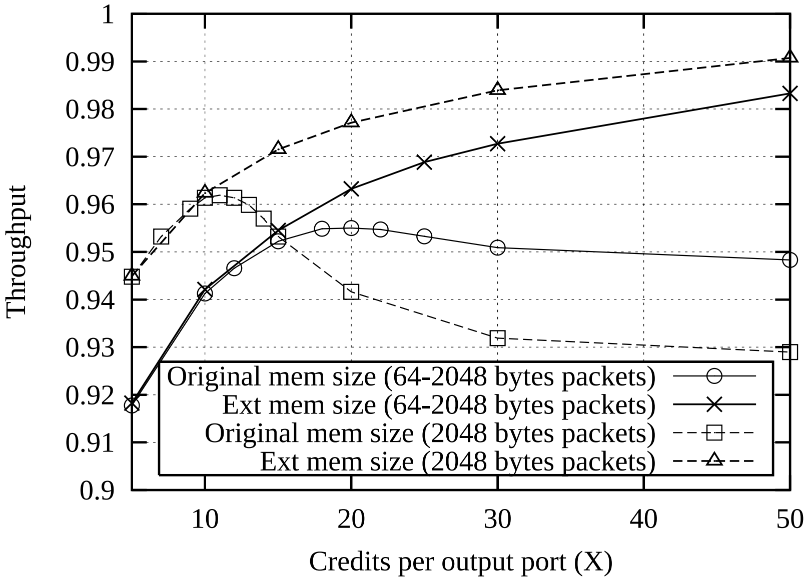


Figure 10.5. Throughput with original and extended memory in the fabric

the extended memory size, in the case of variable- and maximum-size packets. With extended memory, throughput grows fairly rapidly as more credits are available, and finally approaches 100%. Table 10.6 shows that backpressure is never active, but a small (and decreasing) amount of starvation is still experienced by packets even for very large X. This is due to the fact that with so much memory in the switching fabric, the control-loop delay becomes larger and larger as X increases. The more credits are available, the slower they are returned.

The improvement achieved in this scenario is not negligible, but its cost is too high. Fabric memory is perhaps the most scarce resource in the system and it is not reasonable to assume that it is readily available in large quantities. The results presented in this scenario are only meant to be taken as a reference.

10.6.3 Link speed-up between the switching fabric and the linecards

We finally explore the effects of speed-up on the uplink and the downlink (internal fabric speed-up is kept equal to 2). We assume a data rate of 34 Gbps, corresponding to a link speed-up of $34/32 = 1.06$. The possibility to change link speed between the linecards and the switching fabric is another benefit deriving from the asynchronicity of the design.

Figure 10.6 shows the throughput vs. number of credits curves for variable-size and maximum-size packets, with and without speed-up. We observe that both curves with speed-up are monotonically increasing, whereas those without have a maximum and then

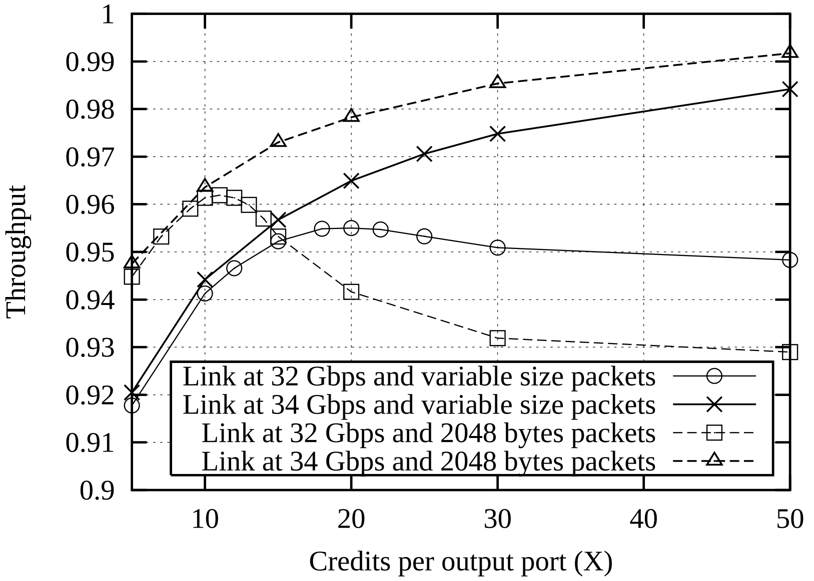


Figure 10.6. Throughput with link speed-up between the switching fabric and the linecards

start decreasing. Moreover, the asymptotic values obtained for large X are much better: both with maximum-size and variable-size packets throughput grows beyond 98%.

Cre-dits	**Thru-put**	**Pkt. block prob.**		**Av. fabric FIFOs occ.** (Bytes)	
		St.	Bp.	Inputs	Outputs
5	0.920	0.080	0	1056 (5.1%)	4513 (11%)
10	0.944	0.056	0	1191 (5.8%)	6268 (15%)
15	0.957	0.043	0	1363 (6.7%)	7286 (18%)
20	0.965	0.034	0.001	1607 (7.8%)	7979 (19%)
25	0.971	0.027	0.002	1810 (8.8%)	8397 (21%)
30	0.975	0.023	0.002	2020 (9.9%)	8868 (22%)
50	0.984	0.014	0.002	2342 (11%)	9638 (24%)

Table 10.7. Stationary results under uniform traffic and variable-size packets, in the case of communication links between the switching fabric and the linecards equal to 34 Gbps

Table 10.7 reports numerical data for the variable-size packets case. Fabric output queues occupancy is low and grows very slowly, as the queues are drained faster. Lower fabric output queues occupancy also translates to lower fabric input queues occupancy,

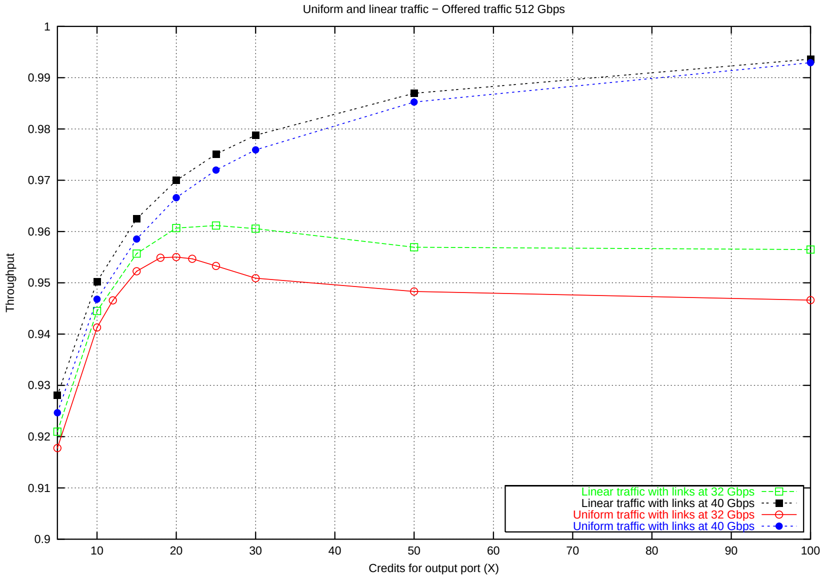


Figure 10.7. Throughput under uniform and linear traffic, with variable packet size

because they are blocked less often and can be drained faster as well. This, in turn, leads to very low backpressure rate towards the In-module.

Starvation remains the only cause of throughput degradation, but its impact is minimal, because higher speed links and shorter queues imply higher credit release rate.

Overall, this solution is very effective and, above all, practical.

10.7 Linear traffic

Linear traffic is an unbalanced pattern in which each port transmits at different rates to all the other ports. It is a variation of *log-diagonal* traffic, described in [73], suitable for a switch with a high number of ports. The traffic matrix is:

$$\tilde{\Lambda} = \frac{2}{N(N+1)} \begin{pmatrix} N & N-1 & \cdots & 2 & 1 \\ 1 & N & \cdots & 3 & 2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ N-2 & N-3 & \cdots & N & N-1 \\ N-1 & N-2 & \cdots & 1 & N \end{pmatrix}$$

For the system under test (with $P_{LC} = 16$ and $N_{LC} = 16$), each linecard transmits to itself roughly 12% of the packets (as opposed to 6% under uniform traffic) and the remaining 88% to other linecards.

Figure 10.7 compares the performance obtained under uniform and linear traffic, for variable link speed-up (32 Gbps or 40 Gbps). Performance under linear traffic is slightly better than under uniform traffic, proving the fact that the fabric does not suffer from traffic unbalance, but actually benefits from reduced contention.

As in the uniform traffic scenario, performance is significantly improved with moderate speed-up on the uplinks and downlinks.

