Report Number: WUCSE-2004-42

2004-08-09

Scheduling Algorithms for CIOQ Switches

Prashanth Pappu and Jonathan S. Turner

Most scalable switches are required to buffer packets at both their inputs and outputs to overcome the slow memory speeds of packet queues. This thesis deals with the design of scheduling algorithms for such Combined Input and Output Queued (CIOQ) switches. For crossbar based CIOQ switches, we demonstrate the underperformance of commercially used scheduling algorithms under overload traffic conditions using targeted stress tests and present ideas to develop robust, stress resistant versions of these algorithms that are still simple enough to be implemented in high speed switches. To regulate the flow of traffic in buffered, multi-stage switches, we introduce... **Read complete abstract on page 2.**

Scheduling Algorithms for CIOQ Switches

Prashanth Pappu and Jonathan S. Turner

Complete Abstract:

Most scalable switches are required to buffer packets at both their inputs and outputs to overcome the slow memory speeds of packet queues. This thesis deals with the design of scheduling algorithms for such Combined Input and Output Queued (CIOQ) switches. For crossbar based CIOQ switches, we demonstrate the underperformance of commercially used scheduling algorithms under overload traffic conditions using targeted stress tests and present ideas to develop robust, stress resistant versions of these algorithms that are still simple enough to be implemented in high speed switches. To regulate the flow of traffic in buffered, multi-stage switches, we introduce a novel mechanism called distributed scheduling. Distributed scheduling is similiar to crossbar scheduling used in switches with small port counts, but is both distributed and coarse-grained to enable high-speed implementations of scheduling algorithms in high capacity, high performance switches. In this thesis, we comprehensively study and evaluate distributed scheduling.

2004-42

Scheduling Algorithms for CIOQ Switches, Doctoral Dissertation, August 2004

Authors: Pappu,Prashanth

August 9, 2004

Abstract: Most scalable switches are required to buffer packets at both their inputs and outputs
to overcome the slow memory speeds of packet queues.
This thesis deals with the design of scheduling algorithms for such Combined Input and Output Queued (CIOQ) switches.

For crossbar based CIOQ switches, we demonstrate the underperformance of commercially used scheduling algorithms
under overload traffic conditions using targeted stress tests and present ideas to develop robust,
stress resistant versions of these algorithms that are still simple enough to be implemented in high speed switches.

To regulate the flow of traffic in buffered, multi-stage switches, we introduce a novel mechanism called distributed scheduling. Distributed scheduling
is similar to crossbar scheduling used in switches with small port counts, but is both distributed
and coarse-grained to enable high-speed implementations of scheduling algorithms in high capacity,
high performance switches. In this thesis, we comprehensively study and evaluate distributed scheduling

SEVER INSTITUTE OF TECHNOLOGY

DOCTOR OF SCIENCE DEGREE

DISSERTATION ACCEPTANCE

(To be the first page of each copy of the dissertation)

DATE: July 27, 2004

STUDENT'S NAME: Prashanth Pappu

This student's dissertation, entitled Scheduling Algorithms for CIOQ Switches has been examined by the undersigned committee of five faculty members and has received full approval for acceptance in partial fulfillment of the requirements for the degree Doctor of Science.

APPROVAL: ______________________ Chairman

____

____

____

____

Short Title: Scheduling Algorithms for Switches Pappu, D.Sc. 2004

WASHINGTON UNIVERSITY

SEVER INSTITUTE OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

SCHEDULING ALGORITHMS FOR CIOQ SWITCHES

by

Prashanth Pappu

Prepared under the direction of Dr. Jon Turner

A dissertation presented to the Sever Institute of
Washington University in partial fulfillment
of the requirements for the degree of

Doctor of Science

August, 2004

Saint Louis, Missouri

WASHINGTON UNIVERSITY
SEVER INSTITUTE OF TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

ABSTRACT

SCHEDULING ALGORITHMS FOR CIOQ SWITCHES

by Prashanth Pappu

ADVISOR: Dr. Jon Turner

August, 2004
Saint Louis, Missouri

Most scalable switches are required to buffer packets at both their inputs and outputs to overcome the slow memory speeds of packet queues. This thesis deals with the design of scheduling algorithms for such Combined Input and Output Queued (CIOQ) switches.

For crossbar based CIOQ switches, we demonstrate the underperformance of commercially used scheduling algorithms under overload traffic conditions using targeted *stress tests* and present ideas to develop robust, *stress resistant* versions of these algorithms that are still simple enough to be implemented in high speed switches.

To regulate the flow of traffic in buffered, multi-stage switches, we introduce a novel mechanism called distributed scheduling. Distributed scheduling is similar to

crossbar scheduling used in switches with small port counts, but is both *distributed* and *coarse-grained* to enable high-speed implementations of scheduling algorithms in high capacity, high performance switches. In this thesis, we comprehensively study and evaluate *distributed scheduling.*

To my parents

Contents

List of Tables

List of Figures

Acknowledgments

To these, I owe a debt past telling.

First, my advisor Dr. Jon Turner for all his guidance and wisdom. Also, all the committee members for their support and feedback.

My colleagues at the Applied Research Laboratory for all their help and company.

And finally, my parents and family for their infinite patience and unconditional love and support.

Prashanth Pappu

Washington University in Saint Louis
August 2004

Chapter 1

Introduction

1.1 The Scheduling Problem

1.1.1 Anatomy of a Router

The main function of a router[1] is to forward packets from its input ports to its output ports. Fig. 1.1 shows the various components of a router. A switching fabric connects the input side Port Procesors (PPs) to the output side PPs. The port processors queue packets and perform all packet processing functions like packet classification, route lookup and packet scheduling. The input and output transmission interfaces (ITI and OTI) terminate the physical links and provide the requisite conversion and encoding functions for transmitting/receiving packets on the target physical layer. The control processor implements the routing and other network management protocols.

1.1.2 Output Queuing (OQ)

Ideally, we would like all packets in the router to be buffered only at the output ports. In such a router (called Output Queued (OQ) router), when two or more packets destined to the same output arrive simultaneously at different input ports, they are immediately transferred to the output queue to avoid any packet loss. This architecture not only simplifies the design of the router but also

1. Maximizes the throughput of the router.

[1]We use the terms *router and switch* and *packet and cell* interchangeably, unless explicitly specified.

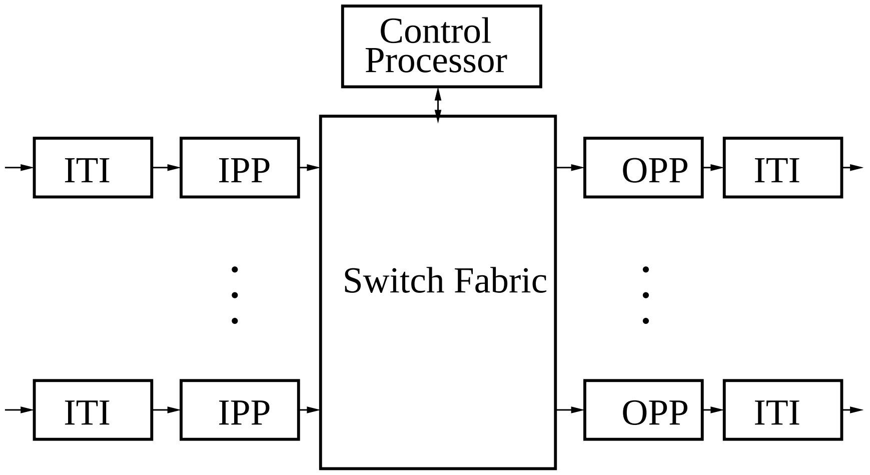


Figure 1.1: Router Architecture

2. Enables the ready use of packet scheduling algorithms [67] for providing Quality of Service (QoS) guarantees to individual flows.

Since there is no queuing at the inputs, the output queues of an OQ router with N line cards each connected to a line operating at a rate R should have a bandwidth of $(N+1) \times R$ (to support N writes for each read). Unfortunately, although router capacities ($N \times R$) have increased by about 2.2 times every 18 months (slightly faster than Moore's law), router buffer (DRAM) speeds have only increased by about 1.1 times every 18 months (slower than Moore's law) [57]. This mismatch makes the use of output queuing infeasible in scalable routers.

1.1.3 Combined Input and Output Queuing (CIOQ)

To overcome the mismatch between the capacities of routers and the speed of memories, most switches queue packets at both input and output ports of the switch. This Combined Input and Output Queuing (CIOQ) lets us use lower speed memories for buffering packets. When two or more packets at different inputs, contend to go to the same output some of them are temporarily held in the input queues before being transferred to the outputs.

The switching fabric of a CIOQ switch is itself operated at a speed S (called *speedup*) times the link rate R. Though, the speedup of a CIOQ switch can be any value between 1 and N, in practice, it is usually a small constant (typically, ≤ 2). Hence, a CIOQ switch with a speedup of S needs memories with a bandwidth of just

$(1+S) \times R$. A switch with a speedup of 1 effectively queues all packets only at inputs and is called an Input Queued (IQ) switch.

While a CIOQ switch requires lower speed switching fabrics and memories, it also introduces a scheduling problem. A decision needs to be made every time slot to determine which inputs are allowed to transfer cells to which outputs. The design of scheduling algorithms to perform this function is the focus of this dissertation. The objective in the design of these schedulers is to approximate the throughput and delay properties of a pure output queued switch.

1.2 CIOQ Switches

1.2.1 Crossbar based CIOQ Switches

Commercial CIOQ switches with small port counts often use a non-blocking crossbar as the switching fabric. An $N \times N$ crossbar is organized as an $N \times N$ matrix to connect input ports to output ports as shown in Fig. 1.2. A crossbar allows multiple cells to pass in parallel to distinct outputs. Though a crossbar has quadratic complexity ($O(N^2)$), it concentrates this complexity within a single chip or a chipset for moderate scale switches, reducing impact on the system cost. The scheduling problem in crossbar based CIOQ switches is reduced to configuring the non-blocking crossbar using a centralized controller every time slot to determine which inputs are allowed to transmit to which outputs.

Though the throughput and delay properties of a scheduler determine its performance, it is the implementation simplicity which is the primary factor in determining which scheduling algorithms are used in high speed switches. For instance, in a switch with external link rates of 10 Gb/s, the scheduler has less than 40 ns to make a scheduling decision.[2]. Hence, though a number of scheduling algorithms have been proposed and studied in the literature (see Chapter 2), very few of them can be implemented and used in practice.

This has created a situation where algorithms with proven performance under a variety of traffic conditions are often not implementable, and implementable algorithms (which do not lend themselves to theoretical analysis) are evaluated only according to their packet delays under admissible and/or random uniform traffic. Hence,

[2]Assuming a minimum packet size of 50 bytes.

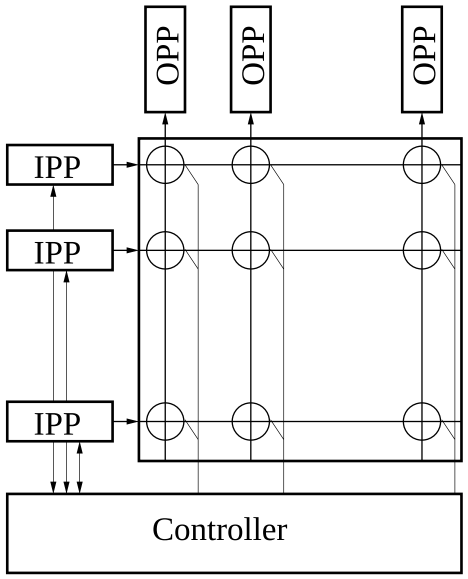


Figure 1.2: Crossbar based switch.

it is unclear how most commercially used algorithms perform under extreme traffic conditions frequently encountered in unregulated IP networks.

1.2.2 Buffered, Multi-stage CIOQ Switches

In crossbar based switches, all the line cards and the crossbars in the switching fabric must be synchronized to the centralized scheduler. The frequency (*once in less than* 40 *ns for a switch with 10 Gb/s links*) and the complexity ($O(\log N)$ *iterations for even simple algorithms*) of the scheduling algorithms, makes it infeasible to use crossbars for large switching systems.

One approach to alleviate the centralized scheduling problem is the introduction of *buffers* at the crosspoints of a crossbar. Such a switching element is called a *buffered crossbar*. With these buffers, cells are sent from inputs into the crossbar only if the corresponding crosspoint has empty buffers. Thus, the ingress scheduling problem is reduced to a simple flow control mechanism. A backpressure signal is used to indicate if the input can forward cells to the crosspoint. On the egress side, the outputs schedule cells from one of the N cross-points destined to them. Unfortunately, the discrete nature of buffering makes this architecture memory intensive. A 16×16 crossbar needs 256 buffers, each with a space for at least a few cells.

The buffered crossbar architecture can be further simplified by using shared memory within the crossbar as shown in Fig. 1.3. The switching element has an

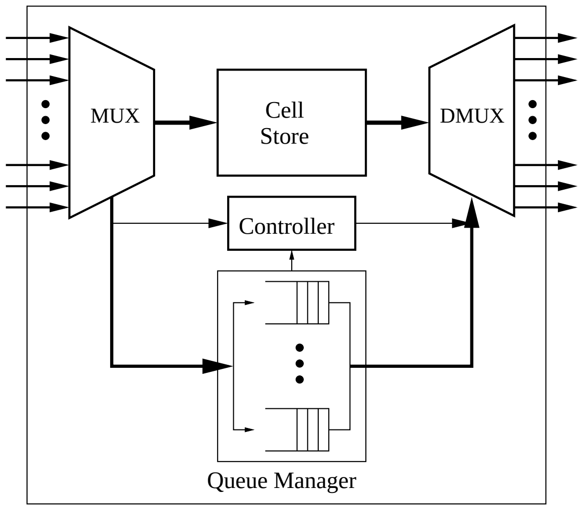


Figure 1.3: Buffered crossbar switch element.

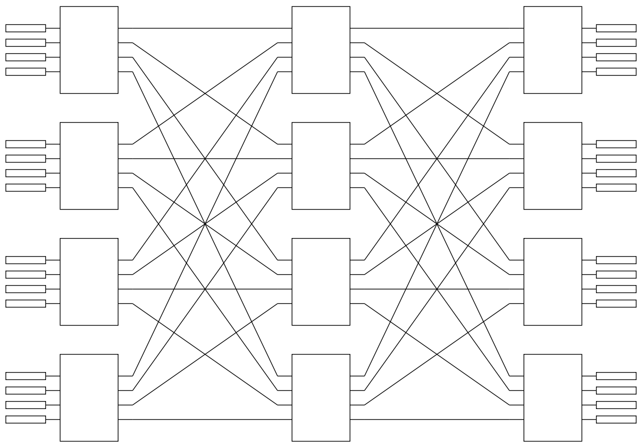


Figure 1.4: Multi-stage switch.

internal shared buffer (*cell store*) for up to a few thousand cells with current technology. The cells are multiplexed from the inputs of the switching element to free slots in the cell store. These cells are then forwarded to the outputs as and when they become idle. A controller uses per output queue information to configure the input side multiplexer and the output side demultiplexer.

Such switching elements can be used in multi-stage switching fabrics with inter stage flow control to build large switching systems as shown in Fig. 1.4. The simple flow control mechanism along with a modest speedup can alleviate the need for a centralized scheduler and maintain throughput even under temporary overloads.

The performance of such switching systems can degrade drastically in the presence of sustained overloads. In extreme traffic conditions, when a single output port *of a switch is under sustained overload, the shared memory buffers in the switching fabric can be congested with cells attempting to reach the overloaded output, interfering with other traffic directed to non-overloaded outputs. The unregulated nature of IP traffic makes such overloads a normal fact of life, which router designers must address if their systems are to be robust enough to perform under the most demanding of traffic conditions.*

1.3 Thesis Overview

1.3.1 Stress Resistant Crossbar Scheduling Algorithms

Crossbar scheduling algorithms are commonly evaluated according to the packet delay under random admissible traffic which tends to obscure significant differences that affect the robustness of different algorithms when exposed to extreme conditions. Commercially used crossbar scheduling algorithms for CIOQ switches can perform poorly under extreme traffic conditions, frequently failing to be work-conserving. On the other hand, there are algorithms with provably good worst-case performance that do not lend themselves readily to high performance implementation. In this thesis, we advocate evaluating crossbar scheduling algorithms using targeted *stress tests* which seek to probe the performance boundaries of competing alternatives when exposed to extreme traffic conditions. Appropriately designed stress tests can reveal key differences among algorithms and can provide the insight needed to spur the development of better solutions.

In this thesis, we introduce the use of stress testing for crossbar scheduling which can be used to evaluate the performance of various kinds of scheduling algorithms (maximal, maximum weighted and work-conserving). We present results that show that maximal size matching and maximum weight matching algorithms need large speedups in order to perform well on stress tests, while work-conserving algorithms like LOOFA can deliver excellent performance, even for speedups less than 1.5. We present ideas to develop improved versions of these algorithms, which take output queue lengths into account, making them much more robust (stress resistant). We also present an algorithm which closely approximates the behavior (and performance) of LOOFA, but which admits a straightforward, high performance hardware implementation.

1.3.2 Distributed Scheduling

In this thesis, we introduce *distributed scheduling* as a means of regulating the flow of traffic through large, high performance multi-stage routers which use buffered crossbar switching elements. The mechanism is both **distributed** and **coarse-grained** to enable high speed implementations of the algorithms. Distributed scheduling, unlike crossbar scheduling, does not seek to schedule the transmission of individual packets. Instead, it regulates the *rates* at which traffic is forwarded through the switching fabric from inputs to outputs. These rates are themselves determined and readjusted at pre-determined time periods using distributed algorithms to let the mechanism scale to switches with large capacities. This also implies that distributed scheduling can only approximate the performance of a pure output queued switch.

In this thesis we present a comprehensive study and evaluation of scheduling algorithms for buffered, multi-stage switching systems including:

1. Work conserving scheduling algorithms.

2. Iterative, work conserving, distributed scheduling algorithms.

3. Single iteration, distributed scheduling algorithms.

4. Performance analysis of distributed scheduling algorithms.

1.3.3 Organization

This thesis is organized as follows. Chapter 2 presents a survey of literature related to the problem of scheduling in CIOQ switches. In this chapter, we identify several areas that require further research and motivate the problems we address in the subsequent chapters. In Chapter 3, we introduce the idea of *stress testing* and present *stress-resistant* scheduling algorithms that are simple to implement and have good performance under a wide variety of traffic conditions. In Chapter 4, we introduce the idea of distributed scheduling for buffered, multi-stage switching systems and present a comprehensive study and evaluation of this mechanism. Conclusions and future work are presented in Chapter 5.

Chapter 2

Related Literature Survey

2.1 System Model and Definitions

In this chapter, we consider CIOQ cell based switches with N input lines and N output lines, all operating at the same cell rate R. Also, let the internal switching rate be $R \times S$, where S denotes the speedup of the switch. Thus, the internal switching fabric operates S times faster than the external input/output links. Although the internal switch speedup can, in general, be obtained in several domains (time, space, wavelength etc), we assume that the CIOQ switch operates in the time domain. In such switches, time is slotted and synchronized and packets from different input queues can be moved simultaneously to different output queues with the conditions that

1. Inputs/outputs can receive/transmit at most one cell in each time slot.
2. Any input can transfer (and any output can receive) a maximum of S cells in each time slot.

It is the job of the scheduling algorithm used in the CIOQ switch to examine the contents of the various packet queues and determine which inputs are allowed to transfer cells to their corresponding outputs.

2.1.1 Arrival Traffic

The arrival traffic to a switching system can be viewed as a set of arrival processes $A_{i,j}(t)$, where $A_{i,j}(t)$ is the discrete-time arrival process of cells at input i to output j. The set of all arrival processes at various inputs is together simply referred to as the *arrival process.* Let $\lambda_{i,j}$ be the average arrival rate of cells at input i destined to output j and let Λ be the matrix of all average arrival rates, i.e, $\Lambda = [\lambda_{i,j}]$.

DEFINITION 1: An arrival process is said to be **admissible** *if no input or output is oversubscribed, i.e,*

$$\forall i \ \lambda_i = \sum_{j=1}^{N} \lambda_{i,j} \leq 1 \tag{2.1}$$

$$\forall j \ \lambda_j = \sum_{i=1}^{N} \lambda_{i,j} \leq 1 \tag{2.2}$$

else, the arrival process is said to be **inadmissible**.

DEFINITION 2: An arrival process is said to be **uniform**, *if all the arrival rates $\lambda_{i,j}$ are equal , otherwise the traffic is* **non-uniform.**

DEFINITION 3: An arrival process is called independent and identically distributed **(i.i.d)** *if all arrivals (both at the same input and across all inputs) are independent of each other and are identically distributed.*

2.1.2 Stability Results

Given an admissible traffic pattern, ideally, we would like to prove that a given scheduling algorithm is **stable** under that traffic.

DEFINITION 4: A scheduling algorithm is said to lead to **weak stability** *if for every $\epsilon > 0 \ \exists \ D > 0$ such that, $\forall i,j \ \lim_{t\to\infty} P\{X_t(i,j) > D\} < \epsilon$, where, $X_t(i,j)$ denotes the number of cells queued at input i for output j at time t.*

DEFINTION 5: A scheduling algorithm is said to lead to **strong stability** *if $\forall i,j \ \lim_{t\to\infty} E(X_t(i,j))$ is finite.*

If an algorithm can be shown to lead to *strong stability*, it is said to achieve 100% throughput and all cells are guaranteed a bounded delay. Performance results which show various scheduling algorithms to be stable under various traffic conditions are simply referred to as *stability results.*

2.1.3 Worst Case Results

Scheduling algorithms cannot be shown to be *stable*[1] (by definition) under inadmissible traffic patterns. But inadmissble traffic patterns that overload ports of a switch are commonly encountered in IP networks.

Scheduling algorithms are evaluated under such **worst case** traffic conditions by comparing their performance with the performance of an ideal output queued switch under the same traffic conditions.

DEFINITION 6: A CIOQ switch is said to **behave identically** *to an output queued switch, if under identical inputs, the departure time of every cell from both switches is identical.*

Consequently, a CIOQ switch that behaves identically to an OQ switch has exactly the same delay and throughput properties under all traffic patterns. This requirement can be relaxed to decouple the definition of the ideal throughput and delay properties.

DEFINITION 7: A CIOQ switch is said to be **work-conserving** *if, in a given operating cycle, every output that has cells queued for it in the system (at various input queues or the output queue) transmits a cell on its outgoing link.*

Performance results which show a CIOQ switch to behave identically to an OQ switch or show that it is work-conserving are referred to as *worst case results.*

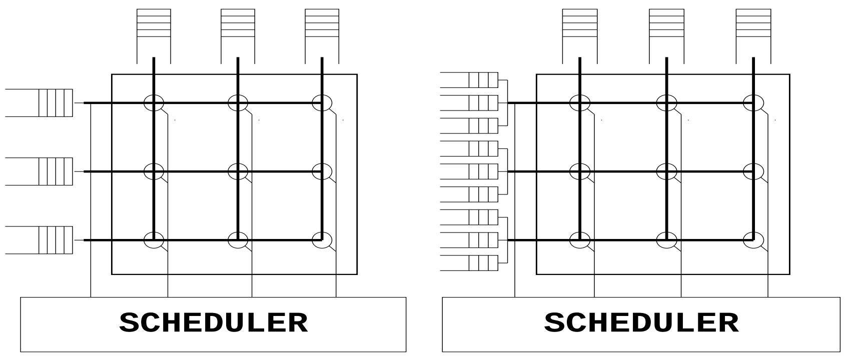
2.1.4 Chapter Organization

This study of the literature related to scheduling algorithms for CIOQ switches is organized as follows. First, we study the queuing policies used in CIOQ switches in Section 2.2. CIOQ switching architectures can be broadly divided into

1. Crossbar based switches.
2. Multi-stage switches.
3. Load balanced switches.

We study the related work for each of these architectures in the following sections. Among these, crossbar based CIOQ switches have been studied in great detail and

[1]When we say a scheduling algorithm is/isn't stable, we mean the switch employing the scheduling algorithm is/isn't stable under the traffic pattern being considered.



(a) FIFO Queuing (b) Virtual Output Queuing

Figure 2.1: FIFO vs Virtual Output Queuing

we study the relevant queuing policies, scheduling algorithms and their related performance and complexity measures in Section 2.3. Section 2.4 presents notes on multi-stage switches (with emphasis on buffered multi-stage switches) and Section 2.5 presents the recently introduced load balanced switches. In Section 2.6, we summarize these studies and results and identify several unanswered questions and areas for further research.

2.2 Queuing Policies

2.2.1 FIFO Queueing

While combined input and output queuing (CIOQ) is a more practical design when compared to output queuing, it can lead to poor performance of the switching system. When the input queues in a CIOQ switch are First In First Out (FIFO) queues (as shown in Fig. 2.1(a)), it is well known that for uniformly distributed, Bernoulli i.i.d arrival traffic, the maximum throughput achieved by any scheduling algorithm is limited to $2-\sqrt{2} = 58\%$ of the link bandwidth [30]. This is because, in FIFO queues, only the first cell in each queue is eligible to be forwarded, leading to the notorious *Head Of Line* (HOL) blocking problem. If a cell at the head of an input queue is blocked, all cells behind it in the queue are also prevented from being transmitted, even when the output link they need is idle.

Also, FIFO queuing can have worse performance under certain traffic patterns. For example, if several input ports each receive a burst of cells destined to the same output, all the cells that arrive later for other outputs will be delayed. It has been shown that such periodic traffic patterns can make the throughput of the switch as small as the throughput of a single link leading to *stationary blocking* [36].

Many techniques have been suggested to overcome HOL blocking. One method is to consider the first $K(K > 1)$ cells in a FIFO queue for transmission during each scheduling cycle [23]. Although these algorithms can improve the throughput of the switch, they are still sensitive to traffic arrivals and perform as badly as FIFO queuing under bursty traffic.

2.2.2 Virtual Output Queuing

HOL blocking can be completely eliminated by the use of separate queues at each input for every output (as shown in Fig. 2.1(b)) [61]. These queues are called Virtual Output Queues (VOQs) and the VOQ at input i for queuing cells destined to output j is represented by $voq_{i,j}$. These VOQs are implemented as linked lists, so the only per queue overhead is the queues' head and tail pointer. We note that despite the increased complexity at the input buffer, the memory bandwidth does not increase: each input still receives and transmits at most one cell per input.

2.3 Crossbar based switches

With the use of VOQs, the scheduling problem for crossbar based switches can be approached in two ways:

1. as a **bipartite graph matching** problem.
2. as a **matrix decomposition** problem.

Each of these approaches leads to distinct solutions as presented in the following sections.

2.3.1 Bipartite Graph Matching

The task of the scheduler is to determine which inputs are allowed to transmit cells to which outputs in a given time slot with the condition that no input or output sends or receives more than one cell. Thus, with the use of VOQs, the task of the scheduler in a crossbar based CIOQ switch can be reduced to finding a matching on a bipartite

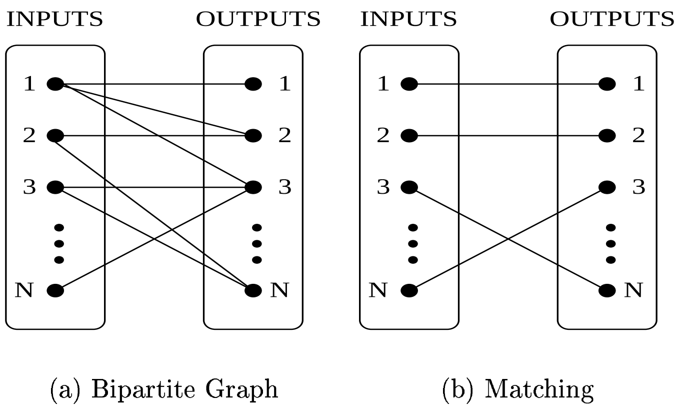


Figure 2.2: Representation of the scheduling problem in a CIOQ switch as a bipartite graph and a corresponding matching.

graph [62] whose vertices are the inputs and the outputs, and an edge between an input and an output denotes that the input has a cell queued for that output [6]. This matching is then used to configure the crossbar.

DEFINITION 8: An undirected **graph** $G = (V, E)$ *connects the set of vertices V with a set of edges E. The graph is said to be bipartite if the set of vertices V can be partitioned into two sets (the corresponding inputs and outputs in the scheduling problem) such that every edge in E has one end in each set. A* **matching M** *on a graph is a subset of the edges E such that no two edges in E have a common vertex.*

Fig. 2.2 shows an example of a bipartite graph and a matching on this graph. Different solutions to the bipartite graph matching problem lead to different scheduling algorithms for the corresponding CIOQ switch.

2.3.2 Maximum Size Matching

A desirable solution to the bipartite graph matching problem would be one that would maximize the number of edges selected in a matching. This approach called maximum size matching provides the highest possible *instantaneous* throughput in a given time slot. A number of stability results have been presented regarding the performance of maximum size matching schedulers:

- In [44], it was demonstrated using simulations that the maximum size matching algorithm is stable for *uniform i.i.d* traffic arrivals for up to a load of 100%.

- But, in [46], the authors prove (using counter examples for $N \geq 3$) that even under Bernoulli admissible traffic, maximum size matching can lead to instability and unfairness, especially, under non-uniform traffic patterns.

  This result was extended to all switch sizes ($N \geq 2$) in [33].

- Also, under inadmissible traffic patterns, the maximum size matching algorithm can lead to *starvation* [46].

- However, the Longest Port First (LPF) algorithm presented in [48] showed that a *carefully chosen* maximum size matching algorithm can be provably stable under Bernoulli admissible traffic for up to a load of 100%.

The maximum size matching can be found by transforming the bipartite graph into a *network flow* (with unit edge capacities) and finding the *maximum flow* in this network [62]. But these techniques are not suitable for implementation in CIOQ switches because it takes too long to find a maximum size matching. The best known algorithm has a complexity of $O(N^{\frac{5}{2}})$, where N is the number of ports of the switch [22].

2.3.3 Maximum Weight Matching

The maximum size matching algorithm makes scheduling decisions that depend on whether a $voq_{i,j}$ is empty or not. Since the scheduler does not make use of the relative backlogs of different queues, it cannot adjust its decisions to favor VOQs with larger backlogs. This can cause it to make poor decisions when exposed to non-uniform traffic. A maximum weight matching (MWM) algorithm can be employed to use greater information about the *state* of the various VOQs in finding a matching.

DEFINITION 9: A weight $w_e = w_{i,j}$ is assigned to each edge $e = (i, j)$ in $\mathbf{E}$ of the bipartite graph $\mathbf{G}$. The MWM algorithm finds a matching $\mathbf{M}$ which maximizes the sum of the weights of all the edges in $\mathbf{M}$. We denote the weight of the matching by $W(M)$, i.e, $W(M) = \sum_{e \in M} w_e$.

Different maximum weight matching algorithms can be obtained by using different weights.

1. **Longest Queue First (LQF)**

   LQF presented in [46] uses the backlog of $voq_{i,j}$ (denoted by $B_{i,j}$) as the weight

$(w_{i,j})$ of the edge $e = (i, j)$ in the bipartite graph.

$$w_{i,j} = B_{i,j}$$

The LQF maximum weight matching algorihtm has been proven to be stable for all admissible i.i.d arrival processes. However, the LQF algorithm can lead to permanent starvation of non-empty VOQs as illustrated in [46].

2. **Oldest Cell First (OCF)**
   OCF, also presented in [46], uses the waiting time of the cell at the head of $voq_{i,j}$ (denoted by $\tau_{i,j}$) as the weight $(w_{i,j})$ of the edge $e = (i, j)$.

   $$w_{i,j} = \tau_{i,j}$$

   The OCF maximum weight matching algorithm has been proven to be stable for all admissible i.i.d arrival processes. Clearly, unlike LQF, there can be no indefinite starvation of any VOQs when using the OCF algorithm.

3. **MWM-$f(B_{i,j})$**
   The authors in [32] conjectured that if $w_{i,j} = B_{i,j}$ can lead to stability then $w_{i,j} = f(B_{i,j})$ will also lead to stability and perhaps, even lesser delays, where $f(.)$ represents a class of nonnegative and continuously differentiable functions.

   The algorithms MWM-$B^2_{i,j}$, MWM-$B^\alpha_{i,j}$ $(\alpha > 1)$ are all stable under admissible i.i.d traffic and a theorem proved in [32] suggests stronger stability with increasing powers of backlog. However, increasing powers of $B_{i,j}$ do not lead to smaller delays implying that perhaps, stronger stability and smaller delays are not necessarily linked.

4. **Longest Port First (LPF)**
   The LPF algorithm presented in [48], by a careful choice of weights, makes the desired matching (the solution) both a maximum weight and a maximum size match. In LPF,

   $$w_{i,j} = \begin{cases} \sum_{k=1}^{N} B_{i,k} + \sum_{k=1}^{N} B_{k,j} & \text{if } B_{i,j} > 0 \\ 0 & \text{otherwise} \end{cases}$$

   LPF is also stable for all admissible i.i.d traffic arrival processes. Also, since LPF finds a match that is both maximum weight and maximum size, it has

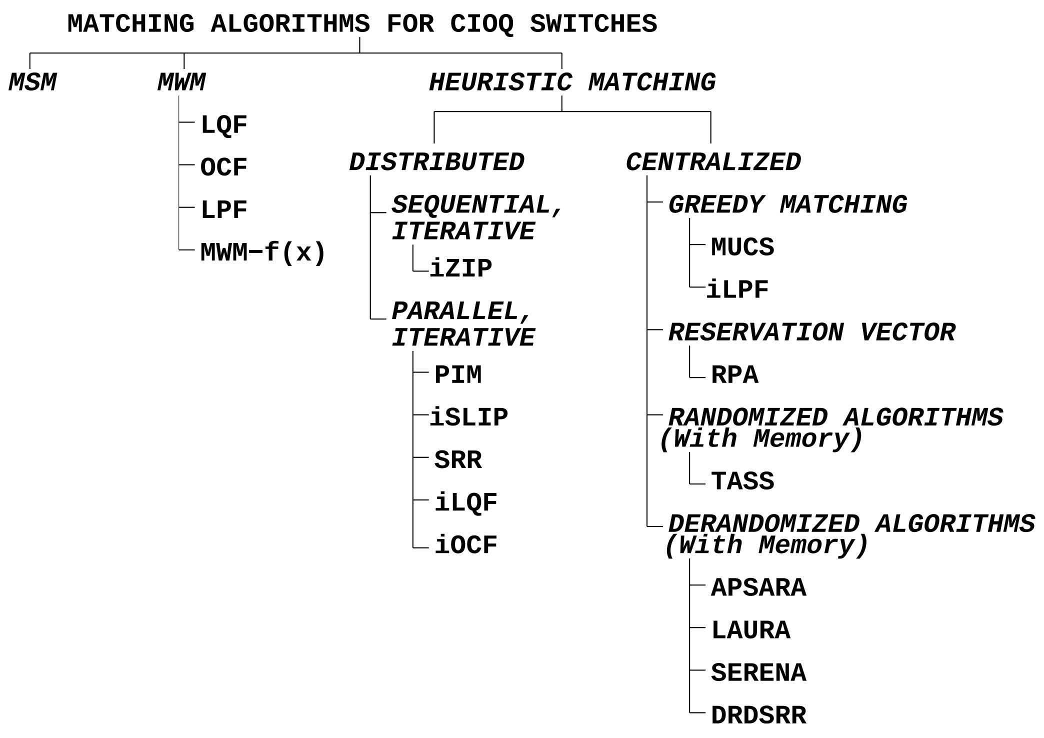


Figure 2.3: Classification of various heuristic matching algorithms used in scheduling CIOQ switches.

smaller cell delays. The run-time complexity of LPF is the same as that of finding a maximum size match $O(N^{2.5})$.

A maximum weight matching can be obtained by solving the equivalent minimum cost network flow problem. The most efficient known algorithm for sequential computation of MWM takes $O(N^{2.5}\log N)$ [62] and $O(N^{2/3}(\log N)^4)$ for a parallel computation of MWM with a polynomial number of processors [7].

2.3.4 Heuristic Matching Algorithms

Both the maximum size and weight matching algorithms are too complex for implementation in high speed routers. Hence, a number of simpler, implementable heuristics have been studied to find a *good* matching on the bipartite graph. Given that a maximum size matching is also a maximum weight matching (with binary weights), it can be argued that these simpler algorithms have been designed to approximate a maximum weight matching.

These approximate maximum weight matching algorithms can be broadly classified as being distributed or centralized [41] as shown in Fig. 2.3.

Distributed algorithms can be implemented such that each input port is required to know only the status of its own queues or cells (*local information*) in finding the desired match. The various input ports use only this local information and follow a distributed protocol (involving exchange of control cells) to converge on a matching. Therefore, these algorithms are good candidates for use in switches which enable incremental scalability, where additional switching capacity and ports are added by introducing more line cards. However, these algorithms can also be implemented in a centralized fashion by just aggregating the status of all ports at a single centralized location where the scheduling decision is made.

Centralized algorithms, on the other hand, require input ports to use information about the status of cells or queues at other input ports also in finding the matching. It is conceivable that this information is small enough to still enable a distributed implementation where individual ports exchange both control and status information (this is possible when the switch has a small number of ports or when the status information exchanged, itself, is minimal). However, for the purposes of classification, we still refer to them as centralized algorithms. Though, as has been noted above, in a few cases, this **does not** preclude their distributed implementation.[2]

In most cases, both distributed and centralized schedulers attempt to at least converge on a *maximal size matching.*

DEFINITION 10: A **maximal** *match on a bipartite graph is one for which at least one endpoint of every edge in the graph is matched.* Thus, no more edges can be trivially added to a maximal matching. A maximum size or weight match is maximal; the converse is not true.

In the following sections, we briefly enumerate the broad techniques used by the two classes of schedulers.

2.3.5 Sequential and Iterative Matching

This simple, distributed technique is executed over multiple *iterations.* In each iteration, a single input (according to a pre-defined *sequence*) gets to pick an *unmatched* output to which it wishes to send a cell. If the input picks an output, both ports are declared *matched* and are unavailable for other unmatched inputs in succesive

[2]The distributed scheduling algorithms for multi-stage, buffered switching systems, introduced in chapter 4 are an instance of such "centralized" algorithms which have been designed explicitly for distributed implementations.

iterations. Clearly, this technique requires N iterations to converge on a maximal matching.

The *i*ZIP algorithm discussed in [41] and the LOOFA algorithm introduced in [34] are examples of algorithms that use sequential and iterative matching.

2.3.6 Parallel and Iterative Matching

This distributed technique, like the sequential and iterative technique, is executed over multiple iterations. However, in each iteration, all the unmatched inputs send *requests in parallel* to all the unmatched outputs for which they have queued cells. The outputs resolve contending *requests* by sending a *grant* to one of the requesting inputs and similarly, the inputs resolve contending output grants by sending an *acknowledge* signal to one of the outputs.

Different parallel and iterative matching algorithms can be obtained with different *contention resolution* techniques.

1. **Using Randomness**
   A simple way to resolve conflicts is to randomly pick one of the contending inputs or outputs. PIM (Parallel Iterative Matching) introduced in [6] is a maximal matching algorithm that tries to approximate the performance of a maximum size matching algorithm. By using randomness, the algorithm also avoids the starvation problem of maximum size matching. PIM can be proven to converge on a maximal size matching in just $O(\log N)$ iterations with high probability. PIM can provide 100% throughput under uniform i.i.d traffic though, with just a single iteration, the throughput of PIM is limited to 63%.

2. **Round Robin Pointers**
   *i*SLIP [42] is a parallel, iterative matching algorithm that uses input and output pointers to resolve contention. In this technique, input (output) ports resolve contention by picking the output (input) which comes first in a round robin order starting from their output (input) pointer. In *i*SLIP, the pointers themselves, are updated after each round to reflect the outcome of the matching. The specific method used by *i*SLIP tends to keep the pointers from being synchronized, but the algorithm does not guarantee any explicit desynchronization properties.

   *i*SLIP is also a maximal size matching algorithm but can provide 100% throughput under admissible, uniform i.i.d traffic with just one iteration. But PIM and iSLIP are designed to handle uniform traffic and have poor performance under non-uniform traffic patterns with *speedup* $= 1$. However, both PIM and iSLIP (and in general, all maximal matching algorithms) have 100% throughput for all admissible, i.i.d traffic with speedup$= 2$ [27].

3. **Desynchronized Round Robin Pointers**
   Unlike the update of pointers in *i*SLIP, it is possible to ensure that the input and output pointers are always desynchronized by starting with a desynchronized setting and updating them in a pre-defined order. This is the technique used in SRR (Static Round Robin) [28].

4. **Using Weights**
   Another way of resolving contention is to use the *weights* of the requests themselves in prioritizing them. Using appropriate weights, we can obtain the parallel and iterative versions of LQF and OCF called iLQF, and iOCF [47].

   The iLQF and iOCF algorithms have been shown (using simulations) to have performance comparable to LQF and OCF under both admissible uniform and non-uniform traffic. These algorithms (like *i*SLIP) require N iterations to converge on a match in the worst case, although $O(\log N)$ iterations have been deemed to be enough in practice.

2.3.7 Greedy Matching

Centralized algorithms, as opposed to distributed algorithms, have access to complete state information from all inputs. Hence, a simple, greedy technique for approximating a maximum weight match is one which iteratively, picks an edge with the maximum weight in the bipartite graph. Once the edge is picked, the corresponding input and output are matched and all edges incident on them are removed from the graph. This process can be continued until no edges are left in the graph, resulting in a greedy, maximal size matching.

The Matrix Unit Cell Scheduler (MUCS) [18] is an example of such an algorithm. The algorithm has been shown to perform well (using simulations), under uniform random and bursty traffic. In spite of the $O(N^3)$ complexity of the MUCS scheduler, it permits a transistor level hardware implementation with an execution time of less than 100 ns using 2 μm CMOS technology.

*i*LPF, an iterative version of the LPF algorithm, is also based on a similar technique. Unlike, *i*LQF and *i*OCF, *i*LPF uses information from all inputs in its weight assignment for each edge of the bipartite graph. Hence, we consider, *i*LPF to be a centralized algorithm. In *i*LPF, pre-sorting of rows and columns is done to make the greedy choice of the maximum weighted edge trivial. Simulation results show that the performance of *i*LPF is comparable to that of LPF under all admissible traffic [47]. Also, *i*LPF has been shown to give smaller cell delays than *i*LQF for non-uniform traffic.

2.3.8 Reservation Vector

Another centralized technique is to use a single *global array* that can be accessed by all inputs to share common information necessary for finding a good matching. This technique is used by the Reservation with Preemption and Acknowledgement (RPA) algorithm, where a single global *reservation vector* is sequentially accessed by all the inputs to reserve access and also preempt other inputs from accessing the same output [40].

The RPA algorithm requires $2N$ iterations and has $O(N^2)$ complexity. The algorithm has been proven to converge on a matching whose weight is more than half the weight of the maximum weight matching on the same bipartite graph.

2.3.9 Randomized Algorithms with Memory

Randomness has proven itself to be a useful technique for developing provably efficient, algorithmic solutions to problems whose deterministic solutions are too complex. Randomized algorithms are particularly simple to implemement because they work on a few randomly chosen samples rather than on the whole state space. A simple randomized solution to the problem of finding an approximate maximum weight matching as introduced in [20] is:

At each time step t, let the schedule $S(t)$ used by the algorithm be the heaviest of d ($d > 1$) matchings chosen at random.

However, [20] proves that the above algorithm is not stable even when $d = O(N)$. Tassiulas [39] presents an improved *randomized algorithm with memory* which works as follows:

At the end of each time step $t-1$, the scheduler keeps in memory the schedule $S(t-1)$. At the following time step t, it computes a new matching $M(t)$ (which is chosen randomly from all possible matchings) and compares the $W(M(t))$ with $W(S(t-1))$ keeping the one which has the higher weight, i.e.,

$$S(t) = \begin{cases} M(t) & \text{if, } W(M(t)) > W(S(t-1)) \\ S(t-1) & \text{otherwise} \end{cases}$$

Thus, by using memory, this algorithm (called TASS) always looks to randomly find a *better* match. The above algorithm has been proved to be stable for all Bernoulli i.i.d traffic and requires just one iteration with $O(\log N)$ steps to compare the matchings.

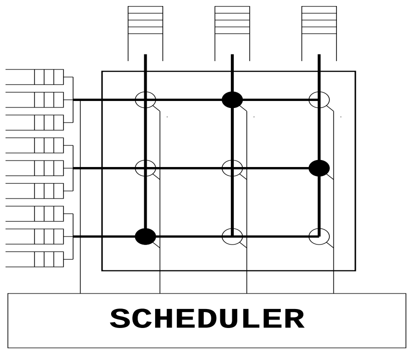
2.3.10 De-Randomized Algorithms with Memory

In high capacity switches (which either have a large number of ports or, high individual link capacities), the state of the switch, carried in its queue lengths, changes slowly with time. This observation is used in [20] to contend that a matching with maximum weight at time t, will be very similar (more precisely, a *neighbour*) to the matching with the maximum weight at time $t-1$.

DEFINITION 11: For a given bipartite graph G, let G' be the complete bipartite graph on the same set of input and output vertices, i.e, every input i has a connecting edge to every output j in G'. Two matchings M and M' on G' are said to be **neighbours***, if they have exactly two inputs i_1 and i_2 which are connected to outputs j_1 and j_2 in M, but are connected to j_2 and j_1 respectively, in M'. And the matchings $M \cap G$ and $M' \cap G$ are said to be neighbours on G.*

Note that, either of the cross-edges, (i_1, j_2) or (i_2, j_1) might not be present in G. Hence, a neighbour $M' \cap G$ can have fewer valid edges than $M \cap G$

The authors in [20] conjecture that instead of randomly picking d ($d > 1$) matchings and choosing the heaviest (as is done in TASS), it is better to randomly choose d neighbours and then choose the heaviest. This is the APSARA algorithm. Using hardware to compute the weights of the d neighbours in parallel, APSARA requires just a single iteration to compute its matching. We note that the matching computed by APSARA need not be maximal. However, APSARA has 100% throughput under admissible, Bernoulli i.i.d traffic [19].



$$\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$$

(a) Crossbar Configuration (b) Permutation Matrix

Figure 2.4: Configuration of a 3x3 crossbar and the corresponding permutation matrix.

The original randomized algorithm (TASS) has poor delay performance. In [20] the authors present two new improved algorithms LAURA and SERENA which use only the *heavy edges* (edges with higher weight) from the previous matching (instead of the whole matching itself) and information of new cell arrivals to find better matching. In LAURA, at time t, the heavier edges of the matching used at time $t-1$ and a randomly chosen neighbour are *merged* to find the matching used at time t. In SERENA, at time t, the new cell arrival information at time t is *merged* with the heavier edges of the matching used at time $t-1$ to find the new matching. Both LAURA and SERENA are stable under all Bernoulli i.i.d traffic, although LAURA has $O(N(\log N)^2)$ complexity and SERENA has $O(N)$ complexity.

Reference [37] presents a Derandomized Rotating Double Static Round Robin (DRDSRR) algorithm that combines the low delay properties of SRR and stability properties of randomized algorithms with memory. The algorithm has been proven to be stable for all admissible, i.i.d traffic and has $O(\log N)$ complexity greater than SRR (which has a pipelined implementation like *i*SLIP).

2.3.11 Matrix Decomposition

All the algorithms presented until now, view the problem of crossbar scheduling as a bipartite graph matching problem. In [16], the authors redefine the problem of crossbar scheduling as a matrix decomposition problem. Firstly, the schedule used by a crossbar scheduler (a matching in the context of a bipartite graph) can be viewed as just a *permutation matrix.*

DEFINITION 11: An $N \times N$ permutation matrix has exactly one of the elements in each row and column set to 1. The rest of the elements of the matrix are set to 0.

Fig. 2.4 shows an example of the schedule used by the crossbar scheduler and the corresponding permuation matrix. Thus, given an arrival traffic rate matrix $\Lambda = [\lambda_{i,j}]$ ($\lambda_{i,j}$, being the average arrival rate of traffic from input i to output j), the problem of crossbar scheduling can be defined as a decomposition of Λ into a series of permutation matrices (P_k), such that,

$$\Lambda \leq \sum_k \phi_k P_k$$

where, $0 < \phi_k \leq 1$ and $\sum_k \phi_k = 1$. With the decomposition, the permuation matrices can then be used to configure the crossbar for the corresponding fraction of time (denoted by ϕ_k), such that, the switching system is stable under the arrrival traffic, Λ.

2.3.12 Birkhoff-von Neumann scheduler

In [16], the authors present a scheduler that uses the Birkhoff-von Neumann algorithm to perform decomposition of any *admissible* arrival traffic rate matrix.

DEFINITION 12: An admissible traffic matrix is also known as a **doubly substochastic** *matrix. Furthermore, if the net arrival rate of traffic to any input or output is exactly equal to* 1*, then, the matrix is said to be* **doubly stochastic**.

The scheduler presented in [16] consists of the following steps:

1. It first uses a result by von Neumann [65], to find a doubly stochastic matrix that is not smaller than the original arrival traffic rate matrix (a doubly substochastic matrix). This algorithm has $O(N^3)$ complexity.

2. It then uses a result by Birkhoff [11], to find a decomposition of the doubly stochastic matrix obtained from the first step. This algorithm has $O(N^{4.5})$ complexity.

3. It finally, uses the Packetized Generalized Processor Sharing (PGPS) algorithm in [53] to determine which permutation matrix (obtained from the decomposition in step 2) is to be used in a given time slot. This algorithm has $O(\log N)$ complexity.

The BvN scheduler based on the Birkhoff-von Neumann decomposition has been shown, using simulations, to have *good* throughput for all admissible traffic patterns. However, the BvN scheduler needs to estimate the arrival traffic rates and perform a complex decomposition and hence, is unsuitable for practical implementations.

2.3.13 Work Conserving Schedulers

All the results regarding the performance of various scheduling algorithms, presented in previous sections, are *stability results*. These results, primarily, evaluate the performance of the algorithms under only admissible traffic patterns. However, inadmissible traffic patterns that overload various output ports of a switch are very common in IP networks. Efforts have been made to design scheduling algorithms which can retain the properties of output queued switches under all (both admissible and inadmissible) traffic patterns. These **worst case results** usually need increased speedup in the switch to maintain their throughput under all traffic conditions.

Reference [13] proposes a scheduling algorithm called Critical Cells First (CCF) which (with speedup of 2) can exactly emulate an output queued switch, i.e, it is both work conserving and preserves the cell ordering of an ideal output queued switch. CCF needs to compute a stable matching which can take as many as N^2 iterations and also the algorithm has high information complexity and needs information (for each cell at an input) that depends on the state of all queues in the system. Though there are techniques to reduce the matching complexity to $O(N)$ and the number of cells that need to be considered at each input can be upper bounded by N, they cannot be applied together and as the authors note in [13] CCF remains a complex algorithm to implement using current technology.