Chapter 11

Performance Under Multicast Traffic

Delivery requirements for multicast traffic are more varied than for unicast and strongly depend on the application. For example, data replication among multiple sites may require absolute delivery guarantee, whereas other applications such as video broadcasting might allow, and in case of congestion even encourage, packets discard. Fibre Channel standards provide two classes of service for multicast [74]. The first is datagram-like and potentially unreliable, because end-nodes do not explicitly acknowledge received packets and do not perform end-to-end flow control. The other, instead, is connection-oriented, requires end-nodes to acknowledge received packets and perform end-to-end flow control.

We study system performance under multicast traffic when two different flow-control policies are employed:

1. The system does not try to regulate incoming traffic in any way. All backpressure mechanisms are disabled and whenever a packet cannot be stored in a buffer, it is simply discarded.

2. The system employs backpressure between buffering stages to prevent overflows, as described in Chapter 9. Backpressure signals are not selective and block all flows traversing the buffer. Packets in principle could be discarded by the fabric if they remain blocked for too long[1] (Section 9.4.2).

11.1 Simulation Model

For the analysis of system performance under multicast traffic we adopt a more abstract model of linecard buffers with respect to the description of Section 9.4.1 . [75]. In particular, we neglect the presence of two stages of queues on the ingress side and consider a single module that jointly represents the PSM and the In-module. This component, which

[1] Actually, with current system settings and under the hypothesis that output ports drain data at line-rate, calculations show that timeout T_o cannot expire, so packets are never discarded by the fabric.

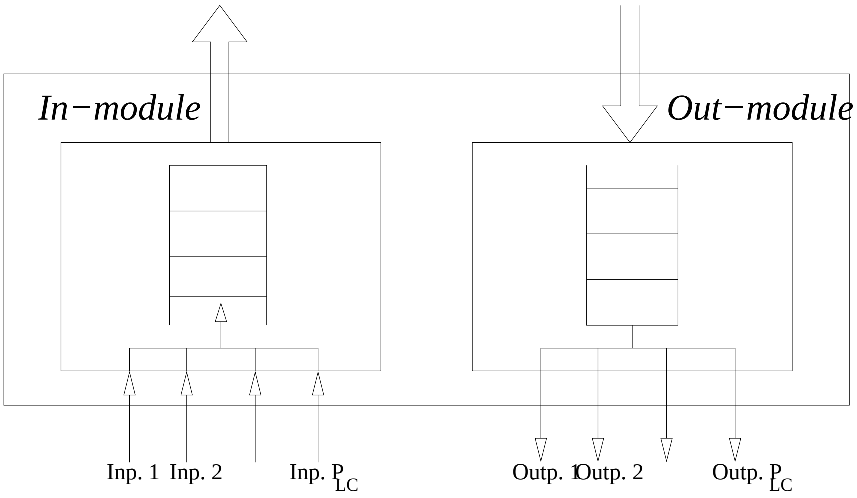


Figure 11.1. Linecard simulation model for multicast

we still call "In-module", consists of a single large buffer organized as a FIFO queue, divided in slots of size equal to a MTU. Total capacity is statically partitioned among input ports; ports can enqueue new packets only if space is available in their memory share. The Out-module is also organized as a single FIFO queue, as previously mentioned. The linecard model implemented in the simulator is show in Figure 11.1.

The Out-module and the switching fabric are modeled according to the description of Chapter 9.

11.2 Simulation settings

The simulated system is a 16×16 switch with 4 linecards hosting 4 input/output ports each. Each port runs at 10 Gbps, hence the aggregate bandwidth is 160 Gbps. Table 11.1 summarizes the values of system parameters we have used.

11.3 Traffic model

In all experiments, three packet size distributions have been considered:

- minimum size (80 bytes) only,
- maximum size (2000 bytes) only,
- uniform between 80 and 2000 bytes, with 40 bytes increment.

Parameter	Symbol	Value
Input - Output ports		
Input-Output ports per linecard	P_{LC}	4
Linecards in the system	N_{LC}	4
Overall number of Input-Output ports	N	16
Link speeds for data and signaling		
Input - Output ports (data path)	—	10 Gbps
Linecard ↔ crossbar (data path)	—	40 Gbps
Packet size		
Minimum packet dimension	—	80 bytes
Maximum packet dimension	MTU	2000 bytes
In-module & Out-module Buffers		
In-module shared buffer size	—	8000 MTUs = 16 MB
Out-module shared buffer size	—	320 KB
Switching Fabric		
Internal speed-up	K	3
Input fabric buffer size	—	10 KB
Output fabric buffer size	—	20 KB
Second timeout value	T_o	15 μs

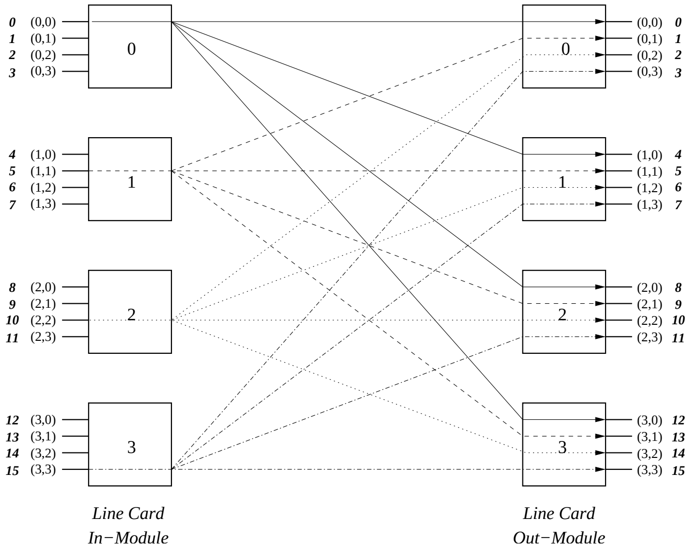
Table 11.1. Summary of the main architecture parameters

Each active source emits a packet with probability ρ_{in}, $0 \leq \rho_{in} \leq 1$ and with probability $1 - \rho_{in}$ remains idle for a period with the same distribution of the packet duration, which can be fixed (minimum or maximum size packets only) or variable (packet size uniformly distributed).

If the backpressure signal from the In-module is active, generation of new packets is blocked. As soon as the backpressure signal is deactivated, the source can start generating again. The effective average input load generated by a source is $\widetilde{\rho_{in}} \leq \rho_{in}$.

Packets generated while a source is experiencing backpressure are simply discarded. We have decided to neglect the fact that in reality these packets would accumulate at the source (Section 8.3.2) because it would introduce a perturbation of the input load and complicate throughput analysis, especially in overload conditions.

The average offered load to an output port is ρ_{out} and it is equal to the sum of the ρ_{in} of input ports transmitting to that output times the probability of selecting that output. If $\rho_{out} > 1$, traffic is not admissible and the output port is overloaded.



Inputs	⟶	Outputs
(0,0)	⟶	[(0,0) - (1,0) - (2,0) - (3,0)]
(1,1)	⟶	[(0,1) - (1,1) - (2,1) - (3,1)]
(2,2)	⟶	[(0,2) - (1,2) - (2,2) - (3,2)]
(3,3)	⟶	[(0,3) - (1,3) - (2,3) - (3,3)]

Figure 11.2. Broadcast traffic scenario with one active port on each linecard

As discussed in Section 9.4, the replication of a multicast packet to multiple ports on the same linecard is instantaneous and does not have any impact on the switching of packets between linecards. Thus, in all the traffic patterns we have selected, the destination ports in the fanout set always reside on different linecards. If the fanout of a packet is F, then it must be replicated to F linecards.

11.4 Broadcast traffic scenario - One active port per linecard

We first present results obtained in two different broadcast scenarios. We call “broadcast” any scenario in which every *linecard* transmits packets to all *linecards* in the system, regardless of how many input ports are active and how many output ports they transmit to.

In Figure 11.2 the multicast pattern under consideration is shown. Each port is identified by the pair (x,y), where x is the linecard number and y the port number on the linecard. On each linecard, only one input port is active and it transmits packets to four output ports on four different linecards. Each output port receives packets from a single

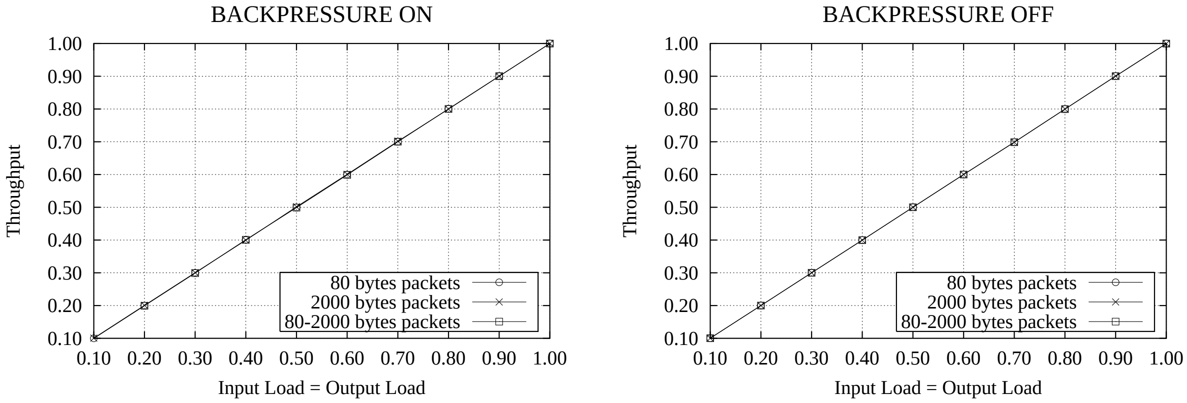


Figure 11.3. Throughput vs. offered load (broadcast scenario, one active port on each linecard)

input port. It follows that traffic is always admissible for any input load $0 \leq \rho_{in} \leq 1$ and that $\rho_{out} = \rho_{in}$.

Figure 11.3 shows average throughput as a function of the total load offered to a single output port, when backpressure mechanisms are enabled (ON) or disabled (OFF).

Despite the high number of conflicts in the crossbar, in both cases the switch sustains the traffic pattern and throughput closely matches the offered load. This is due to the fact that only one port is active on each linecard: the active input port generates as a maximum 10 Gbps of data, which is equivalent to 40 Gbps at the fabric output port due to the fanout 4. Hence the fabric is always loaded at 100% of its capacity.

Table 11.2 reports the numerical values of the average fabric queues occupation at different traffic load. As expected, fabric input and output queue occupancy is quite low. When backpressure is enabled, input queue occupancy is slightly higher, meaning that output queues occasionally reach full occupancy and block input queues. Notice that the table refers to the case of maximum size packets only, when the fabric queues can host a small number of packets.

11.5 Broadcast scenario - Four active ports on each linecard

In this scenario all input ports are active and transmit packets to four output ports on four different linecards. The pattern is described in the following table, using the same numbering scheme of Figure 11.2.

	Backpressure ON			**Backpressure OFF**		
Output Load	**Throughput**	**Xbar Queue**		**Throughput**	**Xbar Queue**	
		In's	**Out's**		**In's**	**Out's**
0.80	0.80	5.8 %	13.2 %	0.80	1.4 %	17.7 %
0.90	0.90	7.0 %	15.7 %	0.90	1.6 %	21.3 %
1.00	1.00	8.2 %	21.6 %	1.00	1.8 %	25.1 %

Table 11.2. Average occupation of fabric input-output memories (broadcast scenario, one active port per linecard, 2000 bytes packets)

Inputs	$\longrightarrow$	Outputs
(0,0) + (1,0) + (2,0) + (3,0)	$\longrightarrow$	[(0,0) - (1,0) - (2,0) - (3,0)]
(0,1) + (1,1) + (2,1) + (3,1)	$\longrightarrow$	[(0,1) - (1,1) - (2,1) - (3,1)]
(0,2) + (1,2) + (2,2) + (3,2)	$\longrightarrow$	[(0,2) - (1,2) - (2,2) - (3,2)]
(0,3) + (1,3) + (2,3) + (3,3)	$\longrightarrow$	[(0,3) - (1,3) - (2,3) - (3,3)]

As each output port receives packets from 4 input ports, $\rho_{out} = 4 \times \rho_{in}$ and traffic is admissible if $\rho_{in} \leq 0.25$. By setting $\rho_{in} > 0.25$ we can generate non-admissible traffic load and observe how the system behaves in overloading conditions.

Notice that from the fabric point of view this scenario is similar to the previous one: all packets coming from a linecard are directed to all linecards and the average load on the fabric input links is 10 Gbps for $\rho_{in} = 0.25$. An important difference regards the burstiness of the traffic arriving on the fabric input links. In this scenario multiple input ports on a linecard can transmit at the same time, effectively generating up to 40 Gbps of traffic towards the fabric, whereas in the previous scenario the load was always strictly limited to 10 Gbps.

Throughput vs. offered output load curves are shown in Figure 11.4. On the top of the graphs the corresponding input load is also indicated.

When backpressure is enabled, throughput closely matches the offered load and saturates to 100% for $\rho_{in} = 0.25$. When backpressure is not used, on the contrary, the system starts experiencing losses when the offered load grows beyond 90%. This is due to the fact that the increased burstiness of the traffic entering the fabric can cause the fabric output queues to temporarily saturate even in underload conditions. Packets that reach a full fabric output queue are simply discarded and throughput decreases. In overload conditions throughput slowly grows to 100%, as packets in excess compensate for those that are discarded. Note that the phenomenon is more evident for maximum-size packets, when the queues can host fewer packets, but it is present also when packet size is variable.

This analysis is confirmed by the numerical results reported in Tables 11.3 and 11.4, referring to the case of maximum size packets. Table 11.3 shows that, when backpressure

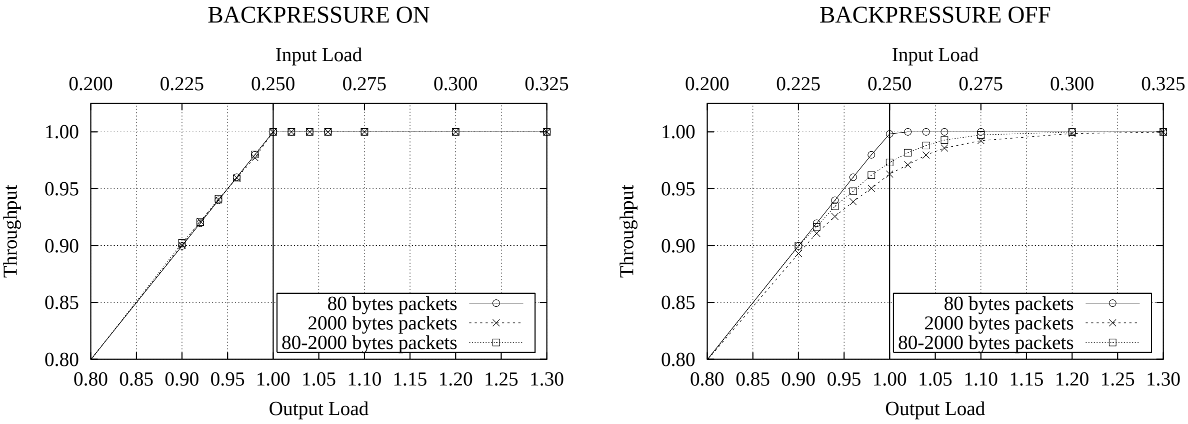


Figure 11.4. Throughput vs. offered load (broadcast scenario, four active ports on each linecard)

Output Load	**Throughput**	**Throughput Loss**			
		In-Mod	**Xbar In**	**Xbar Out**	**Out-Mod**
0.80	0.800 (100 %)	0 %	0 %	0 %	0 %
0.90	0.893 (99.2 %)	0 %	0 %	0.8 %	0 %
0.94	0.923 (98.2 %)	0 %	0 %	1.8 %	0 %
0.98	0.949 (96.8 %)	0 %	0 %	3.2 %	0 %
1.00	0.960 (96.0 %)	0 %	0 %	4.0 %	0 %
1.10	0.992 (90.2 %)	0 %	0 %	9.8 %	0 %
1.20	1.000 (83.3 %)	0 %	0 %	16.7 %	0 %

Table 11.3. Throughput loss (broadcast scenario, four active ports on each linecard, backpressure OFF, 2000 bytes packets)

is off, packets are discarded only at fabric output queues. Table 11.4 reports average fabric queues occupancy. Fabric output queues fill up rapidly when the offered output load becomes larger than 0.90. If backpressure is on, input queues occupancy grows as well, whereas if it is off, this occupancy remains low.

11.6 “Residue” traffic pattern

We now consider traffic patterns that are known to be particularly critical for input-queued switches [30]. These patterns are composed by packets that have a small fanout yet generate a high number of output contentions. It is thus possible to impose high packets

Output Load	Backpressure ON			Backpressure OFF		
	Throughput	Xbar Queue		Throughput	Xbar Queue	
		In's	Out's		In's	Out's
0.90	0.90	5.6 %	35.2 %	0.89	3.9 %	31.1 %
0.94	0.94	9.9 %	48.4 %	0.92	4.2 %	38.2 %
0.98	0.98	29.7 %	71.3 %	0.95	4.5 %	46.5 %
1.00	1.00	93.3 %	92.2 %	0.96	4.7 %	51.7 %
1.10	1.00	94.6 %	91.4 %	0.97	4.8 %	55.3 %
1.20	1.00	94.6 %	92.2 %	1.00	5.0 %	60.2 %

Table 11.4. Average fabric queue occupancy (broadcast scenario, four active ports on each linecard, 2000 bytes packets)

injection rate without violating the admissibility condition (thanks to the small fanout) and, at the same time, stress the switching fabric (due to the high number of contentions). As fabric speed-up may not be sufficient to accommodate all contending packets, some of them receives partial service, i.e. a *residue* is left at the fabric input queue. For this reason, we name this kind of traffic patterns "Residue".