Reference [34] presents a simpler ($O(N)$) algorithm called Lowest Occupancy Output First Algorithm (LOOFA) which keeps the switch work conserving under all traffic conditions. This algorithm can be augmented with timestamps to preserve the cell ordering in a switch with a speedup of 3 [56]. However, these significant algorithms are only of theoretical interest and are not practical for high speed implementations.

2.3.14 Complexity Comparison

Complexity comparison of different scheduling algorithms for crossbar based switches can often be *unfair* because of the varying contexts in which they have their simplest implementations. For instance, while it makes sense to compare the number of iterations to convergence of distributed, iterative scheduling algorithms, the same is not true for centralized algorithms. Even some centralized algorithms, like APSARA, have been designed to take advantage of hardware parallelism and would have higher complexities in a sequential computation. Hence, it would be unfair to compare all the algorithms in a single, centralized context as done in [41]. Detailed implementation complexity comparison of algorithms is often possible only between similar algorithms. For example, [45] has in depth notes on complexity comparison of PIM and *i*SLIP.

However, for the sake of completeness, we highlight the most important, quantitative notes regarding the implementation complexity of various scheduling algorithms in their relevant contexts in Table 2.1.

2.3.15 Performance Comparison

Table 2.3 compares the performance of various scheduling algorithms under three broad classes of traffic

1. Admissible, uniform i.i.d traffic.
2. Admissible, i.i.d traffic.
3. Inadmissible traffic.

From Table 2.1, Table 2.2 and Table 2.3, it is clear that, randomized and derandomized algorithms (like APSARA and DRDSRR) are the best algorithms with respect to both implementation simplicity and performance. However, these algorithms have unpredictable performance under inadmissible traffic conditions.

Also, as can be noted, most of the performance results apply to admissible traffic conditions and there are few algorithms (none of which are simple enough for high speed implementations) that have good performance under inadmissible traffic conditions.

2.4 Buffered Multi-stage Switches

In 1953, Charles Clos published his seminal work, *"A study of non-blocking switching networks"*, in *Bell Systems Technical Journal* [14], in which he showed that it was possible to construct a strictly non-blocking switching network containing fewer crosspoints than a crossbar of the same capacity. The field of multi-stage switching architectures has since developed a rich technical literature and continues to be of practical importance [26, 24]. For example, reference [64] generalizes Clos's results to systems that support connections with varying bandwidth requirements and [58] describes a multi-stage switch built with optical WDM grouped links based on dynamic bandwidth sharing.

The performance of such buffered and unbuffered multistage switching networks has been studied, using analytical models and simulation techniques [55].

In a widely cited paper [66], Jenq proposed an elegant, iterative Markov chain solution for analyzing the performance of binary banyan networks with a single buffer at each switch point. This result has been extended to switching systems constructed from an arbitrary number of line cards and an arbitrary number of buffer slots [60]. Turner et al. [63] developed similar techniques for switching systems with shared buffering and later improved these results in [10].

The general conclusion of these studies is that these systems can provide excellent performance when carrying traffic that does not cause sustained overloads on any output links. In overload traffic conditions, the small buffers in these buffered multi-stage switching fabrics can be congested with cells destined to the overloaded outputs and interfere with the flow of traffic to other non-overloaded outputs causing a significant drop in throughput.

2.5 Load Balanced Switches

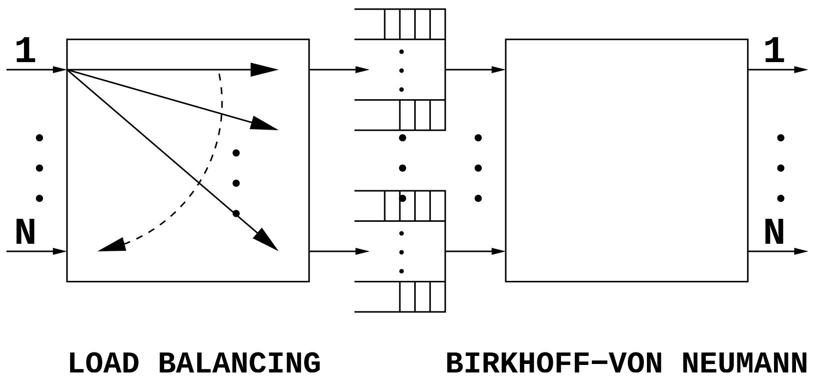


Figure 2.5: The Load Balanced switch.

The Birkhoff-von Neumann scheduler, though impractical, motivated the authors in [15] to propose a much simpler, two-stage switch called the Load Balanced

Birkhoff-von Neumann switch, shown in Fig. 2.5. The intuition behind this switch design is as follows:

1. The BvN scheduler needs to simply cycle through a fixed set of permutation matrices when the incoming traffic is uniform (also, no rate estimation is required).

2. A similar switch (which is also configured using a fixed set of permutation matrices, cyclically) can be used to convert admissible, non-uniform input traffic to uniform traffic! This switch can be used as a *load-balancer* to a second BvN switch which needs to just handle uniform traffic.

Thus, by using a two stage switch, both stages of which are identical and walk through a fixed sequence of configurations (permutation matrices), the load balanced, BvN switch can achieve 100% throughput for all admissible, random traffic. This architecture is the basis for the switch described in [15].

Though, this switch has many attractive properties like simple design, low hardware complexity, no speedup and $O(1)$ complexity of the scheduling algorithm (fixed set of configurations), it also has a significant drawback. This two stage switch can badly resequence packets. To avoid resequencing errors, each output requires a resequencing buffer capable of holding about n^2 packets. These buffers impose a delay that grows as the square of the switch size. For the 600 port switch described in [31], operated with a switching period of 100 ns, this translates to a delay of about 36 milliseconds, a penalty which applies to all packets and not just an occasional packet.

Also, like most of the architectures and scheduling algorithms discussed in this chapter, the load balanced switch has unpredictable performance under inadmissible traffic conditions.

2.6 Summary

In this study of related literature for scheduling traffic in Combined Input and Output Queued (CIOQ) switches, we have identified several unanswered questions.

1. Firstly, as noted in Section 2.3.15, given that most performance studies of implementable scheduling algorithms are primarily stability results, how badly do these algorithms perform under inadmissible traffic conditions?

2. Also, we have seen that, there is **no** algorithm that has a simple implementation and performs well under all traffic conditions.

3. Again, in the case of buffered, multi-stage switching systems, we have identified that these architectures do not have built in mechanisms to maintain their throughput under overload traffic conditions.

In this thesis, we present mechanisms, algorithms and their related performance studies to address these aforementioned issues.

Table 2.1: Comparison of run-time complexity of various scheduling algorithms for crossbar based CIOQ switches.

Algorithm	Complexity
Maximum Size Matching Algorithms	
MSM	Most efficient algorithm currently known converges in $O(N^{5/2})$ time [22] and is equivalent to Dinic's algorithm [17].
Maximum Weight Matching Algorithms	
LQF, OCF, MWM-$f(x)$	Sequential computation takes $O(N^{2.5} \log N)$ [62] while, parallel computation takes $O(N^{2/3}(\log N)^4)$ with a polynomial number of processors [7].
Sequential and Iterative Matching Algorithms	
iZIP	Requires N iterations to find a maximal match [41].
Parallel and Iterative Matching Algorithms	
PIM	Provably converges in $O(\log N)$ iterations on average [6]. Requires $O(N)$ iterations in worst case.
iSLIP	Converges to a maximal match in just *one* iteration on average under uniform random traffic [42]. Requires $O(N)$ iterations in worst case.
SRR, iLQF, iOCF	Like iSLIP, these algorithms require N iterations to converge in the worst case. But, $\log N$ iterations have been deemed sufficient in practice [47].
Greedy Algorithms	
MUCS	Has $O(N^3)$ complexity but allows a fully parallel, transistor level implementation of the core of the algorithm with execution time of < 100 ns using 2μm CMOS technology [18].
iLPF	Has both, an iSLIP like implementation and a $O(N)$ wave front arbiter implementation [47].
Randomized Algorithms with memory	
TASS	$O(\log N)$ complexity for computing and comparing the weight of the new match with that of the previous one [39].

Table 2.2: Comparison of run-time complexity of various scheduling algorithms for crossbar based CIOQ switches (continued from Table 2.1).

Algorithm	Complexity
	De-randomized Algorithms with memory
APSARA	With hardware parallelism, APSARA requires just a *single* iteration. Within each iteration, APSARA takes $O(\log N)$ time for comparing N values [19].
LAURA	Executed for a single iteration with $O(N(\log N)^2)$ complexity [20].
SERENA	Executed for a single iteration with $O(N)$ complexity [20].
DRDSRR	Has $O(\log N)$ complexity greater than SRR which has a pipelined implementation like *i*SLIP [37].
	Birkhoff-von Neumann Scheduler
BvN	The most complex step of the algorithm takes $O(N^{4.5})$ time. If this computation is performed off-line (for slowly changing traffic), it has $O(\log N)$ complexity [16].
	Work Conserving Algorithms
LOOFA	Is similar to sequential, iterative algorithms and has $O(N)$ complexity [34].
CCF	$O(N^2)$ iterations to compute a stable matching. Also has very high *information complexity* [13].

Table 2.3: Summary of performance results of various scheduling algorithms for crossbar based CIOQ switches under various traffic conditions.

Traffic Type	Performance of Algorithms
Admissible, Uniform, i.i.d.	1. Throughput of PIM with one iteration limited to 63%. 2. All other algorithms have 100% throughput. 3. With a speedup of 2, CCF and LOOFA are work-conserving.
Admissible, Non-uniform, i.i.d	1. PIM and iSLIP have poor performance with a speedup of 1. 2. With a speedup of 1, maximum size matching is unstable for all switch sizes. 3. With a speedup of 2, all maximal size matching algorithms (including PIM and *i*SLIP) are stable. 4. MWM, randomized and derandomized algorithms proven to be stable. 5. MUCS, iLQF, iOCF, iLPF, RPA and BvN shown, using simulations, to have good performance. 6. With a speedup of 2, CCF and LOOFA are work-conserving.
Inadmissible	1. With a speedup of 2, CCF and LOOFA are work-conserving. 2. **All the other algorithms have unpredictable performance.**

Chapter 3

Stress Resistant Scheduling Algorithms

3.1 Introduction

3.1.1 Implementation Complexity

In Chapter 2, we reviewed a wide variety of scheduling algorithms for crossbar based CIOQ switches with varying performance and implementation complexities. However, we note that the implementation complexity is the primary factor in determining which algorithms are implemented in most high capacity routers. The capacity of a router is defined as the product of the number of links (N) and the individual link rates (R). Hence, the capacity ($N \times R$) of the router increases with both increasing link rates and increasing number of links supported on each router. Such high capacity routers pose stringent requirements on the execution time of the scheduling algorithms. For example, a switch with links operating at a rate of 10 Gb/s and carrying cells with a minimum size of 50 bytes, has less than 40 ns to make a scheduling decision! Hence, most scheduling algorithms used in practice are heuristic algorithms optimized for implementation simplicity.

3.1.2 Theory vs Simulation

Because of the use of simple, implementable, heuristic scheduling algorithms, most high performance scheduling algorithms and their proven performance results are only of theoretical interest. In particular, the *worst case results* are of little practical use, since the work conserving scheduling algorithms that maintain their throughput in all traffic conditions cannot be used in these high capacity switches because of their high implementation complexity.

Because of the relative difficulty in applying theoretical evaluation techniques to heuristic algorithms in practice, a number of simulation techniques are being widely used.

- These techniques can test the performance of any scheduling algorithm under a variety of traffic conditions.
- Also, with the use of increased speedups in the switch, these techniques can be used to test the performance of the algorithms with varying speedups.
- Any implementation optimization introduced to an algorithm (like execution with a reduced number of iterations) can be tested using these techniques.

3.1.3 Inadmissible Traffic Conditions

Most of the traffic patterns currently used in simulation studies are admissible traffic patterns. These traffic patterns, by definition, do not overload any ports of the switch and tend to obscure significant differences that affect the robustness of different algorithms when exposed to extreme traffic conditions.

This is particularly of concern because of the unregulated nature of IP networks which can cause sustained overloads at output ports of routers. There are a number of factors which can lead to overload problems in IP networks:

- limited route diversity which makes congested links common.
- use of route selection mechanisms which are not guided by session bandwidth needs.
- sudden route changes which can cause rapid traffic shifts.
- use of slow congestion control mechanisms.
- presence of malicious users.

These overload conditions in IP networks are essentially inadmissible traffic patterns that can potentially cause scheduling algorithms to underperform leading to a loss in throughput. Hence, it is not clear how most practical scheduling algorithms used in switches would perform under more realistic, inadmissible traffic conditions in IP networks.

3.1.4 Chapter Outline

To study the performance of scheduling algorithms under these extreme traffic conditions, we have designed a *stress test*. This stress test is a traffic pattern which simulates the unregulated nature of IP networks by overloading the various outputs of a switch with the objective of bringing about the worst case performance of the scheduling algorithms. The test, while not providing any conclusive evidence, helps us in making meaningful distinctions among algorithms operating under extreme conditions. We use this stress test as a tool to gain insight into

- performance of practical scheduling algorithms under extreme conditions (with varying speedups).
- performance of work conserving scheduling algorithms under speedups < 2.
- design of *stress resistant* scheduling algorithms which maintain their throughput both under admissible traffic and extreme traffic conditions and are still simple enough to be used in high speed implementations.

This chapter is organized as follows. In Section 3.2, we first introduce the various traffic patterns and metrics used in simulation studies and present a new metric called *miss fraction* and an inadmissible traffic pattern called the *stress test* to evaluate the performance of scheduling algorithms in overload conditions. We use these various traffic patterns to study the performance of crossbar scheduling algorithms in Section 3.3. In Section 3.5, we use the insights gained from the simulation studies to design *stress resistant* scheduling algorithms for crossbar based CIOQ switches.

3.2 Simulating Overload Traffic Conditions

3.2.1 Throughput Metrics

How do we measure/quantify the throughput of a scheduling algorithm?

1. **Stability**
   In theory, a scheduling algorithm is said to achieve 100% throughput if it is *strongly stable*: i.e, if it can be shown that $E[X_{i,j}(t)] < \infty$, $\forall i, j$ where $X_{i,j}(n)$ is the occupancy at time n of the VOQ at input i that holds cells destined for output j (refer section 2.1.2 for complete definitions).

2. **Queuing Delay**
   In practice, the performance of scheduling algorithms which cannot be theoretically proven to be *stable* is quantified by measuring the *average queuing delay* of packets under various simulated traffic workloads. In some cases, *average input queue length* has also been used as a metric, because, the average queuing delay can be inferred from it by using Little's law. The appeal of this metric actually lies in the fact that it obviates the need for a *packet by packet* simulation which is necessary to measure individual packet delays while directly measuring the average queuing delay.

   Average queuing delay as a metric can be used to study the relative performance of two algorithms for a given workload. But, it is difficult to quantify the absolute performance (throughput) of an algorithm using the measured average queuing delays, especially, when the traffic is non-uniform. Even under uniform traffic, while the maximum throughput achieved can be inferred from the measured average queuing delays, it is non trivial to quantify the exact throughput achieved at various traffic loads. However, scheduling algorithms which provide (measured) bounded, average queuing delays can be concluded to provide 100% throughput under that traffic pattern.

   Unfortunately, neither of these metrics is of any use when the traffic is inadmissible. Inadmissible traffic conditions, by definition, oversubscribe a switch port (an output port in practice) and result in unbounded queue lengths at inputs. This implies that a scheduling algorithm cannot be *stable* under these traffic conditions. Similarly, the measured average queuing delays of the packets are also unbounded in these traffic conditions, with the inherent queuing delays of the packets due to the traffic source dominating over the actual delays induced by the scheduling algorithm.

3. **Work Conserving**
   Hence, in theory, we seek algorithms which can be proven to be **work conserving** under all traffic conditions (including inadmissible traffic). As defined in section 2.1.3, a scheduling algorithm is said to be work conserving if in a given operating cycle, every output for which there are cells queued in the system (at input ports and/or output ports), transmits a cell on its outgoing link, i.e, an output always "works" when there are cells queued in the system. Thus, a work conserving switch is always busy when the corresponding ideal output

queued switch is busy. However, algorithms that can be proved to be work conserving are too complex and many implementable algorithms can be shown (using counter examples) to not be work conserving.

3.2.2 Miss Fraction

In this section, we introduce a new metric called the *miss fraction* to quantify the throughput achieved by a scheduling algorithm used in a CIOQ switch. This metric has been defined to measure the "work-conservingness" of an algorithm under any simulated traffic condition. Hence, this metric can be used to study the performance, under overload traffic conditions, of algorithms which cannot be theoretically proved to be work conserving. In a given measurement period let N_A be the number of cells forwarded by a switch using scheduling algorithm A and N_I be the number of cells forwarded by the ideal output queued switch when subjected to the same workload as algorithm A. Then *miss fraction* is defined as

$$\textit{miss fraction} = 1 - \frac{N_A}{N_I}$$

Thus, the metric essentially determines the relative loss in throughput of a switch using the given scheduling algorithm as compared to the ideal output queued switch under the same traffic conditions. The miss fraction proves to be a particularly useful metric in inadmissible traffic conditions where the average queuing delays are usually unbounded and hence immeasurable.

3.2.3 Admissible Traffic Patterns

The following traffic patterns have been used in simulation studies of scheduling algorithms.

1. **Uniform traffic**
   In the uniform traffic pattern, for a given load $\rho \leq 1$, the average rate of traffic arrival at any input i to any output j is equal to $\frac{\rho}{N^2}$. Thus,

$$\Lambda = \frac{\rho}{N^2}\begin{pmatrix} 1 & 1 & \dots & 1 \\ 1 & 1 & \dots & 1 \\ & \vdots & & \\ 1 & 1 & \dots & 1 \end{pmatrix}$$

where, Λ is the traffic matrix of average arrival rates.

2. **Hotspot traffic**
   In the hotspot traffic pattern, output 1 (the hotspot) receives twice as much traffic as other outputs, from each input. Thus,

$$\Lambda = \frac{\rho}{N^2+N} \begin{pmatrix} 2 & 1 & \dots & 1 \\ 2 & 1 & \dots & 1 \\ & \vdots & & \\ 2 & 1 & \dots & 1 \end{pmatrix}$$

3. **Diagonal trafic**
   In the diagonal traffic pattern, each input i directs a fraction (f) of its incoming traffic to output i and the remaining to output $(i+1)$ **mod** N. This traffic pattern is clearly a lot more skewed than the uniform or the hotspot traffic patterns. The diagonal traffic pattern has been widely used to study the performance of algorithms under non-uniform traffic patterns.

$$\Lambda = \rho \begin{pmatrix} f & 1-f & \dots & 0 \\ 0 & f & \dots & 0 \\ & \vdots & & \\ 1-f & 0 & \dots & f \end{pmatrix}$$

4. **Log-diagonal traffic**
   The log-diagonal traffic pattern is less skewed than a diagonal traffic pattern but still spreads the traffic from various inputs in a non-uniform fashion. In this pattern, each input directs exponentially increasing fractions of its traffic to various outputs. Thus,

$$\Lambda = \frac{\rho}{2^N-1} \begin{pmatrix} 2^{N-1} & 2^{N-2} & \dots & 1 \\ 1 & 2^{N-1} & \dots & 2 \\ & \vdots & & \\ 2^{N-2} & 2^{N-3} & \dots & 2^{N-1} \end{pmatrix}$$

Of these, hotspot, diagonal and log-diagonal are non-uniform but admissible traffic patterns. In general, algorithms tend to have *similar* performance under uniform and

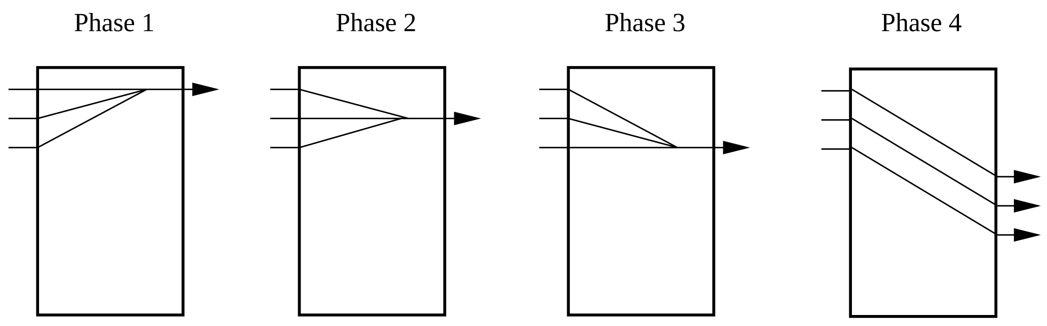


Figure 3.1: Example of stress test with 3 participating inputs and 4 phases

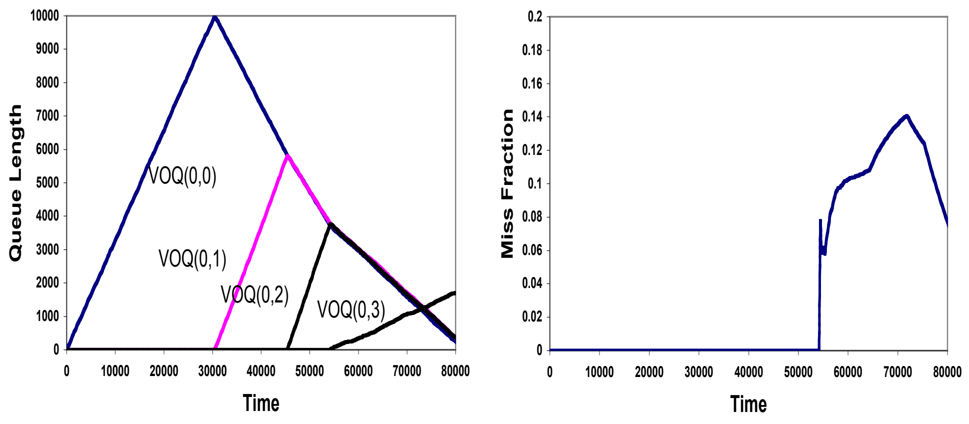
hotspot traffic patterns and diagonal and log-diagonal traffic patterns. But clearly, these traffic patterns do not test the performance of the algorithms in overload conditions.

3.2.4 Stress Test

To test the performance of scheduling algorithms for CIOQ switches under extreme and inadmissible traffic conditions, we have designed a *stress test*. The stress test simulates unregulated traffic by causing sustained overloads at various outputs of the router. Also, while stressing individual outputs, the test attempts to bring about the worst case performance in the work-conserving nature of the scheduling algorithm. To achieve this, the test takes an adversarial approach to stressing various outputs with the goal of increasing the miss fraction of the scheduling algorithm. The adversarial approach of the stress test tries to create conditions where,

1. A single output which has an empty queue has cells queued for it at various inputs.
2. Inputs which have cells queued for an output with an empty queue, also have cells queued for other outputs.

A traffic pattern which can create such conditions can potentially cause a scheduling algorithm to incur larger miss fractions. In particular, the stress test we have designed consists of a series of phases, as illustrated in Fig. 3.1. In the first phase, the arriving traffic at each of several inputs is sent to a single output. This causes each of the inputs to build a backlog for the target output. The arriving traffic at all inputs is then switched to a second output, causing the accumulation of a backlog for the second output at each of the inputs. Successive phases proceed similarly, creating



(a) Queue Lengths (b) Miss Fraction

Figure 3.2: Queue lengths of various VOQs and miss fraction for PIM under a stress test with 3 participating inputs and 4 phases. (N=16, speedup=1.5)

backlogs at each input for each of several outputs. During the final phase, the arriving traffic at each of the inputs is switched to distinct new outputs. Since, these inputs are the only source of traffic for the new target outputs, they must send packets to them as quickly as they come in, while simultaneously clearing the backlog for other outputs, in time to prevent underflow at those outputs. This creates an extreme condition that can cause underflow and increase the miss fraction. The timing of the transitions is chosen to ensure that all the VOQs at each of the participating inputs still have some backlog at the final transition. More specifically, to create the worst case traffic conditions for a given algorithm, the traffic is switched to a new target output when the input backlog for the current target rises to the same level as the input backlog for the previous target. However, when comparing the performance of different schedulers, the transition times and measurement periods are fixed and the same test is applied to all algorithms. The stress test can be varied by changing the number of participating inputs and the number of phases.

Fig. 3.2 better illustrates the progress of a stress test. Fig. 3.2(a) plots the queue lengths of various VOQs of the first input (0), of a switch under a stress test with 3 participating inputs and 4 phases. (This test is illustrated in Fig. 3.1.) The algorithm used in the example is PIM and the speedup of the switch is 1.5. The plot shows how the input directs its traffic to a new output when the input backlog to the current output equals that of the backlog to a previous output. In the last phase input 0 is the only input sending traffic to output 3 but it still accumulates a backlog to that output indicating misses incurred by output 3. Fig. 3.2(b) shows the average miss fraction incurred by the algorithm in this test. We use the interval from the beginning of the last phase to the end of the simulation as a measurement period for the average miss fraction due to the stress test. This explains the spike in the miss fraction curve in Fig. 3.2(b). Algorithms which have smaller miss fractions under these stress tests evidently maintain their throughput even in overload situations.

We note that the stress test only exemplifies a general approach to evaluating CIOQ algorithms under extreme conditions. There may well be other stress test scenarios that are more *stressful* at least for some algorithms and that algorithms that are designed to perform well on the stress test might perform poorly under other tests. However, the intuitive basis of the stress test provides good evidence for distinguishing among algorithms which perform well in overload situations and those that do not.

We study the performance of various centralized crossbar scheduling algorithms under the stress test and present simple improvements which retain the desirable properties of these algorithms and make them *stress resistant.*

3.3 Crossbar Schedulers

For the purposes of our performance evaluation of algorithms, we study representative algorithms from three broad classes of crossbar schedulers.