11.6.1 "Residue 2" traffic pattern

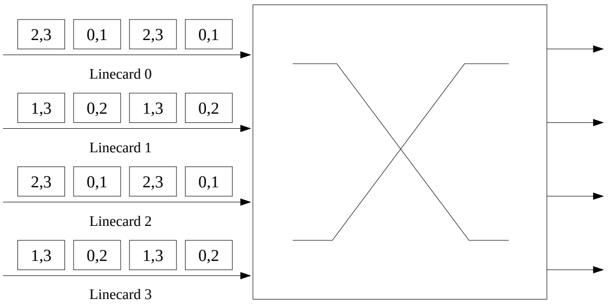


Figure 11.5. "Residue" multicast traffic pattern with fanout 2, from the fabric point of view

The first pattern we consider is summarized in the following table:

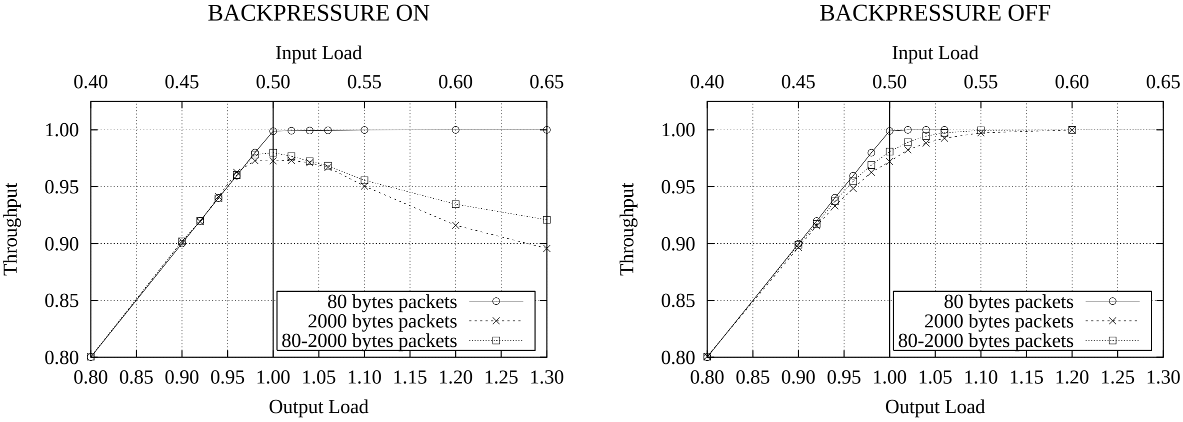


Figure 11.6. Throughout vs. offered load (Residue pattern, fanout 2)

Inputs	⟶	Outputs	Inputs	⟶	Outputs
(0,0)	⟶	[(0, 0) - (1, 0)]	(0,2)	⟶	[(0, 2) - (1, 2)]
(0,1)	⟶	[(2, 1) - (3, 1)]	(0,3)	⟶	[(2, 3) - (3, 3)]
(1,0)	⟶	[(0, 0) - (2, 0)]	(1,2)	⟶	[(0, 2) - (2, 2)]
(1,1)	⟶	[(1, 1) - (3, 1)]	(1,3)	⟶	[(1, 3) - (3, 3)]
(2,0)	⟶	[(0, 0) - (1, 0)]	(2,2)	⟶	[(0, 2) - (1, 2)]
(2,1)	⟶	[(2, 1) - (3, 1)]	(2,3)	⟶	[(2, 3) - (3, 3)]
(3,0)	⟶	[(0, 0) - (2, 0)]	(3,2)	⟶	[(0, 2) - (2, 2)]
(3,1)	⟶	[(1, 1) - (3, 1)]	(3,3)	⟶	[(1, 3) - (3, 3)]

The same pattern, seen from the fabric point of view, is depicted in Figure 11.5. Packets coming from a linecard always contend with at least two packets from other linecards. For instance, packets coming from $LC\,0$ always have one conflict with packets coming from $LC\,1$ and $LC\,3$ and one conflict on average with packets coming from $LC\,2$.

Each linecard has four active input ports and each output port is loaded by two inputs, so traffic is admissible if $\rho_{in} \leq 0.5$.

Figure 11.6 shows throughput vs. offered load curves when backpressure mechanisms are set ON or OFF.

When backpressure mechanisms are OFF, system performance is similar to that obtained in the previous scenario, both in underload and in overload conditions. When backpressure mechanisms are ON, on the contrary, significant differences can be observed. System throughput is close to ideal when the offered load is less than ~ 0.96, but at that point it stops growing and actually starts decreasing. The trend can be better observed in Figure 11.7 where offered load is varied up to its maximum value ($\rho_{in} = 1.0$, $\rho_{out} = 2.0$). Throughput loss is especially evident when maximum size packets are used but, is significant also when packet size is uniformly distributed. With minimum size packets, on the

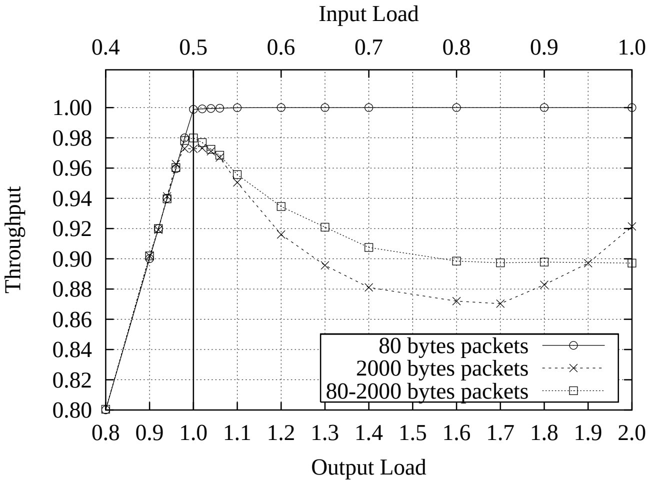


Figure 11.7. Throughout vs. offered load (Residue pattern, fanout 2, backpressure ON)

Input Load	Output Load	Throughput	Average service	Xbar Queue	
				In’s	Out’s
0.40	0.80	0.800	1.471	4.3 %	16.9 %
0.45	0.90	0.899	1.374	7.1 %	29.6 %
0.47	0.94	0.940	1.266	17.4 %	42.8 %
0.49	0.98	0.973	1.041	86.6 %	64.9 %
0.50	1.00	0.973	1.040	86.6 %	64.9 %
0.55	1.10	0.950	1.059	86.9 %	60.8 %
0.60	1.20	0.916	1.067	87.3 %	56.0 %

Table 11.5. Performance results with “Residue” pattern (fanout 2, backpressure ON, 2000 bytes packets)

contrary, no loss is experienced.

Table 11.5 reports fabric queues occupancy when maximum size packets are used. We can see that input queues occupancy grows rapidly as ρ_{out} approaches 0.96 and saturates to $\sim 87\%$. Output queues occupancy, on the contrary, reaches its maximum at 0.96 and steadily decreases afterward.

To understand this behavior, we must focus on what happens at the In-modules. If the average fabric input queues occupancy is high, In-modules are subject to backpressure very often, and they fill up as well. When the In-module memory is almost full, backpressure towards the sources is activated. As the In-module memory is statically partitioned, each source enters and exits backpressure individually; in particular, sources that recently have generated more aggressively enter backpressure earlier. When a source

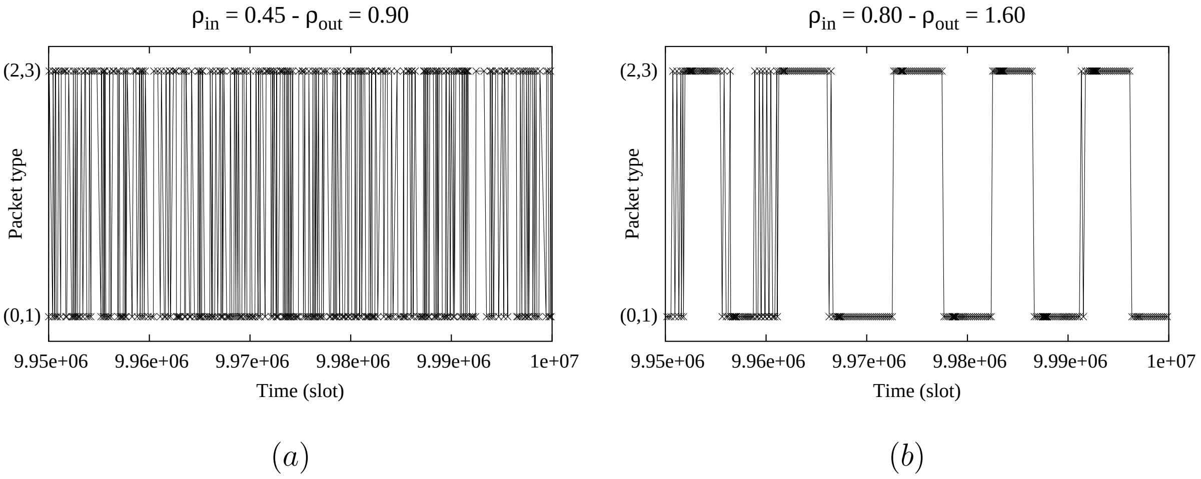


Figure 11.8. Time traces of packets entering the fabric input 0, at low load (a) and high load (b)

experiences backpressure, the process of packets entering the linecard changes. Consider, for instance, $LC\,0$: port (0,0) and (0,2) generate only packets destined to linecards $\{0{,}1\}$, whereas ports (0,1) and (0,3) generate only packets destined to linecards $\{2{,}3\}$. If all ports on the linecard are active, on average half of the enqueued packets are destined to linecards $\{0{,}1\}$ and the other half to linecards $\{2{,}3\}$. Besides, they are roughly alternated, because all sources generate uncorrelated packets. On the contrary, if a $\{0{,}1\}$ source is blocked, more packets destined to linecards $\{2{,}3\}$ are enqueued than packets aimed at linecards $\{0{,}1\}$. It can even happen that both of $\{0{,}1\}$ sources are backpressured at the same time and a long burst of $\{2{,}3\}$ packets enter the In-module. This long burst will reach the fabric input queues as well.

Figure 11.8 shows a trace of packets entering the fabric input 0 queue at low (a) and high (b) load over 50000, hence 0.4 ms. timeslots. The high load graph displays ~ 330 packets, and the bursts are approximately 30 packets long. Burstiness naturally leads to performance penalties. Conflicting bursts in the fabric input queues prevent efficient usage of the crossbar switching capacity. Some linecard may not receive packets for long periods, despite the fact that many packets destined to them are present in the queue. This is a form of head-of-the-line blocking due to the usage of a single queue for multicast traffic.

Notice that this phenomenon is self-sustaining: a source that enters backpressure remains blocked for a long time if large bursts of packets from different sources are present ahead in the In-module queue. The hysteresis mechanism used to activate and deactivate backpressure towards sources further facilitates this phenomenon: a blocked port cannot transmit until a minimum number of packets belonging to it are removed from the In-module queue.

When small packets are used, the In-module queue is drained much faster *in terms of number of packets per unit of time*, so sources remain blocked for shorter periods of time and long bursts do not form.

Finally, as ρ_{in} approaches 1, sources tend to synchronize (because they all generate equal size packets almost back-to-back), so they enter and exit backpressure at the same time and burst length decreases. A corresponding throughput increase is visible in Figure 11.7 for $\rho_{out} \geq 1.7$. If variable size packets are used, sources do not synchronize and no throughput improvement is observed.

11.6.2 Modified "residue" traffic pattern with fanout 2

To evaluate system performance under the Residue traffic pattern but without the induced burstiness, we allow all sources on a linecard to generate with equal probability both kind of packets. For example, all sources transmitting from $LC\,0$ generate with equal probability packets destined to linecards $\{0{,}1\}$ and packets destined to linecards $\{2{,}3\}$. With this "modified" pattern (represented in the table below) we make sure that bursts do not form regardless of how many sources are experiencing backpressure at any time.

Inputs	$\longrightarrow$	Outputs	Inputs	$\longrightarrow$	Outputs
(0,0)	$\longrightarrow$	[(0, 0) - (1, 0)] - [(2, 0) - (3, 0)]	(0,2)	$\longrightarrow$	[(0, 2) - (1, 2)] - [(2, 2) - (3, 2)]
(0,1)	$\longrightarrow$	[(0, 1) - (1, 1)] - [(2, 1) - (3, 1)]	(0,3)	$\longrightarrow$	[(0, 3) - (1, 3)] - [(2, 3) - (3, 3)]
(1,0)	$\longrightarrow$	[(0, 0) - (2, 0)] - [(1, 0) - (3, 0)]	(1,2)	$\longrightarrow$	[(0, 2) - (2, 2)] - [(1, 2) - (3, 2)]
(1,1)	$\longrightarrow$	[(0, 1) - (2, 1)] - [(1, 1) - (3, 1)]	(1,3)	$\longrightarrow$	[(0, 3) - (2, 3)] - [(1, 3) - (3, 3)]
(2,0)	$\longrightarrow$	[(0, 0) - (1, 0)] - [(2, 0) - (3, 0)]	(2,2)	$\longrightarrow$	[(0, 2) - (1, 2)] - [(2, 2) - (3, 2)]
(2,1)	$\longrightarrow$	[(0, 1) - (1, 1)] - [(2, 1) - (3, 1)]	(2,3)	$\longrightarrow$	[(0, 3) - (1, 3)] - [(2, 3) - (3, 3)]
(3,0)	$\longrightarrow$	[(0, 0) - (2, 0)] - [(1, 0) - (3, 0)]	(3,2)	$\longrightarrow$	[(0, 2) - (2, 2)] - [(1, 2) - (3, 2)]
(3,1)	$\longrightarrow$	[(0, 1) - (2, 1)] - [(1, 1) - (3, 1)]	(3,3)	$\longrightarrow$	[(0, 3) - (2, 3)] - [(1, 3) - (3, 3)]

Throughput vs. offered load curves for this scenario are shown in Figure 11.9 and some numerical values are reported in Table 11.6. We clearly see that throughput still reaches its maximum value for $\rho_{out} \simeq 0.96$, but does not decrease afterward. Correspondingly, fabric output queues occupancy grows up to 63% and remains at that level for $\rho_{out} > 0.96$. Switch performance is satisfactory: despite the hardness of the traffic pattern, maximum throughput loss is 5% for 2000 bytes packets and 4% for variable size packets.

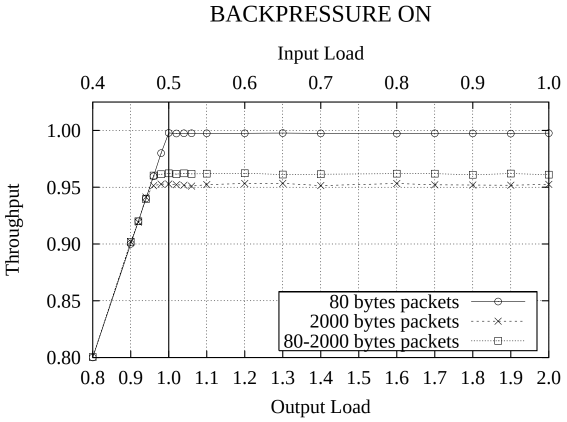


Figure 11.9. Throughput vs. offered load (modified Residue pattern, fanout 2)

Input Load	Output Load	Throughput	Average service	Xbar Queue	
				In's	Out's
0.40	0.80	0.800	1.478	4.4 %	19.6 %
0.45	0.90	0.902	1.318	13.1 %	37.6 %
0.47	0.94	0.941	1.113	47.5 %	55.3 %
0.49	0.98	0.953	1.010	86.1 %	63.2 %
0.50	1.00	0.953	1.003	86.2 %	63.4 %
0.55	1.10	0.952	1.009	86.2 %	63.2 %
0.60	1.20	0.953	1.007	86.1 %	63.5 %

Table 11.6. Stationary results of modified residue pattern (fanout 2) with backpressure ON and 2 Kbytes packets

11.6.3 "Residue 3" traffic pattern

In this section we try a Residue traffic pattern with fanout 3, that further stresses the switching fabric. The pattern is represented from the fabric point of view in Figure 11.10.

Flows have fanout 3 and each packet has at least two conflicts with packets coming from any other linecard. For instance, packets coming from $LC\,0$ always have two conflicts with packets coming from $LC\,1$ and $LC\,3$, as well as two or three conflicts with packets coming from $LC\,2$.

Destination output ports are arranged in such a way that, on average, each output port is loaded by 3 input ports. In particular, each input port generates two kind of packets, aimed at the *same* three linecards. Traffic is admissible for $0 \leq \rho_{in} \leq 0.333$.