1. PIM and *i*SLIP which are maximal size matching algorithms that attempt to approximate a *maximum size matching.*

2. APSARA which is a heuristic algorithm that attempts to converge on a *maximum weight matching.*

3. LOOFA which is a *work-conserving algorithm.*

3.3.1 Parallel Iterative Matching (PIM)

PIM is an iterative matching algorithm which attempts to converge on a maximal match in multiple iterations. Each iteration consists of three steps where

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

2. If an unmatched output receives any requests, it chooses one randomly to grant.

3. If an input receives any grants, it choose one to accept and notifies that output.

In [6], the authors show that the algorithm converges to a maximal match in $O(\log N)$ iterations by showing that each iteration eliminates an average of 3/4 of the remaining requests. It is interesting to note that this property is independent of the way the inputs select a grant in the third step of the algorithm.

Also, since the outputs send their grants randomly, when all the input queues are occupied, the probability that no output will grant to a particular input in one iteration is $((N-1)/N)^N)$. Hence, in a single iteration, the throughput of PIM is limited to $(1-\frac{1}{e})$ for large N, which is approximately 63% for $N = 16$.

3.3.2 Iterative Round Robin Matching with Slip (*i*SLIP)

*i*SLIP, like PIM, is an iterative algorithm but is designed to give higher throughputs even for a single iteration. The algorithm iterates the following three steps

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

2. If an unmatched output receives any requests, it chooses one that appears next in a fixed, round-robin schedule starting from its input pointer. The output notifies each input whether or not its request was granted. The input pointer of the round-robin schedule is incremented (modulo N) to one location beyond the granted input if and only if the grant is accepted in step 3 of the first iteration. The pointer is not incremented in subsequent iterations.

3. If an input receives a grant, it accepts the one that appears next in a fixed, round-robin schedule starting from its own output pointer. The output pointer of the round-robin schedule is incremented (modulo N) to one location beyond the accepted output.

The *i*SLIP algorithm maintains good performance even with a single iteration under heavy loads due to its *desynchronization* effect. Step 2 of the algorithm causes different outputs to send grants to different inputs, particularly, under heavy loads, causing larger matches in a single iteration.

3.3.3 APSARA

APSARA is an algorithm that requires just a single iteration to match the performance of the maximum weight matching algorithm. Here, we describe the randomized variant of APSARA as presented in [19].

> *At the end of time $t-1$, the scheduler keeps in memory the schedule $S(t-1)$. At the following time step t, it randmonly picks d* **neighbours**[1] *of $S(t-1)$ $(N_d(S(t-1)))$ and computes their weights in parallel. The matching that has the maximum weight among $N_d(S(t-1)) \cup S(t-1)$ is used as the schedule for time t, $S(t)$. I.e,*

$$S(t) = arg\ max_{M \in N_d(S(t-1)) \cup S(t-1)} \{W(M)\}$$

This version of APSARA has also been shown to have 100% throughput and good delays under admissible, i.i.d traffic in [19].

3.3.4 Lowest Occupancy Output First Algorithm (LOOFA)

LOOFA is also an iterative algorithm which iterates the following steps till no more matches can be made

1. Each unmatched input sends a request to an output with the lowest occupancy among those for which it has at least one queued cell.

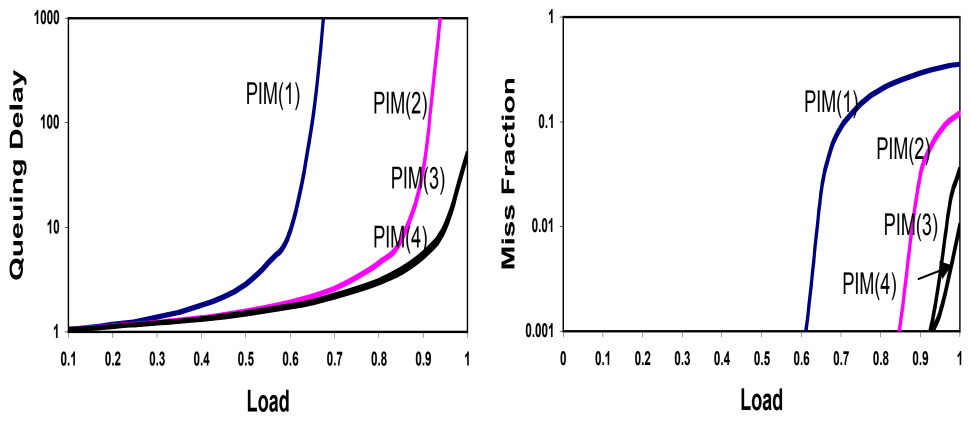
2. Each output, upon receiving requests from multiple inputs, selects one and sends a grant to that input.

It has been proven that a switch using LOOFA with a speedup of 2 is work conserving under all traffic conditions [34]. But the algorithm requires $O(N)$ iterations to obtain a maximal matching. This means that, unlike PIM and iSLIP, the algorithm can have poor performance when used with fewer iterations. For example, under uniform traffic and heavy loads when all inputs have cells queued for all outputs, all the inputs send requests to the same output in a single iteration! Such behaviour makes LOOFA unsuitable for use in high speed implementations where there is only time to perform a few iterations.

[1]Refer section 2.3.9 for the definition of a neighbour

3.4 Performance Evaluation

3.4.1 Miss fraction vs Delay



(a) Delays (b) Miss Fraction

Figure 3.3: Average delays and miss fractions for various iterations of PIM, N=16, speedup=1.0

In this section, we study the performance of PIM and iSLIP under uniform traffic for varying numbers of iterations. The speedup of the switch is 1.0 implying that all queuing is done at the inputs. Fig. 3.3 and Fig. 3.4 show the difference between using miss fraction and average queuing delay as a metric versus offered load for PIM and iSLIP, respectively.

While the maximum load carried by a scheduler can be determined from the delay plot (by looking for the point at which queuing delays become unbounded), the miss fraction curves also indicate the throughput achieved by the algorithms at all loads. It can be inferred from Fig. 3.3(a) that the queuing delays are unbounded for PIM for load > 0.63 and that the miss fraction for PIM in Fig. 3.3(b) increases to 0.36 for an offered load of 1. This is in agreement with the fact that the throughput of PIM is limited to 63% (for $N = 16$) for a single iteration. iSLIP on the other hand performs much better even with a single iteration. The miss fraction curve of iSLIP in Fig. 3.4(b) shows that for load > 0.63 the rate of increase of miss fraction actually shows a sharp decrease. This is due to the *desynchronization effect* of iSLIP which comes into play when almost all VOQs have non-zero backlogs which happens for load > 0.63. The desynchronization effect is also resposible for the *linear* increase in miss fraction of iSLIP.

3.4.2 Varying Speedup and Number of Iterations

As mentioned, one of the advantages of using simulation studies is that we can test the performance of scheduling algorithms under various implementation optimizations. Fig. 3.5 shows the performance of PIM with 1 and 4 iterations under various traffic patterns with a speedup of 1. Clearly, while PIM with 4 iterations has good throughput under uniform traffic, it still performs poorly under non-uniform traffic patterns like the diagonal and log-diagonal traffic patterns. Fig. 3.6 shows the performance of iSLIP with 1 and 4 iterations under various traffic patterns with a speedup of 1. Again, we note that algorithms tend to have similar performance under uniform and hotspot traffic though the latter is a non-uniform traffic pattern. Clearly, iSLIP has good performance under these traffic patterns even with a single iteration but like, PIM, performs poorly under diagonal and log-diagonal traffic patterns.

Fig. 3.7 shows that the poor performance of PIM under non-uniform traffic patterns can be overcome by using increased speedup in the switch. In fact, PIM with a single iteration requires a speedup of < 1.7 and PIM with four iterations requires a speedup of < 1.3 to have negligible miss fractions for all these traffic patterns! Like PIM, iSLIP also has greatly improved performance under all traffic patterns with increased speedup as shown in Fig. 3.8. With just a speedup of 1.5 for iSLIP with a single iteration and a speedup of < 1.3 for iSLIP with four iterations, the miss fractions for these algorithms are almost negligible.

There are results that demonstrate that all maximal size matching algorithms (including PIM and iSLIP) have 100% throughput for admisible, i.i.d traffic [27]. But, one of the conclusions that can be drawn from these studies is that, with sufficient speedup, both PIM and iSLIP can be executed for just a single iteration and still have good performance under these admissible traffic patterns!

Fig. 3.9 shows the performance of APSARA with varying loads under various admissible trafic patterns. Clearly, APSARA (which requires just a single iteration) has good performance under all traffic patterns even for a speedup$= 1$.

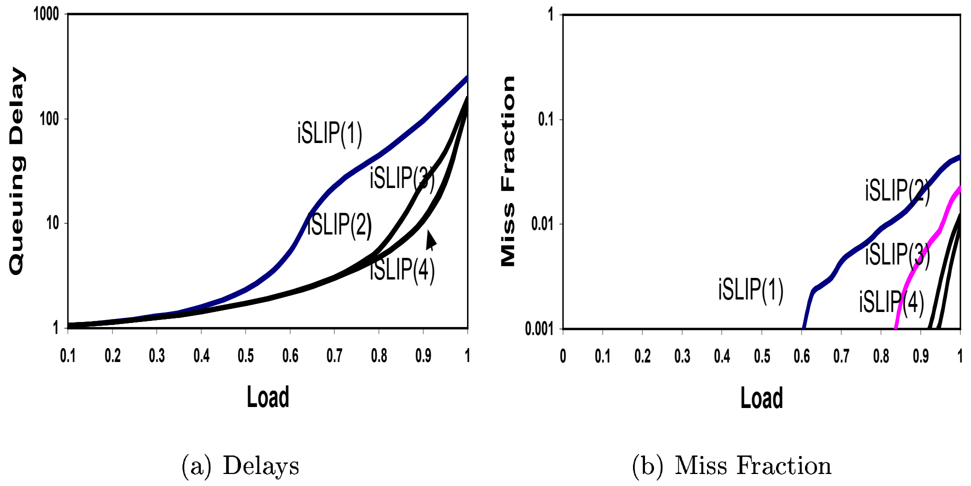


Figure 3.4: Average delays and miss fractions for various iterations of *i*SLIP, N=16, speedup=1.0

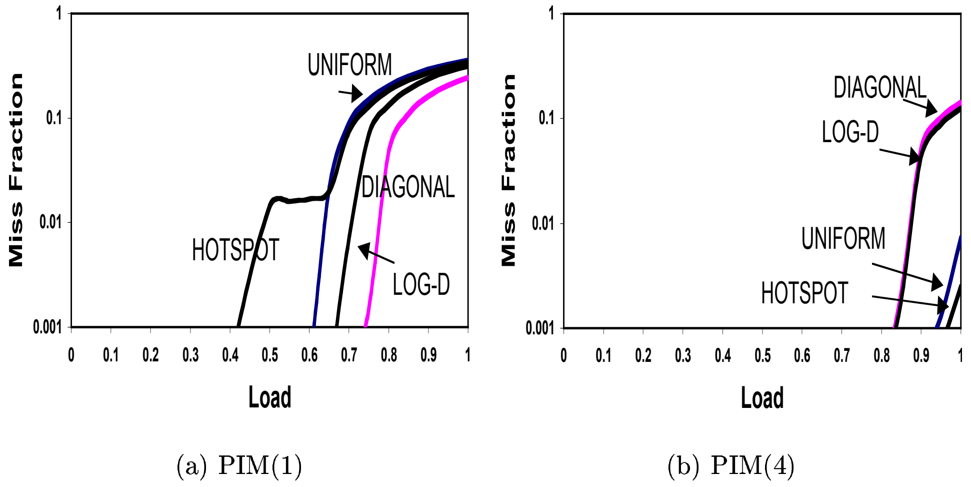
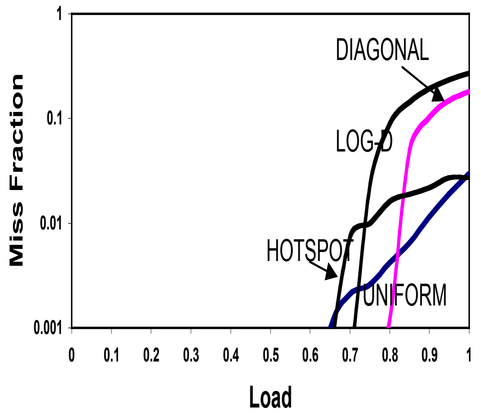
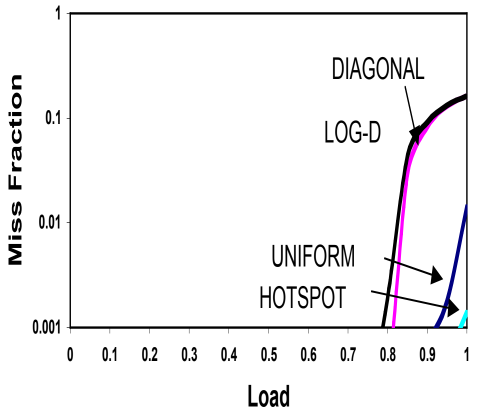


Figure 3.5: Miss fractions for various iterations of PIM (N=16, speedup=1.0), with varying load, under various *admissible* traffic patterns.

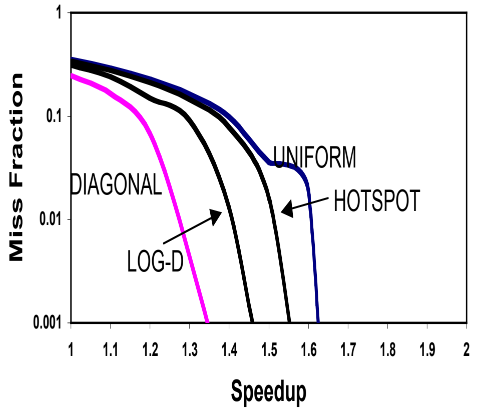


(a) iSLIP(1)

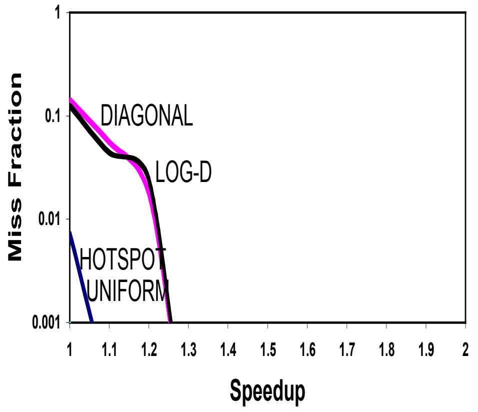


(b) iSLIP(4)

Figure 3.6: Miss fractions for various iterations of *i*SLIP (N=16, speedup=1.0), with varying load, under various *admissible* traffic patterns.

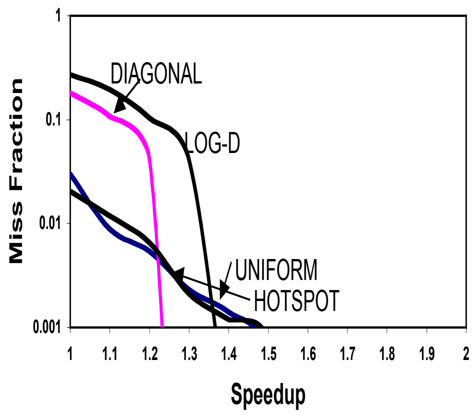


(a) PIM(1)

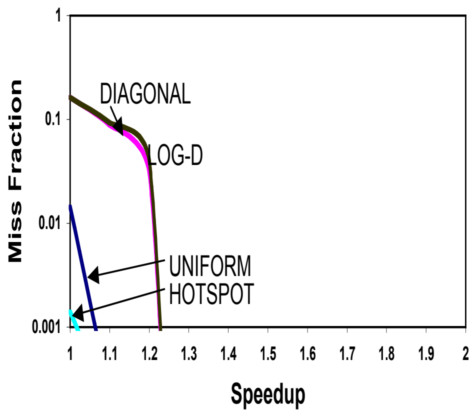


(b) PIM(4)

Figure 3.7: Miss fractions for various iterations of PIM (N=16, load=1.0) with varying speedup, under various *admissible* traffic patterns.



(a) iSLIP(1)



(b) iSLIP(4)

Figure 3.8: Miss fractions for various iterations of *i*SLIP (N=16, load=1.0), with varying speedup, under various *admissible* traffic patterns.

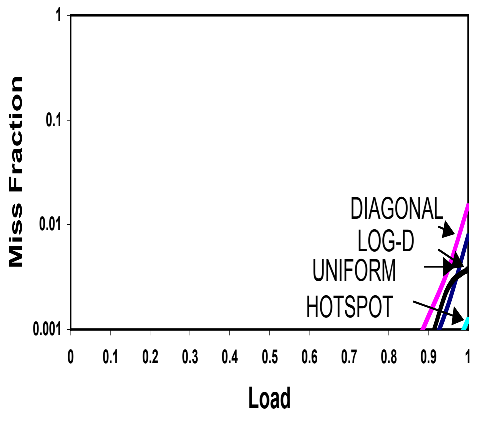
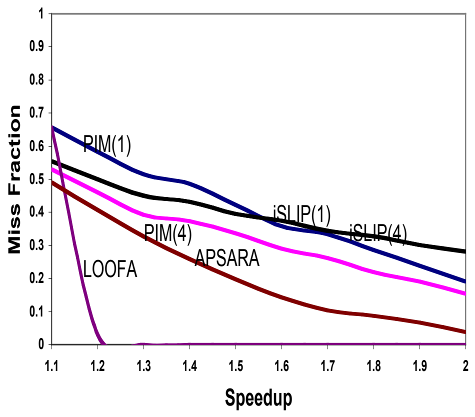
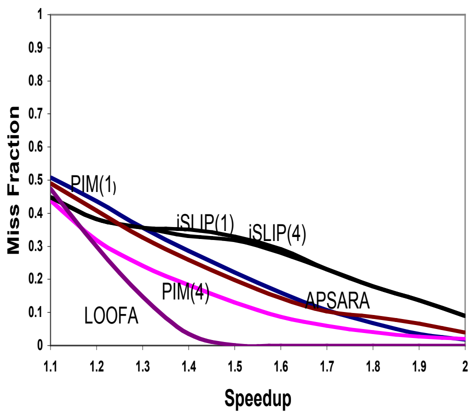


Figure 3.9: Performance of APSARA (with d = 20) under varying loads (N=16, speedup = 1.0), under various *admissible* traffic patterns



(a) Test A



(b) Test B

Figure 3.10: Miss fractions for PIM, iSLIP and LOOFA under stress test with 5 participating inputs and 12 phases.

3.4.3 Stress Test

The admissible traffic patterns can obscure severe shortcomings of algorithms when exposed to extreme traffic conditions. In this section, we use the stress test to study the performance of the algorithms under more demanding overload condtions.

To compare the performance of the algorithms under the stress test, recall that the transitions times and the measurement periods of the test have to be fixed for all algorithms. To determine these basic parameters, we subject one of the algorithms to a stress test where the transitions between various phases takes place when the backlog of the VOQ at an input to a target output equals that of the backlog of a VOQ to the previous target output. In all these stress tests, the initial backlog threshold (inputs switch to phase 2 on building this backlog to output 0) is set to 10000. We then compare the performance of the rest of the algorithms under the same test with these basic parameters.

Fig. 3.10(a) and Fig. 3.10(b) compares the performance of PIM, iSLIP, APSARA and LOOFA under a stress test with 5 participating inputs and 12 phases at various speedups. In Fig. 3.10(a) the basic parameters were determined to create a worst case traffic scenario for PIM(4) under a speedup of 2.0. We denote the test with these parameters as Test A. As can be noted, PIM and iSLIP have poor performance under the stress test. iSLIP has almost the same performance for iterations 1 and 4 and has miss fraction of 30% even at a speedup of 2. PIM shows improvement with increase in iterations from 1 to 4 but still has a higher miss fraction at various speedups when compared to LOOFA. LOOFA has been proven to be work conserving for unlimited iterations under all traffic conditions for a speedup of 2. For this particular test, LOOFA needs a speedup of just 1.3 to eliminate all underflow.

Fig. 3.10(b) compares the performance of the algorithms under a stress test where the basic parameters were determined to create worst case traffic for LOOFA at a speedup of 2.0 (Test B). Again, LOOFA has zero miss fraction for speedups > 1.5 showing that the output ordering based on queue lengths in LOOFA effectively reduces all underflow even in extreme conditions. Although the same test doesn't cause PIM and iSLIP to perform as badly as in Fig. 3.10(a), the performance of these algorithms does not match that of LOOFA.

These tests demonstrate the underperformance of widely used algorithms like PIM and iSLIP under overload traffic conditions.

3.4.4 Conclusions

In the previous sections, we have studied the performance of PIM, iSLIP, APSARA and LOOFA under a variety of traffic conditons. We have found that

1. PIM and iSLIP can in fact, have good performance under all admissible traffic conditions even with a single iteration when used with sufficient speedup.
2. APSARA has good performance under admissible traffic with a speedup of 1.0 but has poor performance under more demanding, inadmissible traffic conditions.
3. LOOFA has good performance under all traffic patterns with modest speedups but only serves as a performance benchmark because of its high implementation complexity.

3.5 Stress Resistant Algorithms

The better performance of LOOFA under the stress test suggests that biasing outputs to favour those with smaller queue lengths is the key to maintaining throughput even under extreme traffic conditions. Unfortunately, complete ordering of outputs and the large number of sequential iterations needed to use this ordering can themselves be obstacles to implementing such algorithms at high speeds. But, we note that the traffic conditions that are under consideration here are essentially persistent traffic conditions and that algorithms which achieve and use even approximate or partial ordering of outputs can perform significantly better than those that do not take output backlogs into consideration at all. In this section we introduce simple heursitics to modify all the discussed scheduling algorithms to improve their performance under the stress test while still keeping them simple enough to implement.

1. For PIM and iSLIP which attempt to approximate the performance of a maximum size matching, we introduce a Lowest Layer Selection (LLS) heuristic which achieves a *coarser* ordering of outputs based on their queue lengths. We use LLS to design stress resistant variants of PIM and iSLIP calles LLS-Random (LLS-R) and LLS-Slip (LLS-S).

2. For algorithms like APSARA that attempt to converge on a maximum weight matching, we present improved weight metrics that include output queue lengths in their computation and present algorithms like Shortest Output Longest Input First (SOLIF), which have improved performance under the stress test.

3. For work-conserving algorithms like LOOFA, we introduce an odd-even sorting heuristic which achieves only an approximation of the ideal ordering of outputs but converges to the ideal ordering under persistent traffic conditions. We use the odd-even sorting technique to design an approximate version of LOOFA called approximate LOOFA (A-LOOFA).

3.6 Lowest Layer Selection

PIM and iSLIP perform poorly compared to LOOFA under the stress test because they ignore output occupancies. On the other hand, they perform well under uniform traffic and also require fewer iterations to converge making them more suitable for high speed implementations.

In this section, we describe a simple low-cost mechanism that can be used to make PIM and iSLIP *stress resistant*. The improved algorithms have the same performance under uniform traffic and have greatly improved performance under the stress test. The idea is to prioritize the outputs based on their queue lengths since, underflow occurs only when an output queue is empty. The various outputs of the switch are divided into *layers* based on their queue length using an exponentially graded scale. Fig. 3.11 shows an instance of such a scale with 16 layers. In this scale, queues with length ≤ 8 are put in layer 0, queues with length > 8 and ≤ 16 are put in layer 1 and so on. The queue length corresponding to a layer i is given by 8×2^i. The last layer of the scale (15) holds all queues with lengths $> 8 \times 2^{14}$, indicating that the scale doesn't differentiate between outputs with the largest queue lengths. Thus the layering of queue lengths

- achieves a coarser ordering of outputs based on queue lengths.
- bigger layers are used for larger queue lengths, indicating that there is less chance of underflow at outputs with large queue lengths.
- beyond a queue length limit (indicated by the final layer), all outputs are treated equally as it is not necessary to order outputs with large queue lengths to avoid underflow.

Hence, the number of layers itself is independent of the number of ports of the switch (N) making it possible to use a single scale with 8 to 16 layers for high speed switches with large numbers of ports. Also, the layers to which various queues belong can be trivially updated whenever cells are added or removed from the various queues.

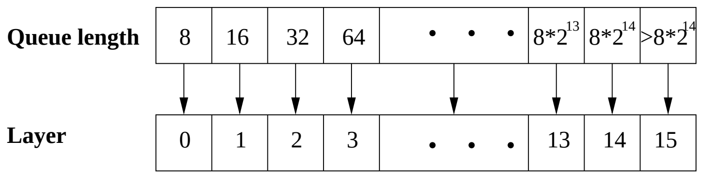


Figure 3.11: Exponentially graded scale used in assigning outputs to layers based on their queue length.

Algorithms use the layers to which the various outputs belong by employing a Lowest Layer Selection (LLS) heuristic. The algorithms (LLS-R and LLS-S) in their accept phase give priority to outputs in the lowest layer. Thus, the use of the Lowest Layer Selection heuristic in these algorithms introduces a bias towards outputs with smaller queue lengths. The exponential scale used in defining the layers determines the extent of this bias, since the algorithms still show their default behaviour to outputs which belong in the same layer. A scale which has *thin* (and hence, more) layers, forces the inputs to always accept outputs with smaller queue lengths and a scale with *thick* layers causes the inputs to pick outputs randomly (in case of PIM) or in a round-robin order (in case of *i*SLIP) irrespective of the queue lengths of the outputs since, outputs more often than not will belong to the same layer.