Conditions that led to the generation of long bursts of packets with the same fanout

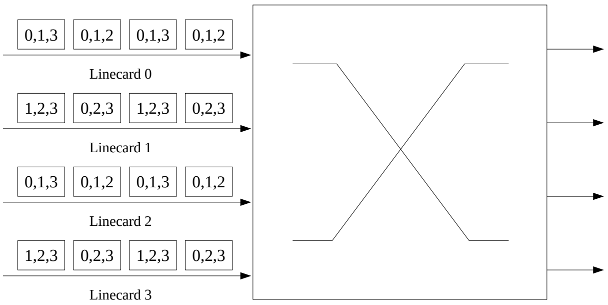
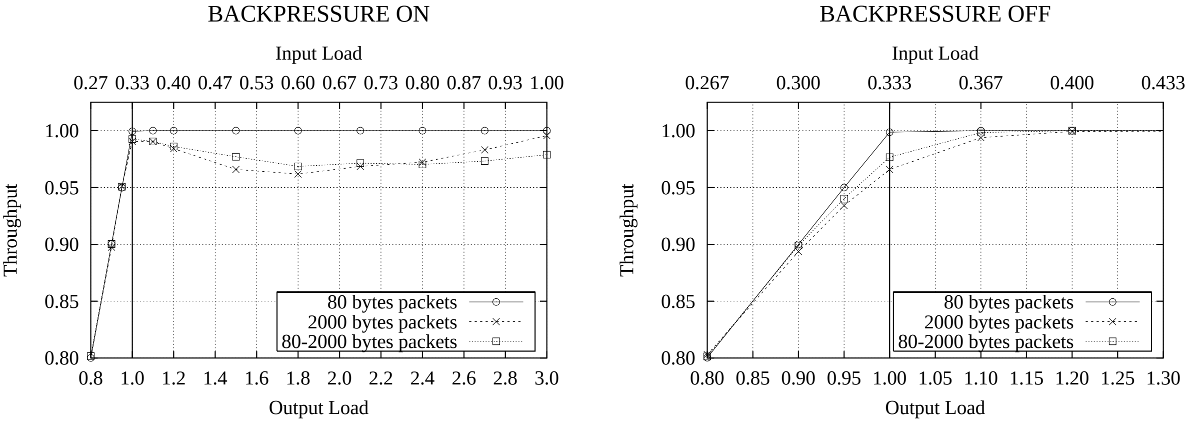


Figure 11.10. “Residue” multicast traffic pattern with linecard packet fanout 3 (four input linecards active)



(a) (b)

Figure 11.11. Throughout vs. offered load (Residue pattern, fanout 3)

in Section 11.6.1 apply to this scenario as well. Therefore, we basically observe in Figure 11.11 the same behavior, although throughput loss in overloading region is much less evident (96% vs. 87% in the worst case).

This is due to the larger fanout of packets in this scenario. When a packet is served by the fabric, it feeds three output queues. Hence, at most one output queue is damaged by bursts at any time. Curves are intermediate between those obtained in the “Residue 2” and in the broadcast scenarios.

Finally, if we modify this traffic pattern to avoid bursts, as we did in Section 11.6.2, we see that throughput remains constant in the overloading region and is higher than 98% for any packet size distribution (Figure 11.12).

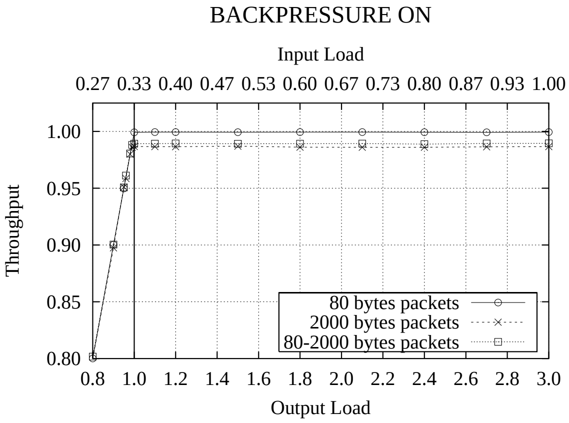


Figure 11.12. Throughput vs. offered load (modified Residue pattern, fanout 3)

11.7 Uniform traffic pattern

In this section we analyze system performance under uniform traffic, i.e. when the fanout set of each packet is chosen randomly and independently over the set of all possible fanouts. Packets can have small or large fanout and the destination output ports do not depend on the source port.

For each packet, we consider each possible destination individually and we include or exclude it depending on the toss of a coin (if the resulting fanout is zero, the procedure is repeated). Given N possible destinations, the average packet fanout is

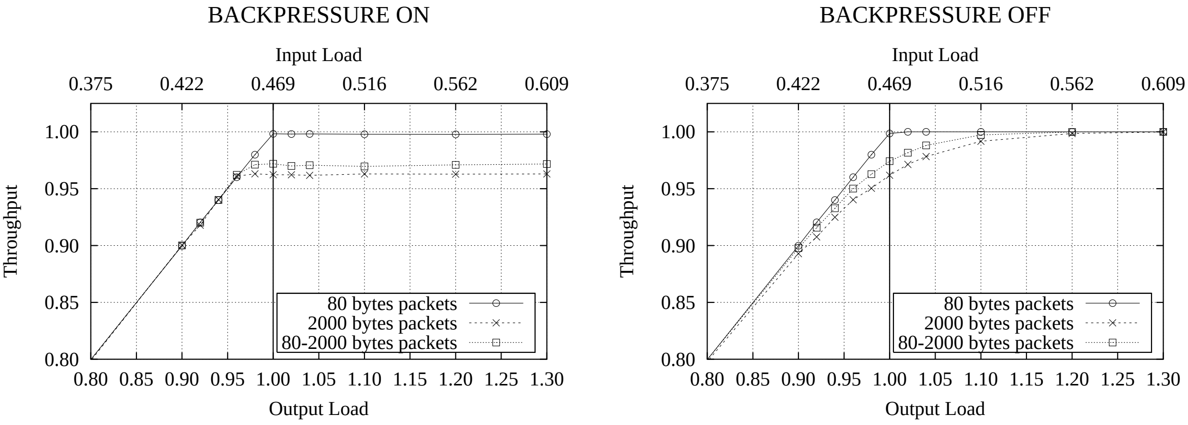
$$\overline{F} = N \, \frac{2^{N-1}}{2^N - 1} \simeq \frac{N}{2}$$

As noted in Section 11.3 the replication of a packet to multiple ports on the same linecard does not affect system performance. Hence, we simplify the pattern by forcing each packet to be addressed to at most one port on each linecard.

As in our configuration $N_{LC} = 4$, every port on a linecard can generate a total of $2^{N_{LC}} - 1 = 15$ packets. Of these, 4 have fanout 1 (unicast), 6 have fanout 2, 4 have fanout 3 and 1 has fanout 4 (broadcast). This corresponds to an average fanout of $32/15$ and traffic is admissible for $0 \leq \rho_{in} \leq 15/32 = 0.48675$.

Figure 11.13 (a) and (b) shows throughput vs. offered load when backpressure mechanisms are enabled and disabled respectively.

When backpressure mechanisms are OFF, system performance is similar to that obtained in previous traffic scenarios. When backpressure is enabled, we see that the system performance is not impacted by the variability of packet fanouts. Its behavior is similar to that observed with the modified version of the “Residue” traffic pattern. Throughput tracks offered load up to $\rho_{out} = 0.96$ and then saturates to 96% for 2 Kbytes packets and



(a) (b)

Figure 11.13. System performance under uniform traffic pattern. (a): backpressure ON. (b): backpressure OFF.

97% for variable size packets. No losses are observed in the overloading region, as packets are uncorrelated and bursts cannot form, regardless of which sources are blocked by backpressure.

Chapter 12

Conclusions – Part II

We have presented a switching architecture specifically designed for Fibre Channel SANs and we have studied its performance by means of simulation under various unicast and multicast traffic patterns.

The system was designed for high-performance and high-scalability. A major role in achieving these goals is played by the asynchronicity of the design that simplifies the implementation of system modules and provides other important benefits.

The demanding requirements of storage traffic, first and foremost loss-free operation, are satisfied with a blend of flow-control and buffer management techniques. In particular, backpressure at every buffering stage avoids packet losses under any circumstances, buffer management policies at the In-modules prevent active ports from monopolizing available space and fine-grained, credit-based, internal flow-control operated by the central arbiter identifies and isolates congesting flows.

Simulation results show that performance is very satisfactory under uniform and non-uniform traffic patterns and for different packet-size distributions. Additional experiments show that it can be further improved by introducing a small speed-up on the uplinks and the downlinks.

The switching architecture can be easily extended to support multicast traffic. The choice of performing replication in two stages, in the switching fabric and in the Out-modules leads to efficient usage of system resources. The switching fabric operates according to an algorithm that tries to gain the benefits of crossbar replication without sacrificing latency.

As the number of potentially active multicast flows grows exponentially with the number of ports, it is not practically feasible to allocate system resources per-flow. In this system multicast packets are simply enqueued in FIFO order, both on the ingress and the egress sides of linecards. This certainly leads to HOL blocking and to unfairness, which become especially dangerous if downstream devices are blocking output ports. Intermediate solutions, entailing a reasonable number of queues and an implementable arbiter are certainly possible, but haven't been investigated yet. We should also keep in mind that strictly lossless behavior for multicast traffic is required only in few particular situations and that in general discarding packets, though undesirable, is acceptable.

Simulation results show that the system achieves satisfactory performance under various multicast traffic patterns, for various packet-size distributions and both in lossy and loss-free operation. We have identified phenomena that can degrade throughput by inducing burstiness on traffic entering the fabric, however, they can only be observed in overloading conditions and under particularly challenging traffic patterns.

Overall we believe that the results of this study prove that this innovative architecture is particularly fit for director-class data-center switches, thanks to its high-performance, robustness and scalability. The phenomena we have identified and the guidelines we have devised can be useful to further evolve this architecture or to develop new ones aimed at the same environment.

Bibliography

[1] H. J. Chao, C. Lam, and E. Oki, *Broadband Packet Switching Technologies*. John Wiley & sons, Sept. 2001.

[2] A. Pattavina, *Switching Theory*. John Wiley & sons, 1998.

[3] J. Y. Hui, *Switching and Traffic Theory for Integrated Broadband Networks*. Boston, MA: Kluwer, 1990.

[4] M. Katevenis, G. Passas, D. Simos, I. Papaefstathiou, and N. Chrysos, "Variable packet size buffered crossbar (cicq) switches," in *Proc. IEEE International Conference on Communications (ICC 2004)*, vol. 2, (Paris, France), pp. 1090–1096, June 20–24, 2004.

[5] K. Yoshigoe and K. Christensen, "A parallel-polled virtual output queued switch with a buffered crossbar," in *Proc. IEEE Workshop on High-Performance Switching and Routing HPSR 2001*, (Dallas, TX), pp. 271–275, May 2001.

[6] A. Bianco, P. Giaccone, E. M. Giraudo, F. Neri, and E. Schiattarella, "Performance analysis of storage area network switches," in *Proc. IEEE Workshop on High-Performance Switching and Routing HPSR 2005*, (Hong Kong), 2005.

[7] W. J. Dally and B. P. Towles, *Principles and practices of interconnection networks*. San Francisco, CA: Morgan Kaufmann, 2003.

[8] N. Karol, M. Hluchyj, and S. Morgan, "Input versus output queueing on a space division switch," *IEEE Trans. Commun.*, vol. 35, pp. 1347–1356, Dec. 1987.

[9] Y. Tamir and G. Frazier, "High performance multi-queue buffers for vlsi communication switches," in *Proc. 15th Ann. Symp. Comp. Archi.*, pp. 343–354, June 1988.

[10] N. McKeown, A. Mekkittikul, V. Anantharam, and J. Walrand, "Achieving 100% throughput in an input-queued switch," *IEEE Trans. Commun.*, vol. 47, pp. 1260–1267, Aug. 1999.

[11] S. T. Chuang, A. Goel, N. McKeown, and B. Prabhakar, "Matching output queueing with a combined input output queued switch," *IEEE J. Sel. Areas Commun.*, vol. 17, pp. 1030–1039, June 1999.

[12] I. Stoica 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), pp. 218–224, May 1998.

[13] J. G. Dai and B. Prabhakar, “The throughput of data switches with and without speed-up,” in *Proc. IEEE INFOCOM 2000*, vol. 2, (Tel Aviv, Israel), pp. 556–564, Mar. 2000.

[14] C. Minkenberg, R. Luijten, F. Abel, W. Denzel, and M. Gusat, “Current issues in packet switch design,” *ACM Computer Commun. Rev.*, vol. 33, pp. 119–124, Jan. 2003.

[15] L. Tassiulas and A. Ephremides, “Stability properties of constrained queueing systems and scheduling policies for maximum throughput in multihop radio networks,” *IEEE/ACM Trans. Automat. Control*, vol. 37, pp. 1936–1948, Dec. 1992.

[16] R. E. Tarjan, *Data Structures and Network Algorithms*. Murray Hills, NJ: Bell Labs, 1983.

[17] T. Anderson, S. Owicki, J. Saxe, and C. Thacker, “High-speed switch scheduling for local area networks,” *ACM Trans. Comput. Syst*, vol. 11, pp. 319–352, Nov. 1993.

[18] N. McKeown, *Scheduling Algorithms for Input-Queued Switches*. PhD thesis, University of California at Berkeley, 1995.

[19] N. McKeown, “The iSLIP scheduling algorithm for input-queued switches,” *IEEE/ACM Trans. Networking*, vol. 7, pp. 188–201, Apr. 1999.

[20] H. Chao and J. Park, “Centralized contention resolution schemes for a large-capacity optical ATM switch,” in *Proc. IEEE ATM Workshop*, (Fairfax, VA), pp. 11–16, May 1998.

[21] D. Serpanos and P. Antoniadis, “Firm: A class of distributed scheduling algorithms for high-speed ATM switches with multiple input queues,” in *Proc. IEEE INFOCOM 2000*, vol. 2, (Tel Aviv, Israel), pp. 548–555, Mar. 2000.

[22] Y. Li, S. Panwar, and H. Chao, “On the performance of a dual round-robin switch,” in *Proc. IEEE INFOCOM 2001*, vol. 3, (Anchorage, AK), pp. 1688–1697, Apr. 2001.

[23] A. Mekkittikul and N. McKeown, “A practical scheduling algorithm to achieve 100% thorughput in input-queued switches,” in *Proc. IEEE INFOCOM ’98*, (San Francisco, CA), pp. 792–799, Apr. 1998.

[24] M. Ajmone Marsan, A. Bianco, and E. Leonardi, “RPA: A simple, efficient, and flexible policy for input buffered ATM switches,” *IEEE Commun. Lett.*, vol. 1, pp. 83–86, May 1997.

[25] A. Smiljanić, R. Fan, and G. Ramamurthy, “RRGS-round-robin greedy scheduling for electronic/optical terabit switches,” in *Proc. IEEE GLOBECOM 1999*, (Rio de Janeiro, Brazil), pp. 584–555, Dec. 1999.

[26] Y. Tamir and H.-C. Chi, “Symmetric crossbar arbiters for VLSI communication switches,” *IEEE Trans. Parallel and Distributed Systems*, vol. 4, pp. 13–27, Jan. 1993.

[27] W. T. Chen, C. F. Huang, C. Y. L., and W. Y. Hwang, “An efficient cell-scheduling algorithm for multicast atm switching systems,” *IEEE/ACM Trans. Networking*, vol. 8, pp. 517–525, Aug. 2000.

[28] P. Gupta and N. McKeown, "Designing and implementing a fast crossbar scheduler," *IEEE Micro*, vol. 19, pp. 20–28, Jan./Feb. 1999.

[29] A. Bianco, P. Giaccone, E. Leonardi, F. Neri, and C. Pilgione, "On the number of input queues required to support multicast traffic in input queued switches," in *Proc. IEEE Workshop on High-Performance Switching and Routing HPSR 2003*, (Torino, Italy), pp. 49–54, June 24–27, 2003.

[30] M. Ajmone Marsan, A. Bianco, P. Giaccone, E. Leonardi, and F. Neri, "Multicast traffic in input-queued switches: Optimal scheduling and maximum throughput," *IEEE/ACM Trans. Networking*, vol. 11, pp. 465–477, June 2003.

[31] J. Hayes, R. Breault, and M. Mehmet-Ali, "Performance analysis of a multicast switch," *IEEE/ACM Trans. Commun.*, vol. 39, pp. 581–587, Apr. 1991.

[32] J. Y. Hui and T. Renner, "Queueing analysis for multicast packet switching," *IEEE Trans. Commun.*, vol. 42, pp. 723–731, feb/mar/apr 1994.

[33] Z. Liu and R. Righter, "Scheduling multicast input-queued switches," *J. Scheduling*, vol. 2, pp. 99–114, 1999.

[34] B. Prabhakar, N. McKeown, and R. Ahuja, "Multicast scheduling for input-queued switches," *IEEE J. Sel. Areas Commun.*, vol. 15, pp. 855–866, June 1997.

[35] "Getting up to speed: The future of supercomputing," tech. rep., National Research Council, 2005.

[36] G. F. Pfister, "An introduction to the InfiniBand architecture," in *High Performance Mass Storage and Parallel I/O: Technologies and Applications* (H. Jin, T. Cortes, and R. Buyya, eds.), ch. 42, pp. 617–632, New York, NY: IEEE Computer Society Press and Wiley, 2001.

[37] G. F. Pfister, *In Search of Clusters*. Upper Saddle River, NJ: Prentice Hall, 2 ed., 1998.

[38] J. Hennessy and D. Patterson, *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann Publishers, Elsevier, third ed., May 2002.

[39] S. P. Vander Wiel and D. Lilja, "When caches aren't enough: data prefetching techniques," *IEEE Computer*, vol. 30, pp. 23–30, July 1997.

[40] J. Duato, S. Yalamanchili, and L. Ni, *Interconnection Networks - An Engineering Approach*. Morgan Kaufmann Publishers, Elsevier, revised ed., 2003.

[41] A. Gara, M. A. Blumrich, D. Chen, G. L.-T. Chiu, P. Coteus, M. E. Giampapa, R. A. Haring, P. Heidelberger, D. Hoenicke, G. V. Kopcsay, T. A. Liebsch, M. Ohmact, S.-B. B. D., T. Takken, and P. Vranas, "Overview of the blue gene/l system architecture," *IBM J. Res. & Dev.*, vol. 49, pp. 195–212, march/may 2005.

[42] N. R. Adiga, M. A. Blumrich, D. Chen, P. Coteus, A. Gara, M. E. Giampapa, P. Heidelberger, S. Singh, S.-B. B. D., T. Takken, M. Tsao, and P. Vranas, "Blue gene/l torus interconnection network," *IBM J. Res. & Dev.*, vol. 49, pp. 265–276, march/may 2005.