The heuristic itself can be implemented at negligible cost by maintaining per input *grant vectors*. These vectors have 1 bit corresponding to each layer. When an output sends a grant to an input in a scheduling algorithm, it also sets the bit corresponding to the layer to which its queue length belongs, in the input's grant vector. The input can then easily find the lowest layer of all granting outputs by using a priority encoder to find the first bit set to 1 in the grant vector. Also, for crossbars of moderate size (32 ports), we can quickly determine the output with the smallest layer index using an N-way minimum finding circuit.

3.6.1 Lowest Layer Selection - Random (LLS-R)

LLS-R is an interative matching algorithm which uses the LLS heuristic and per input grant vectors to improve the performance of PIM under the stress test. The LLS-R algorithm iterates the following three steps.

1. Each unmatched input makes a request to every unmatched output for which it has a queued cell.

2. If an unmatched output receives any requests, it chooses one randomly to grant.

3. Inputs use their grant vectors to determine the lowest layer among all the granting outputs and accept an output belonging to this layer and notify that output.

It is evident that the algorithm is actually similar to Parallel Iterative Matching in the first two steps and differs only in the *accept* phase where the inputs pick an output from the lowest layer. Hence, the algorithm will still converge on a maximal match in $O(\log N)$ iterations. The proof of this claim is similar to the argument made in [6] for PIM.

3.6.2 Lowest Layer Selection - Slip (LLS-S)

LLS-S is a variant of the *i*SLIP algorithm obtained by using the LLS heuristic and per input grant vectors. The algorithm iterates the following three steps

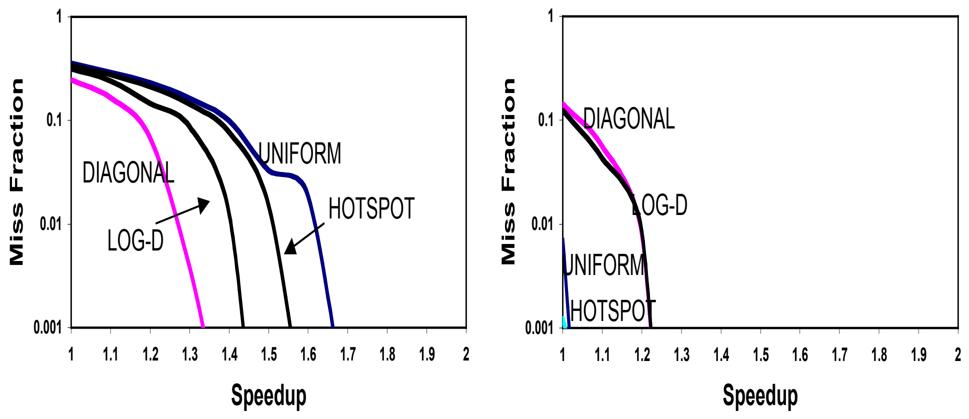
1. Each unmatched input sends a request to every unmatched output for which it has a cell queued.

2. If an unmatched output receives any requests, it chooses one that appears next in a fixed, round-robin schedule starting from its input pointer. The output notifies each input whether or not its request was granted. The input pointer of the round-robin schedule is incremented (modulo N) to one location beyond the granted input if and only if the grant is accepted in step 3 of the first iteration. The pointer is not incremented in subsequent iterations.

3. Inputs use their grant vectors to determine the lowest layer among all granting outputs, and accept an output from this layer, that appears next in a fixed, round-robin schedule starting from their output pointer. The output pointer is then incremented (modulo N) to one location beyond the accepted output.

Again, we note that LLS-S differs from *i*SLIP only in the last step where the inputs select one of multiple outputs belonging to the lowest layer.

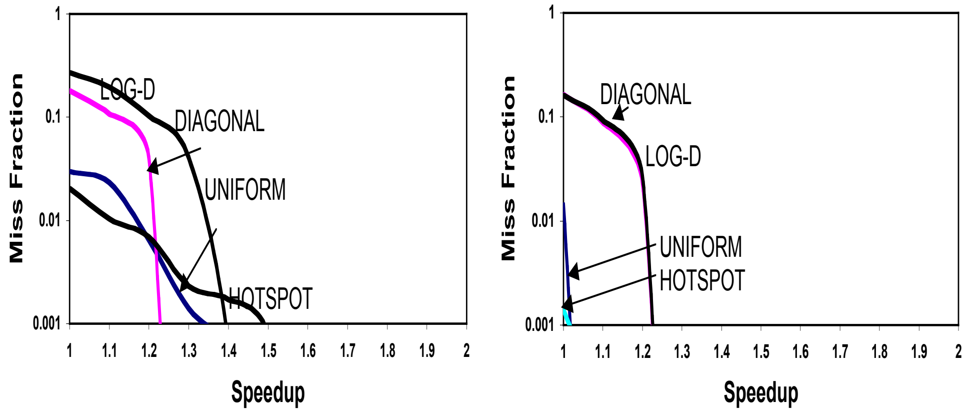
3.6.3 Performance Evaluation



(a) LLSR(1) (b) LLSR(4)

Figure 3.12: Miss fractions for various iterations of LLSR (N=16, load=1.0), with varying speedup, under various *admissible* traffic patterns.

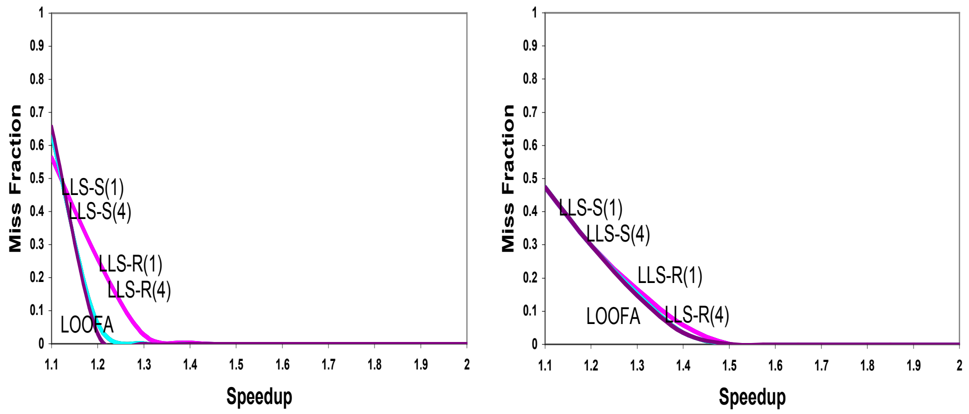
We first note that in purely input queued switches (speedup = 1.0), the algorithms, LLS-R and LLS-S behave exactly like PIM and *i*SLIP, since all output queue lengths are 0. This causes all the outputs to belong to the same layer (layer 0) and the algorithms follow their default behaviour. This observation is also approximately



(a) LLSS(1) (b) LLSS(4)

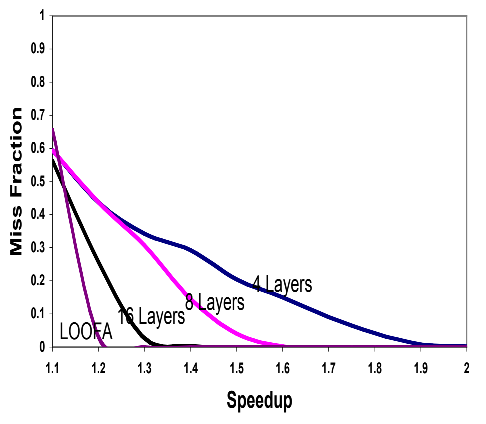
Figure 3.13: Miss fractions for various iterations of LLSS (N=16, load=1.0), with varying speedup, under various *admissible* traffic patterns.

true for uniform random traffic under any speedup, since, all the outputs have approximately the same queue lengths and hence belong to the same layer. Thus, the performance of the LLS-R and LLS-S algorithms under uniform random traffic at any load and speedup is similar to the performance of PIM and *i*SLIP respectively. Fig. 3.12 and Fig. 3.13 show the performance of these algorithms under admissible traffic patterns at speedups greater than 1. LLS-R and LLS-S show marginal improvement over PIM and *i*SLIP with increased speedup.

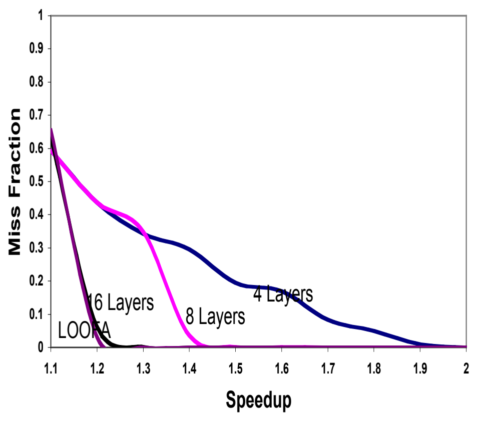


(a) Test *A* (b) Test *B*

Figure 3.14: Miss fractions for LLS-R, LLS-S (using 16 layers) and LOOFA under stress test with 5 participating inputs and 12 phases.



(a) LLS-S



(b) LLS-R

Figure 3.15: Miss fractions for LLS-R and LLS-S (single iteration) with varying layers under stress test with 5 participating inputs and 12 phases (Test A)

Fig. 3.14 compares the performance of LLS-R, LLS-S and LOOFA under a stress test with 5 participating inputs and 12 phases. Under both tests A and B, the algorithms show greatly improved performance over PIM and iSLIP shown in Fig. 3.10. With just a single iteration, LLS-R has zero miss fraction for speedup $\geq$ 1.4 (Fig. 3.14(a)) and has similar performance even with 4 iterations. LLS-S also shows greatly improved performance very similar to that of LOOFA even for a single iteration. Under Test B (Fig. 3.14(b)), the algorithms have almost identical performance.

Fig. 3.15 compares the performance of LLS-S and LLS-R algorithms with varying number of layers. As can be seen from the figures, the algorithms show improvement with increasing number of layers and have performance comparable to that of LOOFA with 16 layers. This comparison of the performance of these algorithms with that of LOOFA shows that these simple algorithms can potentially provide high throughputs even in overload situations even by using only approximate output ordering schemes.

3.7 Using Output Backlogs in Edge Weights

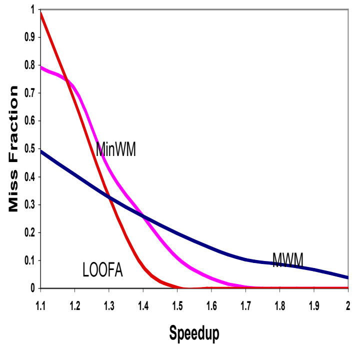
Our primary goal in this section is to use output queue lengths in the computation of weights of the various edges of the bipartite graph in the scheduling problem. The better performance of LOOFA suggests that an edge weight that creates a bias towards outputs with smaller queue lengths can improve the performance of the algorithms under overload conditions.

3.7.1 Minimum Weight Matching (MinWM)

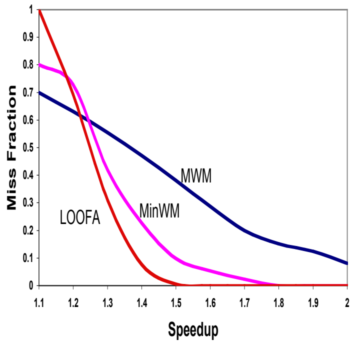
A simple way to introduce a bias towards outputs with smaller queue lengths is to use a *Minimum Weight Maximum Size* (MinWM) matching algorithm on a bipartite graph whose edge weights are just the corresponding output backlogs ($B(j)$). I.e,

$$w(i,j) = B(j)$$

A minimum weight maximum size matching algorithm finds a matching with the minimum weight among all those which have a maximum possible size. This problem can be solved by solving the *minimum cost maximum flow* problem in an equivalent flow graph [62]. We can use the minimum cost augmentation method to find a minimum cost maximum flow (of capacity f) in an acyclic network with integer capacitites, in $O(fn \log n)$ time as compared to $O(N^{2.5} \log N)$ for maximum weight matching (MWM). Although, this is more complex than LOOFA, the simpler approximation of MWM with APSARA suggests that an implementable version of MinWM is also possible.

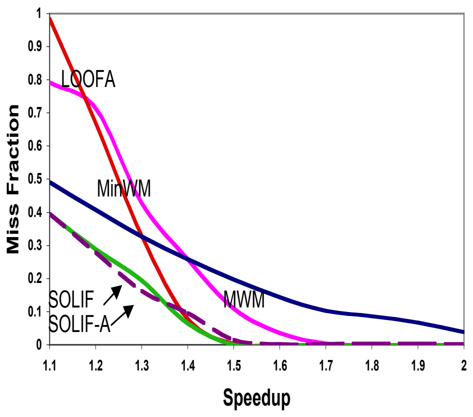


(a) Stress test with parameters derived from LOOFA at speedup=2.

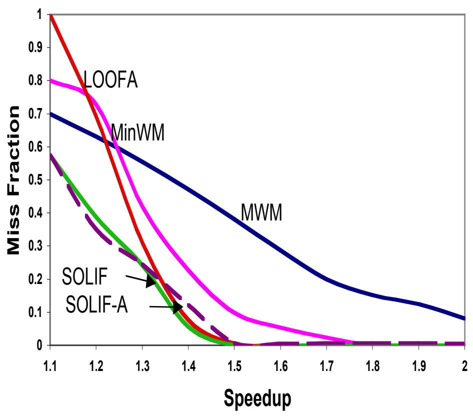


(b) Stress test with parameters derived from MWM at speedup=2.

Figure 3.16: Miss fractions for MWM, MinWM and LOOFA, with varying speedup, under two stress tests.



(a) Stress Test with parameters derived from LOOFA at speedup=2.



(b) Stress Test with parameters derived from MWM at speedup=2.

Figure 3.17: Miss fractions for SOLIF, MWM, MinWM and LOOFA, with varying speedup, under two stress tests.

Fig. 3.16 compares the performance of LOOFA, MWM and MinWM. Clearly, MinWM has improved performance over MWM and in fact, has performance comparable to that of LOOFA. We also note that both LOOFA and MinWM have high miss fractions at lower speedups. This is because, the output queue lengths are very small at these speedups and do not effectively represent the state of the system. On the other hand, MWM has a relatively small miss fraction compared to LOOFA and MinWM at these speedups, suggesting that perhaps algorithms that take both input and output queue lengths into consideration in weight computations will perform well at all speedups.

3.7.2 Shortest Output Longest Input First (SOLIF)

SOLIF is a maximum weight matching algorithm that uses both the input and output queues lengths in its weight computation. In SOLIF,

$$w(i,j) = \begin{cases} B(i,j) + B_{max} - B(j) & \text{if, } B(i,j) > 0 \\ 0 & \text{otherwise} \end{cases} \tag{3.1}$$

where, B_{max} is the maximum of all queue lengths $B(j)$. Clearly, for speedup= 1.0, the algorithm is similar to LPF (because all output backlogs are zero) and

$$w(i,j) = B(i,j)$$

At higher speedups, the algorithm gives higher weight to edges whose outputs have shorter queue lengths. The complexity of SOLIF is the same as that of the MWM algorithm ($O(N^{2.5} \log N)$). Just as APSARA is a simple, single iteration approximation to MWM, we can use a simple, derandomized algorithm with memory (called SOLIF-APSARA (SOLIF-A)) to approximate SOLIF.

It is important to realize that APSARA can approximate MWM so well because, the weights in MWM ($w(i,j) = B(i,j)$) change slowly with time. The lengths of the input queues (weights) can increase by a maximum of 1 and decrease by S in a single cycle. The same is not true for the weights defined in equation 3.1, because of the use of B_{max}. Hence, we redefine the weights used in SOLIF-A as

$$w(i,j) = \begin{cases} 0 & \text{if } B(i,j) = 0 \\ B(i,j) + C - B(j) & \text{if, } B(j) \leq C \\ B(i,j) & \text{otherwise} \end{cases} \tag{3.2}$$

The use of the constant C (called *output threshold*) influences the algorithm in the following ways

1. All edges to outputs with queue length $> C$ do not get any additional weight. This is to reflect the fact that these outputs are less likely to underflow given their high output queue lengths. Thus, the constant C itself is chosen based on the individual link rates. For a link rate of 10 Gb/s, an output threshold $C =$ 1 MByte, ensures that an edge to an output isn't given any additional weight when it is certain that it won't experience underflow in the next 2×10^4 cycles (with minimum cell size of 50 bytes).

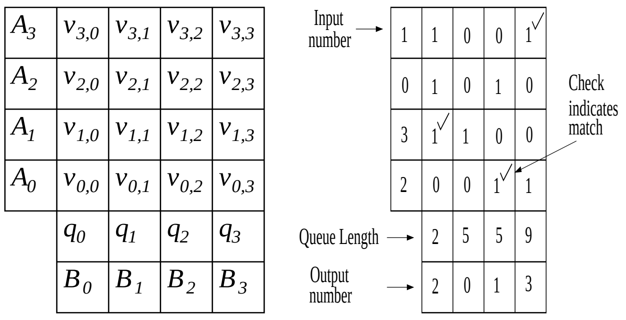
2. Because of the use of C (instead of B_{max}), in the weight computation, the weights of the edges can increase by a maximum of 2 or decrease by $2 \times (S-1)$, thus, allowing the use of a derandomized APSARA like algorithm with memory.

Fig. 3.17 compares the performance of SOLIF with LOOFA, MinWM and MWM. SOLIF has greatly improved performance over MWM (better than LOOFA!) under the stress tests at varying speedups. SOLIF-A is a single iteration algorithm that is similar to APSARA but uses the weight computation defined in 3.2. Fig. 3.17 also shows that SOLIF-A performs as well as SOLIF and has greatly improved performance over MWM and APSARA at all speedups under the stress tests.

3.8 Approximate LOOFA (A-LOOFA)

Although LOOFA itself is too complex for high speed implementation, it can be used as the basis for an algorithm that is practical and which in practice, can provide very similar performance. This algorithm, which we call *Approximate LOOFA* (A-LOOFA), can be implemented in hardware in a way that makes it suitable for routers with 10 Gb/s links.

3.8.1 Hardware implementation of matching


A_3 $v_{3,0}$ $v_{3,1}$ $v_{3,2}$ $v_{3,3}$
A_2 $v_{2,0}$ $v_{2,1}$ $v_{2,2}$ $v_{2,3}$
A_1 $v_{1,0}$ $v_{1,1}$ $v_{1,2}$ $v_{1,3}$
A_0 $v_{0,0}$ $v_{0,1}$ $v_{0,2}$ $v_{0,3}$
q_0 q_1 q_2 q_3
B_0 B_1 B_2 B_3
Input number
1 1 0 0 1√
0 1 0 1 0
3 1√ 1 0 0
2 0 0 1√ 1
Check indicates match
Queue Length
2 5 5 9
Output number
2 0 1 3

(a) Hardware Components

(b) Example Operation

Figure 3.18: Principal hardware components and example operation of A-LOOFA

Fig. 3.18 illustrates the basic concept behind A-LOOFA and its implementation. At the left, we have a set of *row registers*, A_i $(0 \leq i \leq N-1)$, each containing the number of some input. At the bottom, we have a set of *column register pairs*, (B_j, q_j) $(0 \leq j \leq N-1)$ each containing the number of an output (B_j) and its associated output queue length (in cells). The central area contains an $N \times N$ array of *VOQ occupancy bits* $v_{i,j}$ where $v_{i,j} = 1$ if and only if input A_i has one or more cells to send to output B_j. A-LOOFA attempts to maintain the set of column register pairs in sorted order, so that $q_0 \leq q_1 \leq \cdots \leq q_{N-1}$. As will be explained shortly, it only approximates the sorted order, in order to avoid a time-consuming sorting step.

Matching in A-LOOFA is accomplished using a simple combinational circuit. This circuit effectively implements the N step iterative matching process required by LOOFA. While it requires $O(N)$ time to complete, the constant factor is determined by gate delays, making it small enough to allow for high speed implementation.

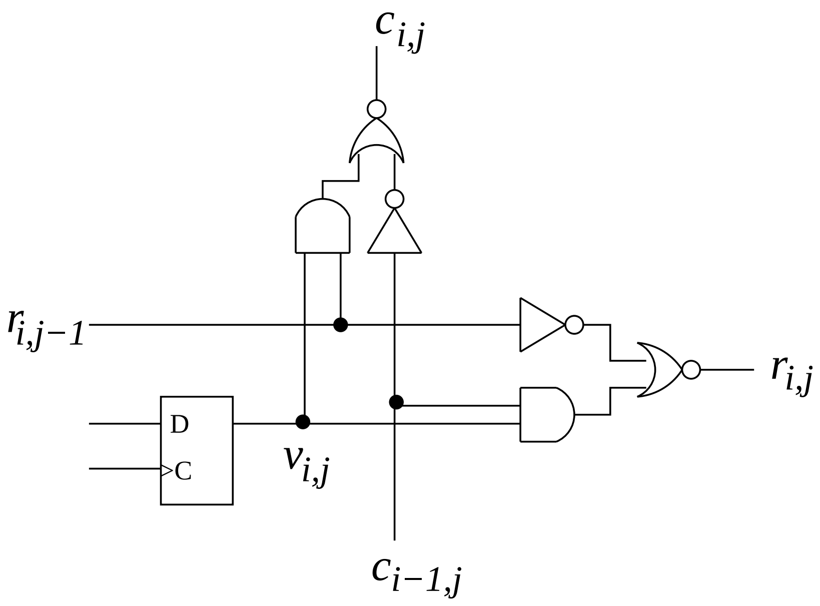


Figure 3.19: Match Logic

Fig. 3.19 shows the match logic that is associated with the VOQ occupancy bit $v_{i,j}$. The input signals $r_{i,j-1}$ and $c_{i-1,j}$ are high if output B_j is available for selection by input A_i. If both are high and $v_{i,j} = 1$ then A_i is matched with B_j and $r_{i,j}$ and $c_{i,j}$ are both pulled low. So,

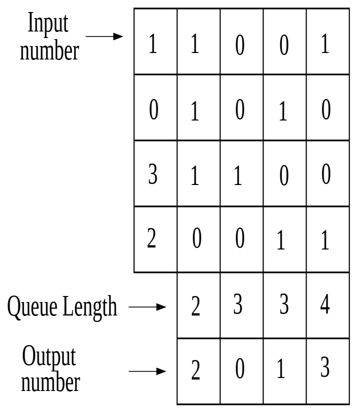
$$\begin{aligned} r_{i,j} &= r_{i,j-1}(\overline{v}_{i,j} + \overline{c}_{i-1,j}) = \overline{(\overline{r}_{i,j-1} + v_{i,j}c_{i-1,j})} \\ c_{i,j} &= c_{i-1,j}(\overline{v}_{i,j} + \overline{r}_{i,j-1}) = \overline{(\overline{c}_{i-1,j} + v_{i,j}r_{i,j-1})} \end{aligned}$$

To complete a matching operation, these signals must propagate throughout the $N \times N$ array, but note that signals propagate upward and to the right, so the delay is $2N$ times the delay in each block, with each block contributing two gate delays. For a modern .13 μm ASIC process, the gate delays are 25-50 ps, allowing a match to be completed in 3.2-6.4 ns (for $N = 32$). A router with 10 Gb/s links and a speedup of 2 will need to complete a crossbar scheduling operation every 20 ns, making the matching delay small enough to allow for high speed implementation.

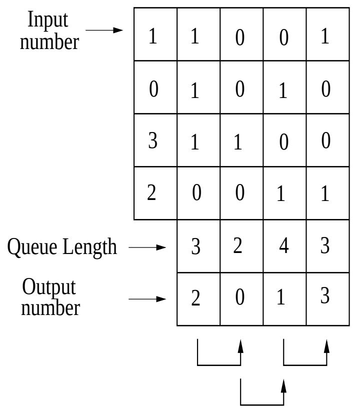
Fig. 3.18(b) shows an example operation of the A-LOOFA circuit. In this example, output 2 which has the shortest queue length has been moved to the first column. Output 2, hence, gets to pick first and chooses input 3 that has a cell destined for it. Output 0 which comes next in the order of increasing queue lengths can no longer pick input 3. Similarly, output 1 picks input 2 and output 3 picks input 1.

3.8.2 Odd-Even Sorting

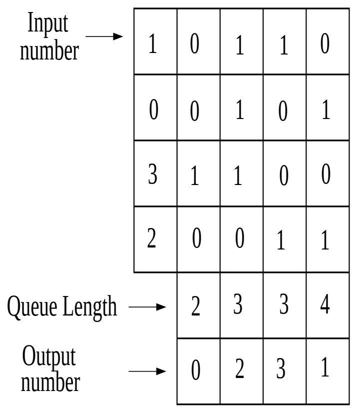
In order for the approach described to exactly implement LOOFA, it's necessary to maintain the column register pairs in sorted order. This is not practical in a high speed implementation. Fortunately, we can still get good (although not provably work-conserving) performance without sorting. Because the queue lengths change slowly, we can maintain an approximate sorted ordering by doing a pair of nearest neighbor swaps (odd-even sorting) after each crossbar scheduling operation. Specifically, for all even $j < N$, we exchange the values of B_j and q_j with B_{j+1} and q_{j+1} if $q_j > q_{j+1}$. Then for all odd $j < N - 1$, we exchange the values of B_j and q_j with B_{j+1} and q_{j+1} if $q_j > q_{j+1}$. Whenever we perform such an exchange, we also exchange the values of the VOQ occupancy bits in the corresponding columns.



(a) Initial state



(b) Change in output queue length



(c) After odd-even sort

Figure 3.20: Example demonstrating the use of the odd-even sorting technique.