[43] C. Clos, "A study of non-blocking switching networks," *Bell System Technical Journal*, vol. 32, pp. 406–424, 1953.

[44] A. Jajszczyk, “Nonblocking, repackable, and rearrangeable clos networks: Fifty years of the theory evolution,” *IEEE Communications Magazine*, vol. 41, pp. 28–33, Oct. 2003.

[45] C. W. Wu and T. Feng, “On a class of multistage interconnection networks,” *IEEE Trans. Computers*, vol. 29, pp. 694–702, Aug. 1980.

[46] C. E. Leiserson, “Fat-trees: Universal networks for hardware-efficient supercomputing,” *IEEE Trans. Computers*, vol. 34, pp. 892–201, Oct. 1985.

[47] A. Schacham, B. A. Small, and K. Bergman, “Interconnection networks for high-performance computing – electronics vs. optics,” internal manuscript, Lighwave Research Laboratory, Columbia University, Mar. 2004.

[48] G. I. Papadimitriou, C. Papazoglou, and S. A. Pomportis, “Optical switching: Switch fabrics, techniques and architectures,” *IEEE J. Lightwave Technol.*, vol. 21, pp. 384–405, Feb. 2003.

[49] S. Kitamura, K. Komatsu, and M. Kitamura, “Very low power consumption semiconductor optical amplifier array,” *IEEE Photonics Technology Letters*, vol. 7, pp. 147–148, Feb. 1995.

[50] F. Masetti, D. Chiaroni, R. Dragnea, R. Robotham, and D. Zriny, “High-speed high-capacity packet-switching fabric: a key system for required flexibility and capacity,” *J. Opt. Netw.*, vol. 2, pp. 255–265, jul 2003.

[51] R. Luijten, C. Minkneberg, R. Hemenway, M. Sauer, and R. Grzybowski, “Viable opto-electronic hpc interconnect fabrics,” in *Proc. ACM Supercomputing 2005*, (Seattle, WA, USA), Nov. 2005.

[52] Q. Yang, K. Bergman, G. D. Hughes, and F. Johnson, “Wdm packet routing for high-capacity data networks,” *IEEE J. Lightwave Technol.*, vol. 19, pp. 1420–1426, Oct. 2001.

[53] R. Hemenway, R. Grzybowski, C. Minkenberg, and R. Luijten, “Optical-packet-switched interconnect for supercomputer applications,” *OSA J. Opt. Netw.*, vol. 3, pp. 900–913, Dec. 2004.

[54] C. Minkenberg, F. Abel, P. Müller, R. Krishnamurty, M. Gusat, P. Dill, I. Iliadis, R. Luijten, B. Roe Hemenway, R. Grzybowski, and E. Schiattarella, “Designing a crossbar scheduler for hpc applications,” *To appear in IEEE Micro*, 2006.

[55] C. Minkenberg, F. Abel, P. Müller, R. Krishnamurthy, and M. Gusat, “Control path implementation of a low-latency optical HPC switch,” in *Proc. of Hot Interconnects 13*, (Stanford, CA), Aug. 17–19 2005.

[56] C. Minkenberg, “Performance of i-SLIP scheduling with large round-trip latency,” in *Proc. IEEE Workshop on High-Performance Switching and Routing HPSR 2003*, (Torino, Italy), pp. 49–54, June 24–27, 2003.

[57] C. Minkenberg, F. Abel, and M. Gusat, “Reliable control protocol for crossbar arbitration,” *IEEE Commun. Lett.*, vol. 9, pp. 178–180, Feb. 2005.

[58] C. Minkenberg, F. Abel, and E. Schiattarella, “Distributed crossbar schedulers,” in

To appear in Proc. IEEE Workshop on High-Performance Switching and Routing (HPSR 2006), (Poland), 2006.

[59] C. Minkenberg, I. Iliadis, and F. Abel, "Low-latency pipelined crossbar arbitration," in *Proc. IEEE GLOBECOM 2004*, (Dallas, TX), Dec. 2004.

[60] E. Oki, R. Rojas-Cessa, and H. Chao, "A pipeline-based approach for maximal-sized matching scheduling in input-buffered switches," *IEEE Commun. Lett.*, vol. 5, pp. 263–265, June 2001.

[61] E. Oki, R. Rojas-Cessa, and H. Chao, "PMM: A pipelined maximal-sized matching scheduling approach for input-buffered switches," in *Proc. IEEE GLOBECOM 2001*, vol. 1, (San Antonio, TX), pp. 35–39, Nov. 2001.

[62] E. Schiattarella and C. Minkenberg, "Fair integrated scheduling of unicast and multicast traffic in an input-queued switch," in *To appead in Proc. IEEE International Conference on Communications (ICC 2006)*, (Istanbul, Turkey), 2006.

[63] "Omnet++ discrete event simulation system." http://www.omnetpp.org/.

[64] K. Pawlikowski, V. Yau, and D. McNickle, "Distributed stochastic discrete-event simulation in parallel time streams," in *Proc. Winter Simulation Conference*, pp. 723–730, 1994.

[65] R. Rojas-Cessa, E. Oki, and H. Chao, "CIXOB-k: combined input-crosspoint-output buffered packet switch," in *Proc. IEEE GLOBECOM 2001*, vol. 4, (San Antonio, TX), pp. 2654–2660, Nov. 2001.

[66] M. Andrews, S. Khanna, and K. Kumaran, "Integrated scheduling of unicast and multicast traffic in an input-queued switch," in *Proc. IEEE INFOCOM 1999*, vol. 3, pp. 1144–1151, Mar. 1999.

[67] M. Song and W. Zhu, "Integrated queueing and scheduling for unicast and multicast traffic in input-queued packet switches," in *Proc. 2nd IASTED International Conference on Communication and Computer Networks*, (M.I.T., Cambridge, MA), Nov. 2004.

[68] A. Smiljanić, "Scheduling of multicast traffic in high-capacity packet switches," *IEEE Communications Magazine*, pp. 72–77, Nov. 2002.

[69] C. Minkenberg, "Integrating unicast and multicast traffic scheduling in a combined input- and output-queued packet-switching system," in *Proc. IEEE ICCCN 2000*, (Las Vegas, NV), pp. 127–234, Oct. 2000.

[70] N. McKeown and B. Prabhakar, "Scheduling multicast cells in an input-queued switch," in *Proc. IEEE INFOCOM 1996*, vol. 1, pp. 271–278, Mar. 1996.

[71] T. Clark, *Designing Storage Area Networks*. Addison Wesley, 2 ed., 2003.

[72] J. Satran, K. Meth, C. Sapuntzakis, M. Chadalapaka, and E. Zeidner, "Internet Small Computer Systems Interface (iSCSI)." RFC 3720 (Proposed Standard), Apr. 2004.

[73] P. Giaccone, B. Prabhakar, and D. Shah, "Randomized scheduling algorithms for high-aggregate bandwidth switches," *IEEE J. Sel. Areas Commun.*, vol. 21, pp. 546–559, May 2003.

[74] ANSI, “Fibre channel - framing and signaling (fc-fs),” INCITS 373-2003, ANSI, Apr. 2003.

[75] A. Bianco, P. Giaccone, E. M. Giraudo, F. Neri, and E. Schiattarella, “Multicast support for storage area network switches,” in *Submitted to the IEEE GLOBECOM 2006*, 2006.

Table of Acronyms

CIOQ:	Combined Input-Output Queued
COTS:	Commercial, Off-the-Shelf
CMOS:	Complementary Metal-Oxide-Semiconductor
CRC:	Cyclic Redundant Check
DSM:	Distributed Shared-Memory
DWD:	Dense Wavelength-Division Multiplexing
EDFA:	Erbium-Doped Fiber Amplifier
E/O:	Electro-Optical
FCFS:	First Come, First Served
FEC:	Forward Error-Correcting Code
FIFO:	First In, First Out
FPGA	Field-Programmable Gate Array
HOL:	Head-Of-the-Line
HPC:	High-Performance Computing
IP:	Internet Protocol
IQ:	Input-Queued
LAN:	Local Area Network
MIN:	Multistage Interconnection Network
MPP:	Massively Parallel Processor
MTU:	Maximum Tranfer Unit
NUMA:	Non-Uniform Memory Access
OBS:	Optical Burst Switching
O/E:	Opto-Electrical
OSMOSIS:	Optical Shared-MemOry Supercomputer Interconnect System
OQ:	Output-Queued
RDQ:	Reliable-Delivery Queue
RTT:	Round-Trip Time
SAN:	Storage Area Network
SMP:	Symmetric Multi-Processor
SOA:	Semiconductor Optical Amplifiers
TCP:	Transport Control Protocol
TTL:	Time To Live
UMA:	Uniform Memory Access
VOQ:	Virtual Output Queue
WAN:	Wide Area Network
WDM:	Wavelength-Division Multiplexing