Fig. 3.20 shows an instance of the progress of the odd even sorting technique. The initial state of the system is shown in Fig. 3.20(a). In a given cycle, the output queue lengths can change by at most 1. The new queue lengths are shown in Fig. 3.20(b). By just swapping the odd and even neighbours the outputs are rearranged in inceasing order of queue lengths as shown in Fig. 3.20(c). Note that when outputs are swapped, the corresponding columns are also swapped.

3.8.3 Input permutation

The combinational matching circuit favors inputs that occupy "lower" rows in the array of VOQ occupancy bits. To ensure fairness among the different inputs, we randomly permute the rows of the array at the end of each crossbar scheduling operation (both the row registers and the VOQ occupancy bits). Specifically, for all even values of $i < N$, we generate a pseudo random bit x_i. This is easy to do in hardware. If $x_i = 0$, then all the values in row i are moved to row $i/2$ and the values in row $i+1$ are moved to row $(N+i)/2$. If $x_i = 1$, then all the values in row i are moved to row $(N+i)/2$ and all values in row $i+1$ are moved to row $i/2$. This permutation scheme is based on the well-known perfect shuffle, is easy to implement and ensures long-term fairness. Fig. 3.21 shows an instance of the random permutation of inputs. Note that when inputs are moved, the corresponding rows are also shifted.

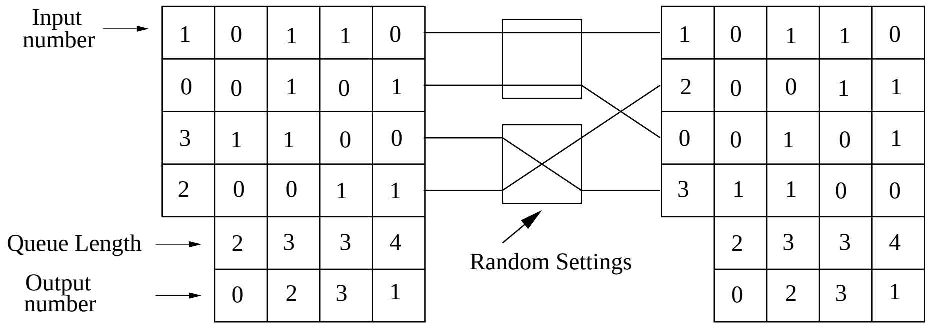


Figure 3.21: Example showing the permutation of inputs.

3.8.4 Implementation issues

There are a few other issues that need to be addressed to complete the description of the implementation. First, when we get a match, we need a way to pass the identity of the matching input to the circuitry that controls the output, so that the appropriate crossbar control signals can be asserted. This requires a $\lg N$ bit wide data path for each row and column of the array and a switch that forwards the value on row i to column j if there is a match at location (i, j). Second, we need a way to load new values in the VOQ occupancy bit. To do this, the circuitry controlling an input sends an output number along its row, which is compared at each location (i, j) to j. At the location where these values match, the VOQ bit is selected to receive a new value.

Finally, we need to maintain a connection between the IO pins of the device and the registers associated with each input and output. Since the pins of the device have a fixed association with specific inputs and outputs, we need to maintain connections between these fixed pin locations and the associated registers, which are constantly exchanging values, as the algorithm proceeds. This requires two special purpose crossbars, one on the input side and one on the output side. The crosspoint settings in these crossbars change with each row and column swap to maintain the required connections to the fixed IO pins. The output side crossbar carries a two bit signal from the output pins, indicating whether a given output queue length is to increase by one, decrease by one or stay the same. The input side crossbar carries a $2 + \lg N$ bit signal indicating whether the VOQ occupancy bit for a specified ouput is to be set, reset or stay the same (note that during one operation cycle, only two VOQs at any input can change their status).

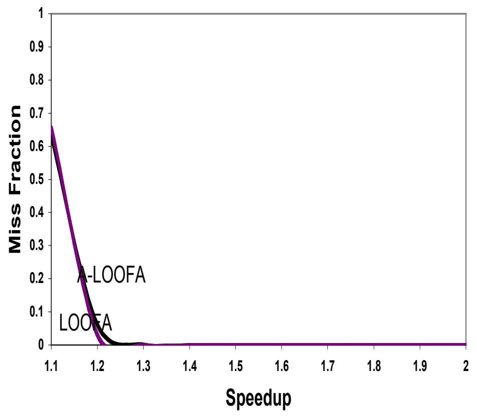
The gate complexity of the A-LOOFA control circuit has the form $C_1 N^2 \lg N + C_2 N^2 + C_3 N \lg N + C_4 N$, for constants $C_1, \ldots, C_4$. We estimate $C_1 \approx 10$, $C_2 \approx 30$, $C_3 \approx 50$ and $C_4 \approx 20$, yielding an overall estimate of less than 90,000 gates. While not a trivial circuit, to be sure, it is well within the capabilities of modern ASICs.

We note that similar high speed implementations (with conceivably the same execution times) can be achieved using the *i*MCRA (*i*POINT Multicast Contention Resolution Algorithm) [38]. This algorithm reduces the execution time by using an efficient pipelined implementation. The swapping of rows and colums in A-LOOFA (when input or output order is changed) makes it unsuitable for pipelined implementation. However, A-LOOFA and *i*MCRA are also structurally different because the basic entities used in A-LOOFA correspond to VOQs (which causes the execution of the algorithm to progress along the *diagonal* of the VOQ *occupancy bit matrix* leading to smaller execution time) whereas the entities in *i*MCRA correspond to input ports.

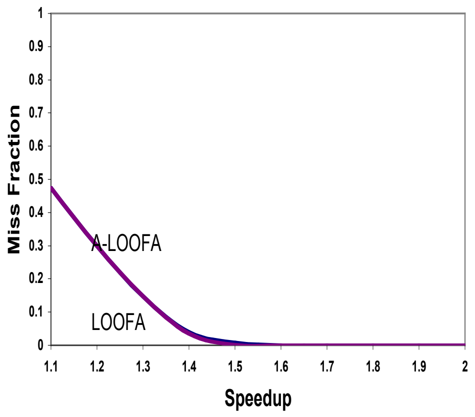
3.8.5 Performance Evaluation

To compare the performance of LOOFA and A-LOOFA we subjected both the algorithms to the stress tests, Test A and Test B. As can be seen from Fig. 3.22, they have almost identical performance under both tests indicating that even partial ordering techniques like the odd-even sorting used in A-LOOFA can perform well due to the slowly changing nature of the output queue lengths.

The problem of overload conditions in IP networks makes it important to study the performance of practical scheduling algorithms under extreme traffic conditions. The stress test that we have presented in this paper, helps us to determine which



(a) Test A



(b) Test B

Figure 3.22: Miss fractions for A-LOOFA and LOOFA under stress test with 5 participating inputs and 12 phases.

algorithms perform best under these conditions. Using the stress test, we have studied the performance of a wide variety of crossbar scheduling algorithms like PIM, *i*SLIP, APSARA and LOOFA under overload conditions and have designed improved and implementable *stress resistant* variants of these algorithms, LLS-R, LLS-S, SOLIF-A and A-LOOFA which can maintain their throughput under both uniform traffic and stress tests.

Chapter 4

Distributed Scheduling

4.1 Introduction

It is common knowledge that increasing traffic demands [5] are making it infeasible to use low capacity routers in IP networks. Even when a number of low capacity, low density routers (with 16-32 ports) are configured to together behave like a single large router, a majority of the ports are used in interconnecting the routers themselves. Hence, most Internet Service Providers (ISPs) prefer large capacity routers for their core networks.

Most scalable, high capacity routers currently under development are multi-rack systems (to reduce power density) and employ distributed, multi-stage switching fabrics [3, 1, 2] . This distributed, multi-stage architecture also enables *incremental scaling* of the switching system, where additional switching capacity can be incrementally added to the system with increasing traffic demands.

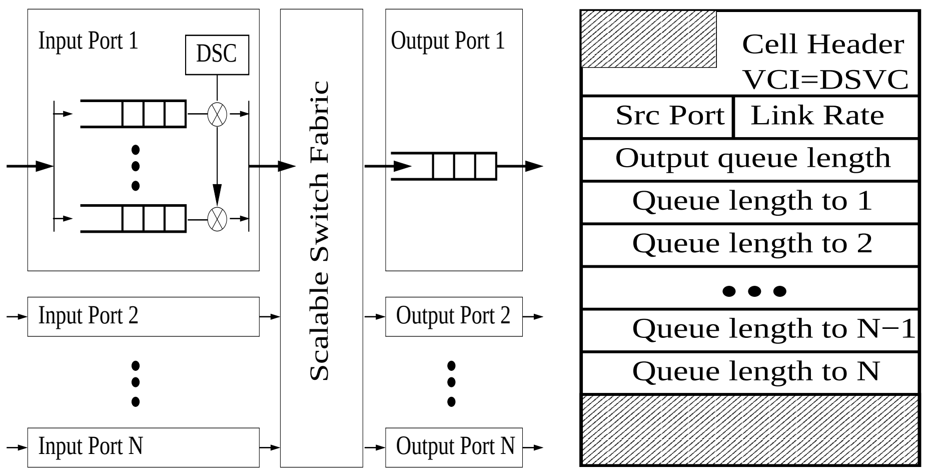
Unfortunately, multi-stage switching fabrics have lacked mechanisms to ensure high throughput when faced with extreme traffic conditions. In the presence of a sustained overload at an output port, such systems can become congested with traffic attempting to reach the overloaded output, interfering with the flow of traffic to other (possibly, non-overloaded) outputs. Thus, the performance of these systems can degrade unpredictably, especially in unregulated IP networks. This is undesirable, since network operators need switching systems that can operate at throughputs of 100% to use the full capacity of expensive long haul links.

In this chapter, we introduce a novel and scalable mechanism called Distributed Scheduling (DS) to provide performance guarantees in such high capacity, buffered, multi-stage switching systems.

4.2 Distributed Scheduling

Distributed Scheduling (DS) is a method for regulating the flow of traffic through large routers employing multi-stage switching fabrics which use buffered switching elements. DS, unlike crossbar scheduling, does not seek to schedule the transmission of individual cells every time slot. Instead, it regulates the *rates* at which traffic is forwarded through the switching fabric from various inputs to outputs using coarse-grained scheduling. This means that the *rates* themselves are determined and readjusted, only at a pre-determined update period (T). While this approach keeps the mechanism scalable, this also implies that DS can only approximate the throughput and delay properties of a pure output queued switch.

4.2.1 Mechanism



(a) Distributed Scheduling (b) Control Cell

Figure 4.1: Router with distributed scheduling.

Fig. 4.1(a) shows a simplified block diagram of a router that implements distributed scheduling. Each output port contains a FIFO queue and each input port contains a set of N virtual output queues. The *VOQs* are rate controlled by a Distributed Scheduling Controller (DSC). The DSCs at various input ports execute the following pseudocode every update period (T).

```
DQ()
{
        Send_VOQ_status();
        Recv_VOQ_status();
        Allocate_rates();
        for(int i=1 to N){
            SetPace(VOQ(i),new_rate);
        }
}
```

The DSCs periodically exchange information about the VOQs and the output queues and use this information to determine new rates at which their VOQs are to be paced. The exchange of information by inputs can be achieved by using control cells. Fig. 4.1(b) shows the conrol cells used for distributed scheduling in the Dynamically Extensible Router (DER) [35] developed at Washington University. The information exchanged in these particular cells is simply the queue length of various VOQs and outputs, though, in general, any information that incurs an acceptably small overhead can be exchanged. The DER, in particular, is small enough to enable complete sharing of VOQ information by multicasting. Even in larger systems, the overhead due to the exchange of this information is only a small fraction of the system bandwidth. For example, in a system with 1000 links, each operating at 10 Gb/s and using an update period of 100 μs, the overhead due to exchange of queue length status is just 5% of the system bandwidth. Also, the exchange of this information can be made more scalable by having an output aggregate the VOQ information from the various inputs and send this smaller information back to the inputs. From the pseudocode, it is clear that the most important component of DS is the rate allocation algorithm.

4.2.2 Constraints

The rate allocation algorithms used in DS are limited by the following constraints

$$\forall i \ \sum_j r_{i,j} \leq S \times R \tag{4.1}$$

$$\forall j \ \sum_i r_{i,j} \leq S \times R \tag{4.2}$$

where $r_{i,j}$ is the rate at which DSC at input i forwards traffic to output j, S is the speedup of the switch and R is the external link rate. Equation 4.1 *(referred to as the output constraint)* restricts the net rate at which traffic is sent to each output to the speed of the switching fabric itself. Equation 4.2 *(referred to as the input constraint)* denotes that the total rate at which traffic is transferred from each input is limited by the speed of the switching fabric itself. A good rate allocation algorithm should satisfy these constraints and emulate the behaviour of an output queued switch as closely as possible.

4.2.3 Distributed Scheduling vs Crossbar Scheduling

Distributed scheduling and crossbar scheduling are similar in their use of VOQs. While crossbar scheduling seeks to schedule the transmission of individual packets, distributed scheduling regulates the total number of packets *pushed* by the input ports into the buffered multi-stage switching fabric during a given scheduling period. This brings about two important differences. Firstly, the distributed nature of the methods used in DS rules out the iterative matching methods that have proved effective in crossbar scheduling, since each iteration would require an exchange of information, causing the overhead of the algorithm to increase in proportion to the number of iterations. On the other hand, the shift to coarse scheduling provides some flexibility that is not present in crossbar scheduling. In crossbar scheduling, it is necessary to match inputs to outputs in a one-to-one fashion during each scheduling cycle. In distributed scheduling, we allocate the interface bandwidth at each input and output and are free to subdivide that bandwidth in whatever proportions that will produce the best result. These differences lead to different specific solutions, although high level ideas can be usefully transferred between the two contexts.

4.3 Work Conserving Scheduling Algorithms

4.3.1 Problem Definition

We note that the DS mechanism is scalable compared to crossbar scheduling because

1. it is distributed.

2. it is coarse-grained; it performs a scheduling decision only at pre-determined update periods (T).

3. Also, as we will show in further sections, the needed hardware complexity and execution time of algorithms at each port is $O(n)$ or at most $O(n \log n)$ with small constants and each port has to send and receive $O(n)$ information per update period.

The primary issue that we wish to investigate in this section is the effect of making a scheduling decision only at fixed time periods on the throughput of the switch.

4.3.2 System Model

To address this problem we adopt a somewhat idealized view of the system operation. (In Section 4.11, we discuss the implications of these assumptions for real systems.)

A crossbar based CIOQ switch can be viewed as operating in three phases. An arrival phase, a transfer phase and a departure phase. In the arrival phase and departure phase a maximum of one cell can arrive at an input or depart from an output respectively and in the transfer phase, up to S (speedup) cells can be transferred from an input or to an output. Ideally, we would like such a system to be *work conserving.* This implies that at the beginning of each departure phase, every output port which has cells queued for it in the system (at various input ports) has at least one cell queued in its output queue. This ensures that an output is always "working" when it has cells queued for it in the system.

We generalize this simple model and performance metric to define a batch CIOQ (T-CIOQ) switch. In a T-CIOQ switch, a scheduling decision is made only every T time units. A T-CIOQ switch can also be viewed as operating in three phases, where, in the arrival phase an input can receive up to T cells and during the departure phase, each output can send up to T cells on its output link. At the beginning of each transfer phase, a scheduling decision is made to determine which cells are transferred from inputs to outputs with a limit of $S \times T$ cells on each input and output. Also, the best performance that we can expect from such a system is for it to be T-work conserving! We say that a T-CIOQ switch is T-work conserving, if at the beginning of every departure phase, every output port which has cells queued for it in the system, has at least T cells queued in its output queue. We note that the normal CIOQ switch is simply the 1-CIOQ switch and the work conserving property in this context is the same as 1-work conserving.

We ask the question, *is there a scheduling algorithm that can keep a T-CIOQ switch, T-work conserving?*

4.3.3 Maximal and Ordered Scheduling Algorithms

In this section, we describe a general scheduling strategy that can be used to obtain T-work conserving scheduling algorithms for speedups of 2 or more. While these algorithms are not practical for real systems, they provide a conceptual foundation for other algorithms that are practical.

The scheduling strategies that we study in this chapter, maintain an ordering of non-empty VOQs at each input. This ordering is changed (if need be) only at the beginning of any of the three phases. The ordering of the VOQs can be extended to an ordering of the cells at an input. Two cells in the same VOQ are ordered according to their position in the VOQ. Cells in different VOQs are ordered according to the order of the VOQs.

The following definitions will be useful in describing the T-work conserving scheduling algorithms.

DEFINITION 1: We say that a cell b **precedes** *a cell c at the same input, if b comes before c in the cell ordering at the input.*

DEFINITION 2: We refer to a cell c as an ij-cell if it is at input i and is destined for output j. For an ij-cell, let $p(c)$ be the number of cells that precede c at input i and $q(c)$ be the number of cells at output j.

DEFINITION 3: Given a method for ordering the cells at each input, a scheduling algorithm is said to be **ordered** *if for any ij-cell c that is not transferred, no cell preceded by c at input i gets transferred unless output j gets $S \times T$ cells.*

DEFINITION 4: A scheduling algorithm is said to be **maximal** *if for any ij-cell c that is not transferred in a transfer phase, either $S \times T$ cells are transferred from input i or $S \times T$ cells are transferred to output j.*

Our scheduling strategy produces schedules that are both maximal and ordered. We can vary the strategy by using different ordering methods. We describe two ordering methods that each lead to T-work conserving scheduling algorithms for speedups of 2 or more. In fact, because there are many different maximal, ordered scheduling algorithms for any specific ordering method, we obtain two *families* of T-work conserving scheduling algorithms.

To prove that these algorithms are T-work conserving, for any ij-cell c, we define the quantity $slack(c) = p(c) - q(c)$. For each of the methods studied, we'll show that at the beginning of each departure phase, $slack(c) \geq T$, if $S \geq 2$. This implies that for any output with fewer than T cells in its outgoing queue, there can be no cells that are destined to this output and are still waiting in any inputside VOQs. This in turn proves that the system is T-work conserving.

4.4 Batch Critical Cells First Algorithm

Table 4.1: Notation used in describing batch critical cells first algorithm.

Notation	Definition
ij-cell c	Cell c at input i destined to output j.
$q(c)$	Queue length of output j to which cell c is destined.
$p(c)$	Number of cells preceding c at its input i.
$slack(c)$	p(c)-q(c).

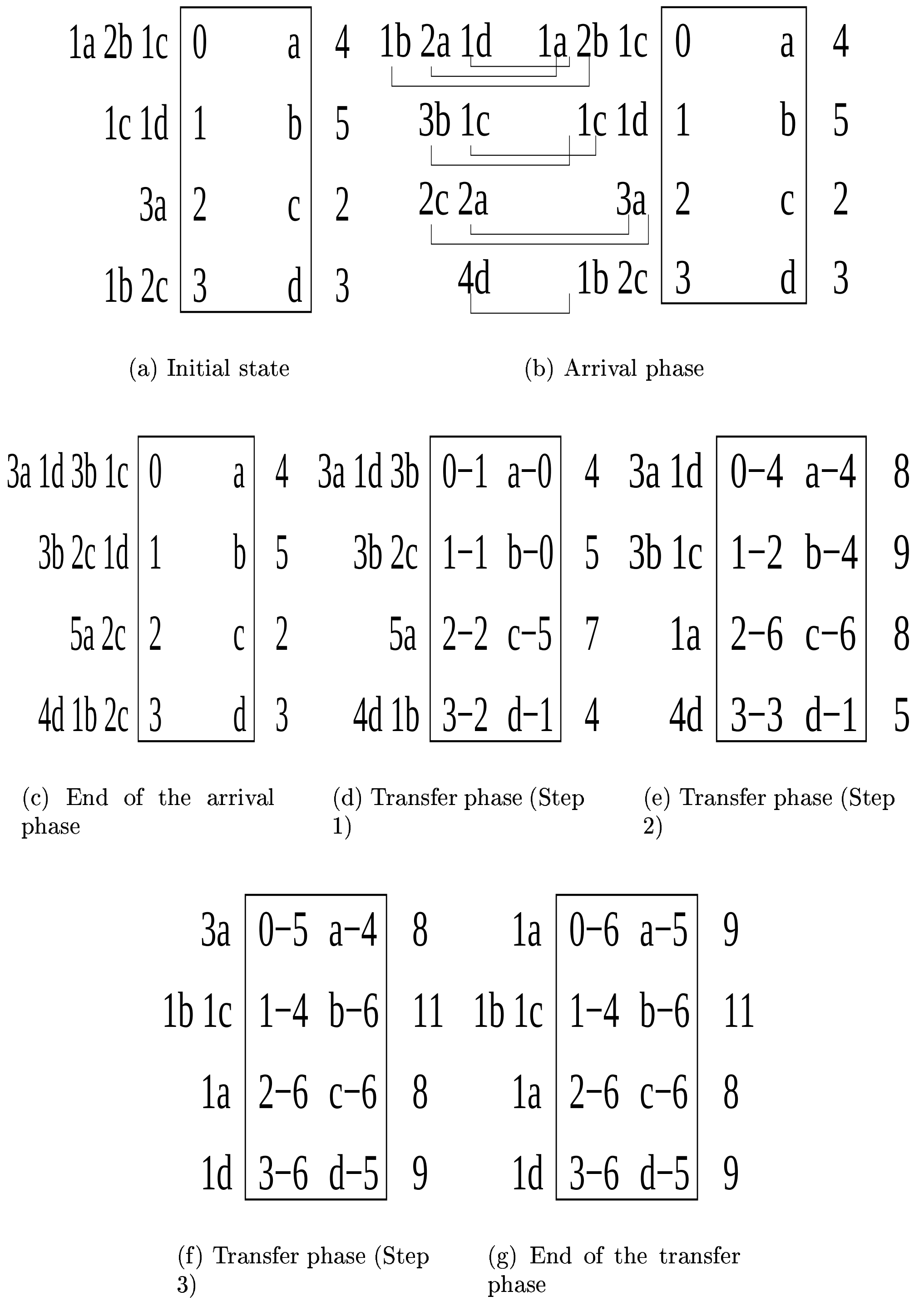
The algorithm that we describe in this section is based on ideas first developed for the *Critical Cells First* algorithm of [13]. Hence, we refer to it as the *Batch Critical Cells First* method.

4.4.1 VOQ Ordering

In the BCCF method, the relative ordering of two VOQs remains the same so long as they remain non-empty, but when a new VOQ becomes non-empty, it must be ordered relative to the others. When a cell c arrives and the VOQ for c's output is empty, we insert the VOQ into the existing ordering based on the magnitude of $q(c)$ (refer Table 4.1 for notation). In particular, if the ordered list of VOQs is $v_1, v_2, \cdots$, we place the VOQ immediately after the queue v_k determined by the largest integer k for which the total number of cells in $v_1, \cdots, v_k$ is no larger than $q(c)$. Notice that this ensures that slack of c is non-negative right after c arrives. A specific scheduling algorithm is an instance of the BCCF method if it produces methods that are maximal and ordered with respect to this VOQ ordering method.

4.4.2 Example

Fig. 4.2 presents an example of the operation of the BCCF algorithm. In the example, the system has 4 inputs and 4 outputs and has a speedup of 1.5. The update period (T) is 4. Hence, the algorithm can transfer $ST = 6$ cells from any input or to any output in the transfer phase.



(a) Initial state

(b) Arrival phase

(c) End of the arrival phase

(d) Transfer phase (Step 1)

(e) Transfer phase (Step 2)

(f) Transfer phase (Step 3)

(g) End of the transfer phase

Figure 4.2: Example operation of the BCCF algorithm.

Each figure indicates the number and ordering of cells queued at an input (inputs are numbered $0, 1, 2$ and 3) to various outputs (a, b, c and d). The figures also indicate the total backlog at the various outputs. For instance, the initial state of the system is shown in Fig. 4.2(a). This figure shows that at input 0, the VOQ to output c (with queue length $= 1$), has the highest priority followed by the VOQs to output b (with queue length $= 2$) and output a (with queue length $= 1$). Also, the figure shows that outputs a, b, c and d have an initial backlog of $4, 5, 2$ and 3, respectively. Note that in Fig. 4.2(a), the slack of all cells at the inputs is greater than 0. Fig. 4.2(b) shows the arrival phase of the system. In this phase, each input receives exactly 4 cells. For instance, input 2 receives two cells each to outputs a and c. When incoming cells are inserted into empty VOQs at an input, the ordering of the VOQs at that input is changed according to the VOQ ordering criterion of BCCF. Hence, as shown in Fig. 4.2(c) (which shows the state of the system after the arrival phase) the VOQ to output d at input 0 is placed after the VOQs to outputs c and b and before the VOQ to output a (because, the queue length of output d is 3 and input 0 has exactly a total of 3 cells queued for output c and b, when the cell destined to d arrives).

Figures 4.2(d), 4.2(e) and 4.2(f) show the progress of the transfer phase of the algorithm. For the sake of illustration, we've broken down the transfer phase into a series of figures, where the algorithm attempts to move only the cells from the highest priority VOQ at each input to the corresponding output. For instance, in step 1 of the transfer phase (Fig. 4.2(d)), the algorithm determines that it can transfer 5 cells to output c from inputs $0, 2$ and 3. Each of these figures also keeps track of the total number of cells transferred from an input or to an output. The notation used in these figures shows that, in Fig. 4.2(d) for instance, input 0 has transferred 1 cell $(0 - 1)$ and also output c has received 5 cells $(c - 5)$. Clearly, a maximum of $ST = 6$ cells can transferred from an input or to an output. The final state of the system, after the transfer phase is shown in Fig. 4.2(g).

4.4.3 Proof

To prove that the BCCF algorithm is T-work conserving, we need to show that for any ij-cell, $slack(c) \geq T$ at the start of the departure phase. We use two lemmas to prove the same.

Lemma 4.4.1 *For a system using the BCCF algorithm, if c is any cell that remains at its input during a transfer phase, then $slack(c)$ increases by at least $S \times T$ during the transfer phase.*

Proof Since the VOQ ordering does not change during a transfer phase (more precisely, VOQs that remain non-empty during the transfer phase have the same relative order), any maximal, ordered scheduling algorithm either causes $q(c)$ to increase by $S \times T$ or causes $p(c)$ to decrease by $S \times T$. In either case, $slack(c)$ increases by $S \times T$. ∎

Note that as long as a cell c remains at an input, each arrival phase and departure phase cause $slack(c)$ to decrease by at most T. So, if $S \geq 2$, $slack(c)$ cannot go down over the course of a complete time step, comprising an arrival phase, transfer phase and a departure phase.

Lemma 4.4.2 *For a system using the BCCF algorithm with $S \geq 2$, if c is any cell at an input just before the start of the departure phase, then $slack(c) \geq T$.*

Proof We show that for any cell c present at the end of an arrival phase, $slack(c) \geq -T$. The result then follows from Lemma 4.4.1 and the fact that $S \geq 2$. The proof is by induction on the time step.

For any cell c that arrives during the first time step, $p(c) \leq T$ at the end of the arrival phase, so $slack(c) \geq -T$ at the end of the arrival phase. Since $s \geq 2$, there can be no net decrease in $slack(c)$ from one time step to the next, so $slack(c)$ remains $\geq -T$ at the end of each subsequent arrival phase, so long as c remains at the input.

If a cell c arrives during step t and its VOQ is empty when it arrives, then the rule used to order the VOQ relative to the others ensures that $slack(c) \geq 0$ right after it arrives. Hence, $slack(c) \geq -T$ at the end of the arrival phase and this remains true at the end of each subsequent arrival phase, so long as c remains at the input.

If a cell c arrives during step t and its VOQ is not empty, but was empty at the start of the arrival phase, then let b be the first arriving cell to be placed in c's VOQ during this arrival phase. Then, $slack(b)$ was at least 0 at the time it arrived and at most $T-1$ cells can have arrived after b did in this arrival phase. If exactly r of these precede b, then at the end of the arrival phase,

$$slack(c) \quad \geq \quad slack(b) - ((T-1) - r) \tag{4.3}$$

$$\geq \quad (-r) - ((T-1) - r) \tag{4.4}$$
$$\geq \quad -T \tag{4.5}$$

If a cell c arrives during step t and its VOQ was not empty at the start of the arrival phase, then let b be the last cell in c's VOQ at the start of the arrival phase. By the induction hypothesis, $slack(b) \geq -T$ at the end of the previous arrival phase. Since the subsequent transfer phase increases $slack(b)$ by at least $2T$ and the departure phase decreases it by at most T, $slack(b) \geq 0$ at the start of the arrival phase in step t. During this arrival phase, at most T new cells arrive at c's input. Let r be the number of these arriving cells that precede b. Then at the end of the arrival phase

$$\begin{aligned} slack(c) &\geq slack(b) - ((T-1) - r) && (4.6)\\ &\geq (-r) - ((T-1) - r) && (4.7)\\ &\geq -T && (4.8) \end{aligned}$$

Hence, $slack(c) \geq -T$ at the end of the arrival phase in all cases and this remains true at the end of each subsequent arrival phase, so long as c remains at the input. ∎

Lemma 4.4.2 leads immediately to a work-conservation theorem for BCCF.

Theorem 4.4.3 *For $S \geq 2$, any scheduler using the BCCF algorithm is T-work conserving.*

4.5 Batch Least Occupied Output First Algorithm

Our second algorithm is based on ideas first developed in the *Least Occupied Output First Algorithm* in [34], so we refer to it as the *Batch Least Occupied Output First Algorithm* (BLOOFA).

4.5.1 VOQ Ordering

In the BLOOFA algorithm, the VOQs are ordered according to the number of cells in the output-side queues. VOQs going to outputs with fewer cells precede VOQs going to outputs with more cells. Outputs with equal number of cells are ordered according to the numbering of the outputs. We define the BLOOFA algorithm to be the combination of this VOQ ordering method with any maximal, ordered scheduling algorithm.

4.5.2 Example

Fig. 4.3 presents an example of the operation of the BLOOFA algorithm. The figure uses the same notation as used in Fig. 4.2 for demonstrating the operation of BCCF. These two figures illustrate the differences in the VOQ ordering policies of the two algorithms. In particular, note that the incoming cells to various inputs in Fig. 4.3(b) are inserted into VOQs ordered according to output backlogs.

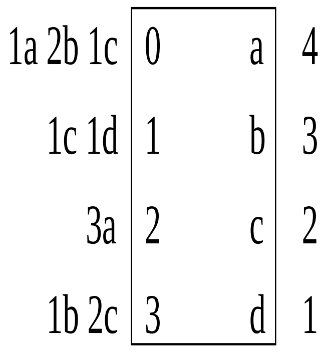
4.5.3 Proof

To prove that the BLOOFA algorithm is T-work conserving, we show that $slack(c) \geq T$ at the start of each departure phase, using the same overall strategy used for BCCF.

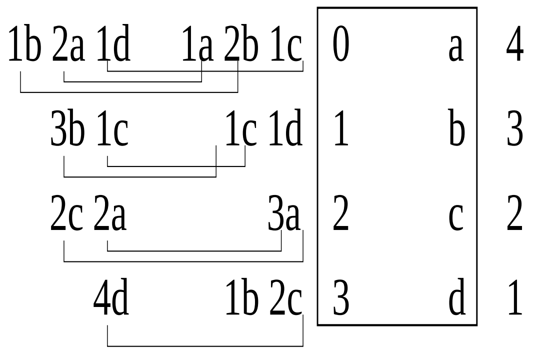
To prove this claim, we use the notation shown in Table 4.2. Let c be any cell at an input and $o(c)$ be the output to which c is destined. For a given phase (arrival, transfer or departure) let $q_0(c)$ be the queue length of $o(c)$ at the *beginning* of the phase and let $q_F(c)$ be the queue length of $o(c)$ at the *end* of the phase. Similarly, let $p_0(c)$ be the number of cells preceding c at the beginning of the phase and $p_F(c)$ be the number of cells preceding c at the end of the phase. Also, define *slack* of cell c at the beginning of the phase as $slack_0(c) = q_0(c) - p_0(c)$ and slack of c at the end of the phase, $slack_F(c)$ as $slack_F(c) = q_F(c) - p_F(c)$. Finally, let $r(c)$ be the number of cells received by $o(c)$ during the transfer phase.

Table 4.2: Notation used in describing the batch least occupied output first algorithm.

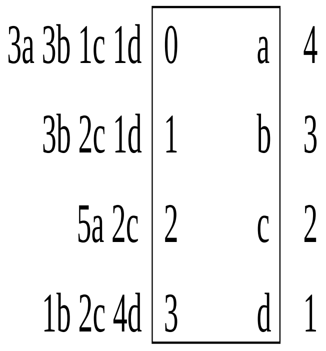
Notation	Definition
$o(c)$	Output to which a cell c is destined.
$q_0(c)$	Queue length of o(c) at the beginning of a phase.
$p_0(c)$	Number of cells preceding c at the beginning of a phase.
$slack_0(c)$	Slack of cell c at the beginning of a phase $= o(c) - p(c)$
$q_F(c)$	Queue length of o(c) at the end of a phase.
$p_F(c)$	Number of cells preceding c at the end of a phase.
$slack_F(c)$	Slack of cell c at the end of a phase $= o'(c) - p'(c)$.
$r(c)$	Number of cells received by o(c) during transfer phase.
T	Maximum number of arrivals at an input in arrival phase.



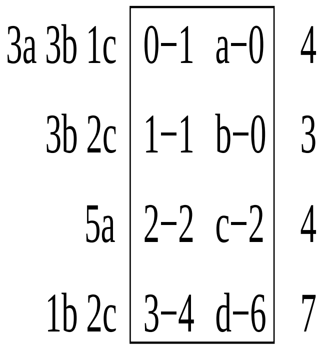
(a) Initial state



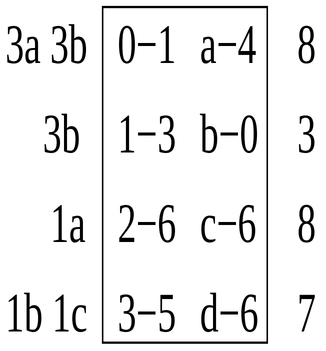
(b) Arrival phase



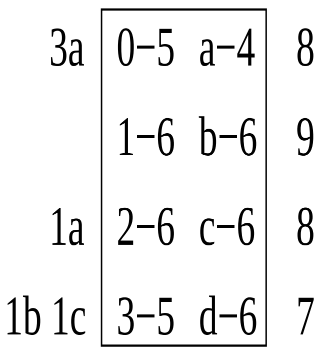
(c) End of the arrival phase



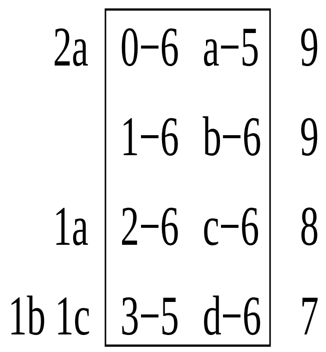
(d) Transfer phase (Step 1)



(e) Transfer phase (Step 2)



(f) Transfer phase (Step 3)



(g) End of the transfer phase

Figure 4.3: Example operation of the BLOOFA algorithm.

Lemma 4.5.1 *If all cells at an input have non-negative slack before an arrival phase then the minimum slack of cells at the input after the arrival phase is $\geq -T$.*

Proof Consider a cell c at an input which didn't arrive in this phase. Clearly, at the beginning of the phase

$$slack_0(c) = q_0(c) - p_0(c) \geq 0 \tag{4.9}$$

The precedence of c can increase by a maximum of T during the arrival phase. Hence, the slack of c after the arrival phase is

$$slack_F(c) = q_F(c) - p_F(c) \geq q_0(c) - (p_0(c) + T) \geq -T \tag{4.10}$$

Now, consider a cell b which arrives in this arrival phase. Let c be the latest cell (w.r.t the cell ordering) among those that didn't arrive in this phase and precede b. (If there is no such cell, clearly, $slack_F(b) \geq -T$.) Since c precedes b after the arrival phase, and at most T cells can arrive in one phase

$$q_F(c) \leq q_F(b) \tag{4.11}$$

Also, since c is the latest among the cells that didn't arrive in this phase and precede b,

$$(p_F(c) - p_0(c)) + (p_F(b) - p_F(c)) \leq T \tag{4.12}$$

since the total number of cells which arrived in this phase and precede b is $\leq T$. Hence,

$$p_F(b) \leq p_0(c) + T \tag{4.13}$$

Using equations 4.9, 4.11 and 4.13

$$\begin{aligned} q_F(b) - p_F(b) &\geq q_0(c) - (p_0(c) + T)) & (4.14)\\ &\geq q_0(c) - p_0(c) - T & (4.15)\\ &\geq -T & (4.16) \end{aligned}$$

∎

Lemma 4.5.2 *For a BLOOFA scheduler with $S \geq 2$, if all cells at an input have slack $\geq -T$ before a transfer phase, then, the slack of all cells at that input is $\geq T$ after transfer phase.*

Proof Let c be a cell at input i that is not transferred in the transfer phase. Also, let the slack of c before the transfer phase be

$$slack_0(c) = -T + x. \tag{4.17}$$

where $x \geq 0$ since, all cells have slack $\geq -T$ before the transfer phase.

Because of the change in output queue lengths during the transfer phase, a set of cells B which didn't precede c before the transfer phase might precede it after the transfer phase. If B is empty, i.e, $B = \{\}$, then either $2T$ cells preceding c at its input were transferred or $2T$ cells were received by the output to which c is destined. This is because of the maximality of the scheduling algorithm. Hence, the slack of cell c after the transfer phase ($slack_F(c)$) is

$$slack_F(c) \geq -T + x + 2T \geq x + T \geq T \tag{4.18}$$

Let B have exactly $c_m > 0$ cells. Let b be a cell such that $b \in B$ and $\forall\, b' \in B$, $p_0(b) \geq p_0(b')$. I.e, b is the cell latest in the list of cells which succeeded c before the transfer phase but now precede it.

Since, b precedes c after the transfer phase (but, succeeded it before the transfer phase)

$$\begin{aligned} q_0(c) &\leq q_0(b) && (4.19)\\ q_F(c) = q_0(c) + r(c) &\geq q_F(b) = q_0(b) + r(b) && (4.20) \end{aligned}$$

And hence,

$$r(b) \leq r(c) \leq 2T \tag{4.21}$$

Equation 4.21 and the maximality of the scheduling algorithm imply that, $2T$ cells preceeding b at input i were transferred during the phase. Let c_p of these $2T$ cells precede c before the transfer phase and c_s of them succeed c, hence, $c_p + c_s = 2T$.

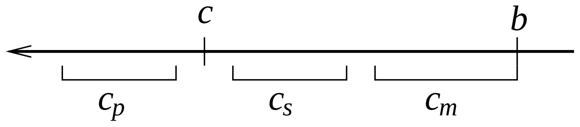


Figure 4.4: Precedence list.

Firstly, as shown in Fig. 4.4,

$$p_0(b) \geq p_0(c) + c_s + c_m \tag{4.22}$$

Also, using equations 4.17 and 4.22 and the fact that $slack(b) \geq -T$, we have

$$\begin{aligned} q_0(c) - p_0(c) &\leq q_0(b) - p_0(b) + x && (4.23) \\ &\leq q_0(b) - (p_0(c) + c_s + c_m) + x && (4.24) \end{aligned}$$

Or simply,

$$q_0(c) \leq q_0(b) + x - c_s - c_m \tag{4.25}$$

Using equation 4.25 in 4.20, we have

$$\begin{aligned} q_0(b) &\leq q_0(c) + r(c) - r(b) && (4.26) \\ &\leq q_0(b) + x - c_s - c_m + r(c) - r(b) && (4.27) \end{aligned}$$

implying that,

$$r(b) - x \leq r(c) - c_s - c_m \tag{4.28}$$

Calculating the new slack of cell c after the transfer phase ($slack_F(c)$), we have

$$\begin{aligned} slack_F(c) &= q_F(c) - p_F(c) && (4.29) \\ &= (q_0(c) + r(c)) - (p_0(c) - c_p + c_m) && (4.30) \\ &= q_0(c) + r(c) - (p_0(c) - (2T - c_s) + c_m) && (4.31) \\ &= slack_0(c) + (2T + r(c) - c_s - c_m) && (4.32) \end{aligned}$$

Using equation 4.28, we have

$$\begin{aligned} slack_F(c) &\geq slack_0(c) + (2T + r(b) - x) && (4.33) \\ &\geq (-T + x) + 2T - x + r(b) && (4.34) \\ &\geq T + r(b) && (4.35) \\ &\geq T && (4.36) \end{aligned}$$

∎

Lemma 4.5.3 *If the slack of all cells is $\geq T$ at the beginning of the departure phase, then, the slack of all cells is ≥ 0 after departure phase.*

Proof The queue length of an output can decrease by a maximum of T during the departure phase. Hence, if no output reordering occurs (which changes the precedence of cells at inputs), the slack of any cell c at an input $slack_F(c) \geq 0$ after the departure phase (since, $slack_0(c) \geq T$ before departure phase).

Now, we prove that there can be no change in the precedence of cells at inputs due to output reordering during the departure phase. Consider two cells b and c at an input before the departure phase. Let,

$$q_0(b) \leq q_0(c) \tag{4.37}$$

Then, after departure phase, one of the following holds true.

$$q_F(b) = q_F(c) = 0 \tag{4.38}$$

$$q_F(b) = 0 \quad \leq \quad q_F(c) = q_0(c) - T \tag{4.39}$$

$$q_F(b) = q_0(b) - T \quad \leq \quad q_F(c) = q_0(c) - T \tag{4.40}$$

Hence, the relative order of output queue lengths does not change during the departure phase.

Theorem 4.5.4 *For any cell c at an input, $slack(c) \geq -T$ after the arrival phase (before the transfer phase), $slack(c) \geq T$ after the transfer phase (before the departure phase) and $slack(c) \geq 0$ after the departure phase (before the arrival phase).*

Proof In the beginning when the system does not have any cells, trivially, the inequalities hold true.

From Lemmas 4.5.1, 4.5.2 and 4.5.3, we have shown that if the slack of all cells is ≥ 0 at the beginning of an arrival phase then all three inequalities hold true and the slack of all cells is ≥ 0 at the next arrival phase too. Hence, by induction, all three inequalities hold true. ∎

From Theorem 4.5.4 it is clear that, no output with cells queued at inputs can have fewer than T cells in its output queue at the beginning of the departure phase. Hence, the BLOOFA scheduling algorithm is T-work conserving with a speedup of 2.

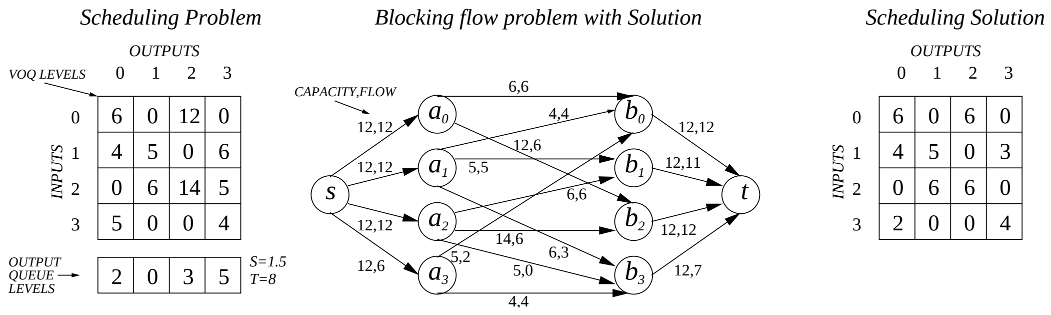


Figure 4.5: Example of a maximal ordered schedule constructed from a blocking flow.

4.6 Implementation of Maximal, Ordered Schedulers

We have shown that the combination of two different ordering strategies with the a maximal, ordered scheduling algorithms ensures work-conserving operation when the speedup is at least 2. We now need to show how to realize a maximal, ordered scheduling algorithm. We start with a centralized algorithm and show how it can be converted into an iterative, distributed algorithm. While the overhead of such iterative algorithms makes them impractical, they provide the basis for non-iterative algorithms that are practical.

The key observation is that the scheduling problem can be reduced to finding a *blocking flow* in an acyclic flow network [62]. A flow network is a directed graph with a distinguished source vertex s, a distinguished sink vertex t and a non-negative *capacity* for each edge. A flow, in such a network, is a non-negative function defined on the edges. The flow on an edge must not exceed its capacity and for every vertex but s and t, the sum of the flow values on the incoming edges must equal the sum of the flow values on the outgoing edges. An edge in the network is called *saturated*, if the flow on the edge is equal to its capacity. A blocking flow is one for which every path from s to t contains at least one saturated edge. (Note that a blocking flow is not necessarily a maximum flow). Also, if both the capacity and flow of all edges are integers, the flow defined on the network is called an *integer flow*.

To convert the scheduling problem to the problem of finding a blocking flow, we first need to construct a flow network. Our network has a source s, a sink t, n vertices referred to as the inputs and another n vertices referred to as the outputs. There is an edge with capacity ST from s to each input. Similarly, there is an edge with capacity ST from each output to t. For each non-empty VOQ at input i of the router with cells for output j, there is an edge in the flow network from input i to output j with capacity equal to the number of cells in the VOQ. (An example of a flow network constructed to solve a particular scheduling problem together with the corresponding solution is shown in Figure. 4.5)

For any integer flow, we can construct a schedule that transfers cells from input i to output j based on the flow on the edge from input i to output j. Note that such a schedule does not violate any of the constraints on the number of cells that can be sent from any input or to any output. Also, note that any blocking flow corresponds to a maximal schedule, since any blocking flow corresponding to a schedule which fails to transfer a cell c from input i to output j cannot saturate the edge from input i to output j, hence it must saturate the edge from s to i or the edge from j to t. Such a flow corresponds to a schedule in which either input i sends ST cells or output j receives ST cells.

Dinic's algorithm [17] for the *maximum flow problem* constructs blocking flows in acyclic flow networks as one step in its overall execution. There are several different variants of Dinic's algorithm, that use different methods of constructing blocking flows. The most straightforward method is to repeatedly search for st-paths with no saturated egde and add as much flow as possible along such paths. **We can obtain a maximal, ordered scheduler by modifying Dinic's algorithm so that it preferrentially selects edges between input and output vertices, according to the VOQ ordering at the input.** The blocking flow shown in Fig. 4.5 was constructed in this way, based on the BLOOFA ordering.

If paths are found using depth-first search and edges leading to dead-ends are removed as they are discovered, Dinic's algortithm finds a blocking flow in $O(mn)$ time where m is the number of edges and n is the number of vertices in the flow graph. Because the flow graphs corresponding to schedules have bounded depth and because the number of inputs, outputs and edges are all bounded by the number of non-empty VOQs, the algorithm finds a blocking flow in these graphs in $O(v)$ time, where v is the number of non-empty VOQs. This yields an optimal centralized scheduling algorithm. However, since v can be as large as n^2 (where n is the number of inputs of the interconnection network), this is not altogther practical.

4.7 Distributed, Iterative Schedulers

We can obtain a distributed, iterative scheduling algorithm based on the ideas presented in the previous section. Rather than state this in the language of blocking flows, we describe it directly as a scheduling algorithm. In distributed scheduling, we first have an exchange of messages in which each output announces the number of cells in its outgoing queue. The inputs use this information to maintain their VOQ order. Note that this requires that each output send n messages and each input receive n messages. Next, the inputs and outputs proceed through a series of rounds.

In each round, the inputs that have uncommitted cells to send and have not yet committed to send ST cells, send *bid* messages to those outputs that are still prepared to accept more cells. The inputs construct their bids in accordance with VOQ ordering. In particular, an input commits all the cells it has for the first output in the ordering and makes similar maximal bids for subsequent outputs until it has placed as many bids as it can. Inputs may not *overbid* as they are obliged to send cells to any output that accepts a bid. Note that at most one of the bid messages an input sends during a round does not commit all the cells that it has for the target output.

During each round, outputs receive bids from inputs and accept as many as possible. If an output does not receive bids for at least ST cells, it does nothing during this round. In particular, it sends no message back to the inputs. Such a "response" is treated as an implicit accept and is taken into account in subsequent bids. Once an output has received bids for a total of ST cells, it sends an accept message to the all the inputs (not just those that sent it bids). The accept message contains a pair of values (i, x) and it means that the output accepts all bids received from inputs whose index is less than i, rejects all bids from inputs whose index is greater then i and accepts exactly x cells from input i. Once an output sends an accept message, its role in the scheduling is complete.

This procedure has some attractive properties. First note that each output sends n messages in the bidding process, so each input receives only n messages from the outputs. Also, an input sends at most two bids to any particular output, so an input sends at most $2n$ bids and an output receives at most $2n$ bids. Thus, the number of control cells that must be handled at any input or output during the scheduling is $O(n)$. Unfortunately, this does not imply that the algorithm runs in $O(n)$ time, since it can require upto n rounds and some outputs may have to handle close to n messages during most rounds.

It's possible to reduce the time for each round by having the switch elements that make up the interconnection network participate in the handling of bids and responses. However, in the next section we turn our attention instead, to algorithms that are simpler to implement and which, while not provably work-conserving, are able to match the performance of work-conserving algorithms even under extreme traffic conditions.

4.8 Distributed BLOOFA (DBL)

The work-conserving algorithms discussed in the previous sections can be implemented using iterative algorithms that require a potentially large number of message exchanges. In this section, we formulate a distributed algorithm that approximates the behaviour of the BLOOFA algorithm while requiring just one exchange of messages. Our *Distributed BLOOFA* (DBL) algorithm avoids the need for many message exchanges by having the inputs structure their bids to avoid the situation where some outputs are swamped with more bids than they can accept, while others are left with no bids. Specifically, we use a technique called *backlog-proportional allocation* to limit the number of bids that are made for any output by the inputs.

DBL starts with each input i sending a message to each output j, telling it how many cells $B(i,j)$ it has in its VOQ for output j. Each output j then sends a message to all inputs containing the number of cells in its output queue ($B(j)$) and the total number of cells that inputs have to send it, ($\sum_i B(i,j)$). Note that each input and output sends and receives n messages. Once this exchange of messages has been made, each input independently decides how many cells to send to each output. To prevent too many cells from being sent to any output, input i is allowed to send at most $hi(i,j)$ cells to output j, where

$$hi(i,j) = ST\frac{B(i,j)}{\sum_i B(i,j)}$$

Each input then orders the outputs according to the length of their output queues and goes through this list, assigning as many cells as it is permitted for each output, before going to the next output in the list. The scheduling is complete when the input has assigned ST cells or has assigned all the cells permitted by the bound.

4.8.1 Example

	a–4	b–3	c–2	d–1
0	3	3	1	1
1	0	3	2	1
2	5	0	2	0
3	0	1	2	4

(a) $B(i,j)$ at the end of the arrival phase

	a–4	b–3	c–2	d–1
0	18/8	18/7	6/7	1
1	0	18/7	12/7	1
2	30/8	0	12/7	0
3	0	6/7	12/7	4

(b) $hi(i,j)$

	a–4	b–3	c–2	d–1
0	2	3	1	1
1	0	2	2	1
2	4	0	1	0
3	0	1	2	4

(c) Integer values of $hi(i,j)$

	a–4	b–3	c–2	d–1
0	1	3	1	1
1	0	2	2	1
2	4	0	1	0
3	0	0	2	4

(d) Number of cells transferred in transfer phase

	a–9	b–8	c–8	d–7
0	2	0	0	0
1	0	1	0	0
2	1	0	1	0
3	0	1	0	0

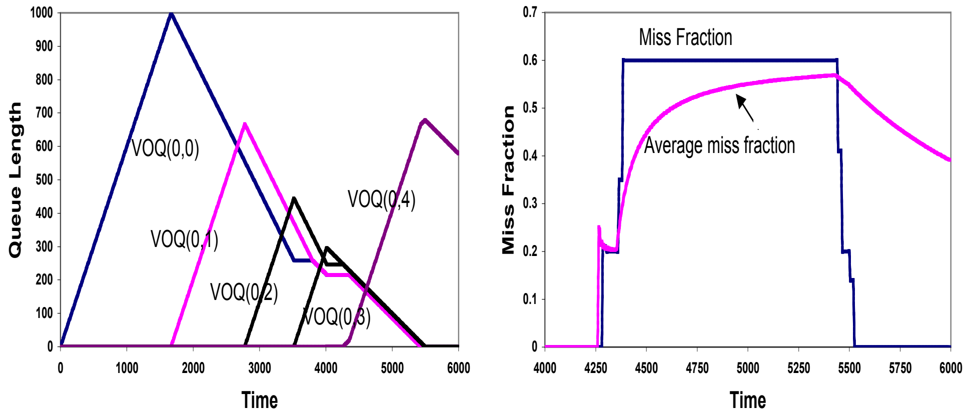
(e) $B(i,j)$ at the end of transfer phase

Figure 4.6: Example operation of the DBL algorithm.

Fig. 4.6 presents an example of the operation of DBL. The initial state of the system (after the arrival phase) is the same as that of the system in Fig. 4.3 which illustrates the operation of BLOOFA. Fig. 4.6(a) shows the matrix of VOQ backlogs at various inputs ($B(i,j)$). Each of these figures also shows the output backlogs. For instance, outputs a, b, c and d have backlogs of $4, 3, 2$ and 1 respectively in Fig. 4.6(a). Fig. 4.6(b) shows the $hi(i,j)$ values calculated by the DBL algorithm. Fig. 4.6(c) shows the integral values of $hi(i,j)$ shown in Fig. 4.6(b), where, for simplicity, we assume that any fractional part > 0.5 (in $hi(i,j)$) is rounded to 1 with a probability of 0.5. Hence, output c, has $hi(i,j)$ values of $1, 2, 1$ and 2 for inputs $0, 1, 2$ and 3 respectively. Fig. 4.6(e) which shows the state of the system after the transfer phase, shows that the number of cells transferred to various outputs in this example is the same when using the BLOOFA algorithm (as shown in Fig. 4.3) demostrating that DBL can effectively approximate the performance of BLOOFA.

4.8.2 Performance Analysis (Stress Test)

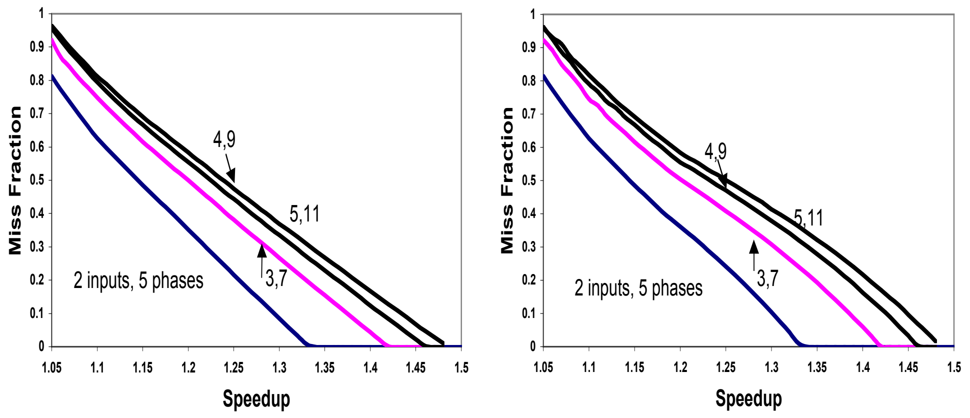
We studied the performance of DBL using simulation for speedups between 1 and 2. The first simulated traffic pattern is the *stress test* described in section 3.2.4. Recall that a stress test consists of a series of phases where various outputs of a switch are overloaded with the objective of creating extreme traffic conditions that can potentially lead to underflow.



(a) Virtual Output Queue Lengths (b) Miss fraction

Figure 4.7: Example stress test (3 inputs, 5 phases, speedup=1.2) on BLOOFA.

Fig. 4.7 shows the progress of a sample stress test on BLOOFA. This stress test has 3 participating inputs and 4 phases and has been simulated at a switch speedup of 1.2. Fig. 4.7(a) shows the buffer levels of the virtual output queues (from those of output 0 to 4) at input 0 (by symmetry, the VOQ lengths at other inputs will be approximately the same as those at input 0). The time unit is the update interval T. The unit of storage is the number of cells that can sent on an external link during the update interval. Notice that during the last stage of the stress test, $B(0,4)$ rises, indicating that though input 0 is the sole sender of cells to output 4, it is unable to transfer them to output 4 as quickly as they come in. This results in loss of link capacity at output 4. Fig. 4.7(b) shows the *miss fraction* (defined in section 3.2.2) for the same stress test. The curve labelled *average miss fraction*



(a) BLOOFA (b) Distributed BLOOFA

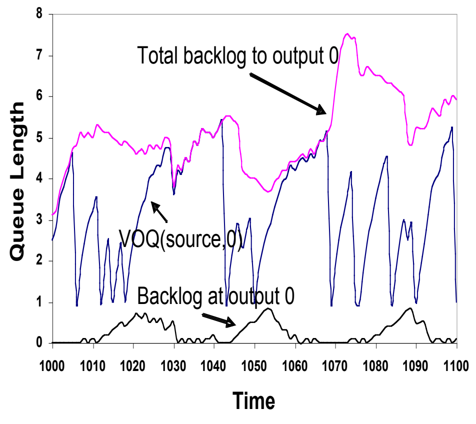
Figure 4.8: Miss fractions for DBL and BLOOFA on a variety of stress tests (with varying inputs and phases).

represents the link capacity lost from that start of the last phase to the time plotted. The curve labelled *miss fraction* measures the average miss fraction during successive measurement intervals (the measurement intervals are 25 time units long). We observe that almost 30% of the link's capacity is effectively lost between the start of the last phase and the period shown.

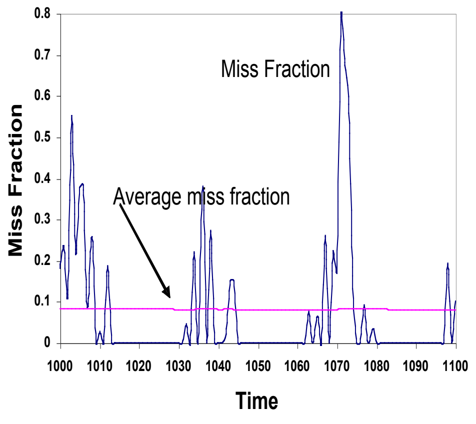
Fig. 4.8(b) shows how DBL performs on a series of stress tests with speedups varying between 1 and 1.5. In these tests, the length of the stress test was set to 1.2 times the length of time that would be required to forward all the cells received during the first phase in an ideal output queued switch. We see here that the average miss fraction (for the output targeted by input 0 in the last phase) drops steadily with increasing speedup, dropping to zero before the speedup reaches 1.5. We performed 90 sets of stress tests, using different number of participating inputs and phases (up to 15 inputs and 15 phases). The results plotted in the figure are the worst cases for 2,3,4 and 5 inputs. In all cases, the average miss fraction for the last phase target output dropped to zero for speedups greater than 1.5.

To compare DBL to BLOOFA, we performed the same series of 90 stress tests on BLOOFA. For speedups below 2, the method used to select which inputs send traffic to a given output can have a significant effect on the performance of BLOOFA. For the results given here, we went through the outptus in order (from smallest output-side backlog to largest) and for each output j, we assigned traffic from the different inputs to output j in proportion to the fraction that each could supply of the total that all inputs could send to j in this update interval. Fig. 4.8(a) shows the results of these stress tests on BLOOFA. From these results, it is clear that the approximation introduced by using the backlog-proportional allocation method to enable efficient distributed scheduling, has a negligible effect on the quality of the scheduling results, even though the distributed version is not known to be provably work-conserving for any speedup.

4.8.3 Performace Analysis (Bursty Traffic)



(a) Queue Lengths



(b) Miss fraction

Figure 4.9: Results from a sample simulation of DBL under bursty traffic (speedup=1.1, load=0.9, mean dwell time = 10).

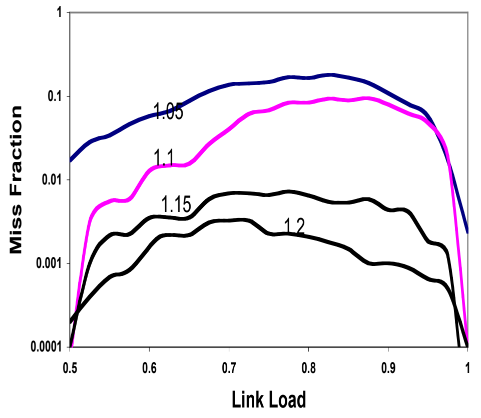


Figure 4.10: Performance of DBL under bursty traffic patterns with varying speedups and subject, target dwell times.

We have also studied the performance of DBL for less adversarial (although, still very demanding) traffic conditions. In particular, we have studied bursty traffic situations in which there is one output (referred to as the *subject* output), for which traffic is arriving continuously at a specified fraction of the link rate. The input at which the subject's traffic arrives changes randomly as the simulation progresses (it remains with a given input for an exponentially distributed time interval). Each of the inputs that is not currently providing traffic for the subject has its own *target* output (not equal to the subject) to which it sends traffic, changing targets randomly and independently of all other inputs (an input retains its current input for an exponentially distributed time interval). With this traffic pattern, roughly one fourth of the outputs that are not the subject are overloaded at any one time (they are targets of two or more inputs). An ideal scheduler will forward cells to the subject output as fast as they come in, preventing any input-side queuing of cells for the subject output. However, the other outputs can build up significant input side backlogs (due to the transient overloads they experience), leading to contention that can affect the subject output. Fig. 4.9 shows an example of what can happen in a system subjected to this type of traffic. Fig. 4.9(a) shows the amount of data buffered for the subject output (which is output 0) at all inputs, the amount of data buffered at the input, which is currently receiving traffic for the subject (VOQ($source, 0$)) and the amount of data buffered at the subject. The unit of storage is the amount of data received on an external link during an update interval and the time unit is the update interval. Fig. 4.9(b) plots the instantaneous and average miss fractions for the same sample test.

Fig. 4.10 shows the average miss fraction from a large number of bursty traffic simulations with varying load and speedups. It's interesting to note that the miss fraction reaches its peak when the input load is between 0.8 and 0.9. Larger input loads lead to a sharp drop in miss fraction. The explanation for this behaviour is that when the input load approaches 1, the output-side backlogs tend to persist for a long period of time and it's only when the output side backlogs are close to zero that misses can occur. As one would expect, the miss fraction drops quickly as the speedup increases. Note that for speedup 1.15 the miss fraction never exceeds 2% meaning that only a fraction of the link capacity is lost. We also note that the bursty traffic model used in these studies represents a very extreme situation. A more realistic bursty trafic model would have a large number of bursty sources (at least a few tens) with more limited peak rates sharing each input link (at least a few tens of sources per link). Such a model is significantly less bursty than the one used here.

4.8.4 Contention Factor

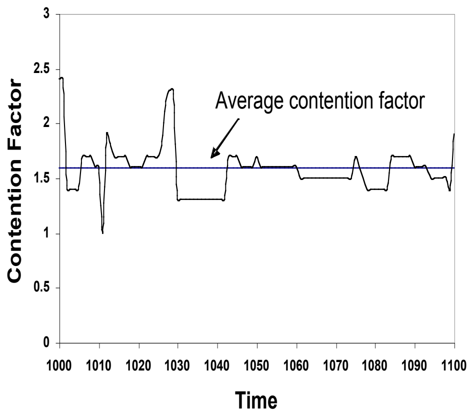


Figure 4.11: Contention factor for the sample bursty traffic test on DBL shown in Fig. 4.9.

It is interesting to note that DBL algorithm performs significantly better on the bursty traffic model (Fig. 4.10) as compared to the stress test (Fig. 4.8(b)). This is an indication of the extreme conditions created by the stress test.

To compare the *severity* of these tests, for a given input i that needs to send traffic to output j, define *contention factor* (η) as

$$\eta = \sum_x \frac{B(i,x)}{\sum_y B(y,x)}$$

Thus, the contention factor denotes the sum of the fractions of traffic to various outputs that a given input holds. If this factor is large, then the input can be expected to incur greater misses when forced to be the sole sender of traffic to a distinct output (j) as is done both in the stress test and the bursty traffic model.

For a stress test with k_1 participating inputs and k_2 phases, in the first $k_2 - 1$ phases, the k_1 inputs build an equal backlog (by symmetry) to each of $k_2 - 1$ outputs and in the final phase (k_2), each participating input sends all its traffic to a new distinct output. Hence,

$$\eta = 1 + \frac{k_2 - 1}{k_1}$$

For a stress test with 3 inputs and 7 phases, $\eta = 3$. We computed η for the sample test of DBL on the bursty traffic model used in Fig. 4.9. As shown in Fig. 4.11, η is relatively smaller in the bursty traffic model and averages just 1.59. This corroborates our contention that the stress test is a very demanding traffic pattern.

We note that though, the contention factor helps to explain the difference in the performance of DBL on the stress test and the bursty traffic model, it does not imply, for example, that increasing the number of phases to number of participating inputs ratio will result in a more demanding stress test. This is because the contention factor doesn't include the output backlogs in its definition which is vital for estimating the miss fraction.

4.9 The Output Leveling Algorithm (OLA)

The intuition behind the BLOOFA algorithm is that by favouring outputs with smaller queues, we can delay the possibility of underflow and with sufficient speedup, potentially avoid that possibility altogether. Theorem 4.5.4 tells us that for a speedup of 2 or more, we can avoid underflow, but it does not say anything about what happens with smaller speedups. When there are several output queues of nearly the same length, BLOOFA transfers as many cells as possible to the shortest queues, possibly preventing any cells from reaching outputs with slightly longer queues. It seems likely that we could get better performance by balancing the transfers so that the resulting output queues are as close to equal as possible. This is the intuition behind the *Output Leveling Algorithm (OLA)*, which we consider next. In this section, we show that OLA, like BCCF and BLOOFA, is work conserving for speedups of 2 or more. Subsequently, we study the performance of OLA and a practical variant of OLA and show that these algorithms can out-perform BLOOFA and DBL.

OLA orders cells at an input in the same way that BLOOFA does. As shown in Table. /refola-notation, let $B_a(i,j)$ and $B_a(j)$ be the lenghts of the VOQs and output queues respectively, immediately before a transfer phase (after an arrival phase) and let $x(i,j)$ be the number of cells transferred from input i to output j during the transfer phase. We say that the transfer is *level* if for any pair of outputs j_1 and j_2,

$$B_a(j_1) + \sum_i x(i,j_1) < B_a(j_2) + \sum_i x(i,j_2) - 1 \tag{4.41}$$

Table 4.3: Notation used in describing the output leveling algorithm.

Notation	Definition
$B_a(i,j)$	Length of $voq(i,j)$ before a transfer phase (after an arrival phase).
$B_a(j)$	Length of output queue j before a transfer phase (after an arrival phase).
$x(i,j)$	Number of cells transferred from input i to output j in the transfer phase.
$x_k(i,j)$	Number of cells transferred from input i to output j in the first k sub-phases.
$q_k(c)$	$B_a(j) + \sum_i x_k(i,j)$, where j is the output to which c is destined
$p_k(c)$	Number of cells preceding c at the end of a sub-phase k.
$slack_a(c)$	Slack of cell c at the beginning of a transfer phase (after an arrival phase).
$slack_k(c)$	Slack of cell c after sub-phase k, $q_k(c) - p_k(c)$.
T	Maximum number of arrivals at an input in arrival phase.

implies that

$$\sum_i x(i,j_1) = min(ST, \sum_i B_a(i,j_1)) \tag{4.42}$$

We now define OLA as a scheduling algorithm that produces schedules that are maximal and level.

4.9.1 Work Conservation

We will essentially use the same strategy as before to show that OLA is work conserving when the speedup is 2. However, to show that the minimum slack increases by ST at each input during a transfer phase, we first need to show how a transfer phase scheduled by OLA can be decomposed into a sequence of *sub-phases*. Each of the sub-phases corresponds to the transfer of one cell from each input and up to one cell to each output. We let $x_k(i,j)$ denote the number of cells trasferred from input i to output j by the first k sub-phases. At the end of the sub-phase k, the outputs are ordered in increasing order of $B_a(j) + \sum_i x_k(i,j)$ with ties being broken according to the output numbers. The ordering of the outputs is used to order the VOQs at each input and this ordering is extended to all the cells at each input. We say that a cell b precedes a cell c before sub-phase k if b comes before c in this cell ordering. We define $q_k(c) = B_a(j) + \sum_i x_k(i,j)$ and we define $p_k(c)$ to be the number of cells at c's input that precede it in the ordering at the end of sub-phase k. We also define $slack_k(c) = q_k(c) - p_k(c)$. Let $slack_a(c)$ be the value of $slack(c)$ before the transfer phase begins (after arrival phase) and note that if k is the last sub-phase, then $slack_k(c)$ is equal to the value of $slack(c)$ following the transfer phase.

Given a schedule constructed by an OLA scheduler, we construct sub-phases iteratively. To construct sub-phase k, we repeat the following steps until there are no outputs that are eligible to be selected.

> *Select an output j that has not yet been selected in this sub-phase for which $\sum_i x_{k-1}(i,j) < \sum_i x(i,j)$ and which, among all such outputs, has the minimum value of $q_{k-1}(c)$. If there are multiple outputs that satisfy this condition, select the output that comes first in the fixed numbering of outputs. Then, select some input i that has not yet been selected in this sub-phase for which $x_{k-1}(i,j) < x(i,j)$. If there is such an input, include the transfer of a cell from input i to output j in sub-phase k, making $x_k(i,j) = x_{k-1}(i,j) + 1$.*

We will use this decomposition to show that the minimum slack at each input increases by at least ST during each transfer phase.

Lemma 4.9.1 *For a system using the OLA method, during a transfer phase, the minimum slack at any input that does not transfer all of its cells during the transfer phase, increases by at least ST.*

Proof Because OLA constructs maximal schedules, any transfer phase that leaves cells at input i must either transfer ST cells from input i or must transfer ST cells to every output j for which a cell remains at input i following the transfer phase. This means that if we decompose a transfer phase into sub-phases, as described above, there will be at least ST sub-phases. We show that every one of these sub-phases increases the minimum slack at input i. Hence, the minimum slack increases by ST over the complete transfer phase.

Let k be the index of any sub-phase and let c be any cell at input i which is not transferred during sub-phase k and for which $slack_{k-1}(c)$ is minimum among all cells at input i. Let j be the output that c is going to. If output j receives no cell during sub-phase k, then input i must tranfer a cell during sub-phase k. The selection rule used to construct sub-phases ensures that the transferred cell precedes c. Hence, $p_k(c) = p_{k-1}(c) - 1$ and thus, $slack_k(c) = slack_{k-1} + 1$.

If output j does receive a cell, then $q_k(c) = q_{k-1}(c) + 1$. If no cell at input i passes c during the sub-phase, then $slack_k(c) \geq slack_{k-1}(c) + 1$. Suppose then, that there is one or more cell that passes c during the sub-phase and let d be such a cell. Since c precedes d before the sub-phase, $q_{k-1}(c) \leq q_{k-1}(d)$ and $p_{k-1}(c) < p_{k-1}(d)$. Since d precedes c after the sub-phase, no cell is received by d's output during the sub-phase and so $q_{k-1}(d) \leq q_{k-1}(c) + 1$. Because, $slack_{k-1}(c) \leq slack_{k-1}(d)$, $p_{k-1}(d) - p_{k-1}(c) \leq q_{k-1}(d) - q_{k-1}(c) \leq 1$ which means that there are no cells that fall between c and d in the cell ordering. This implies that d is the only cell that passes c during the sub-phase. Because d's output receives no cells during the sub-phase, there must be some cell that precedes d that is transferred from input i during the sub-phase and this cell must precede c. Thus, $p_k(c) = p_{k-1}(c)$ and so $slack_k(c) = slack_{k-1}(c)+1$. ∎

As before, we note that each arrival phase causes $slack(c)$ to decrease by at most T. Also, as before, if $slack(c)$ is at least T before the start of the departure phase, then $slack(c)$ is at least zero after the departure phase. This is sufficient to establish that OLA is work-conserving when $s \geq 2$.

Lemma 4.9.2 *For a system using the OLA method with $S \geq 2$, if c is any cell at an input just before the start of the departure phase, then $slack(c) \geq T$.*

The proof of Lemma 4.9.2 is similar to the proofs used in proving that BLOOFA is work conserving (Theorem 4.5.4) except that it uses Lemma 4.9.1 in place of Lemma 4.5.2. Lemma 4.9.2 immediately leads to the work-conservation theorem for OLA.

Theorem 4.9.3 *For $S \geq 2$, any scheduling algorithm using the OLA method is T-work conserving.*

4.9.2 Implementing OLA

An OLA scheduler can be implemented exactly either using linear programming or by solving a minimum cost, maximum flow problem with a convex cost function. We outline the latter approach, as it serves to motivate more practical, approximate variants.

In the classical version of the minimum cost, maximum flow problem [62], each edge has an associated cost coefficient, which is multiplied by the flow on the edge

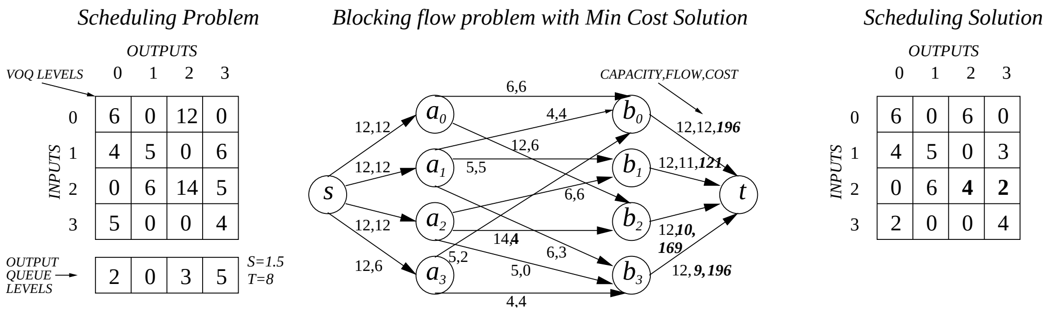


Figure 4.12: Implementing OLA using minimum-cost blocking flow with convex cost function. Differences from earlier solution highlighted in **bold**.

to get the edge's contribution to the overall cost of the flow. There are several well-known efficient algorithms for solving the minimum cost, maximum flow problem. Interestingly, these algorithms can be generalized to handle networks in which the cost is a convex function of the flow on the edge, rather a linear function (x^2 is an example of a convex function).

The OLA scheduling algorithm is reduced to solving a minimum cost, maximum flow problem with a convex edge cost function. An example of such a reduction is shown in the Fig. 4.12, along with a solution and the corresponding schedule. The flow graph is constructed in the same way as was discussed in section 4.6. The only difference is the introduction of non-zero costs on the edges from the output vertices to the sink vertex t. The cost of an edge from output j to t carrying a flow of magnitude x is defined as $C(x) = (x + B(j))^2$. A minimum cost, maximum flow for this network corresponds directly to an OLA schedule. The convexity of the cost function ensures that the flows on different output to sink edges result in costs that are as nearly equal as the various edge capacities allow (if a flow can be shifted from a higher cost edge to lower cost edge, there is a net reduction in the cost, because the lower cost edge has lower incremental cost per unit flow). The use of the offset $B(j)$ in the edge cost means that the costs of the flows on two output-to-sink edges are equal whenever the corresponding schedules yield equal levels at the output queue. Reference [4] describes an algorithm that finds a minimum cost, maximum flow in $O((m \log K)(m + n \log n))$ time on an arbitrary network with n vertices, m edges and maximum edge capacity K. While this algorithm is not useful for distributed scheduling in real systems, it can be used in performance studies to establish a benchmark on more practical algorithms that seek to approximate the behavior of OLA.

4.10 Δ-OLA

In this section, we first describe an approximate centralized version of OLA. We then show how this can be converted to a distributed scheduler, using an extension of the backlog-proportional allocation method introduced earlier.

Our approximate centralized output leveling algorithm (called Δ-OLA), uses an array $x(i,j)$ which is initialized to zero and which defines the number of cells to be transferred from input i to output j, when the scheduling algorithm completes. It also uses a parameter $\Delta \leq ST$, which determines the accuracy of the approximation. During its execution, the algorithm maintains a list of the outputs, sorted in increasing order of $\sum_i x(i,j) + B(j)$. The algorithm repeats the following step so long as there are at least two outputs on the list.

> *Let j_1 and j_2 be the indices of the first two outputs on the list. Increase $\sum_i x(i,j_1)$, by repeatedly increasing $x(i,j)$ for selected values of i (input selection criteria are discussed below). Stop when $\sum_i x(i,j_1) + B(j_1) = \sum_i x(i,j_2) + B(j_2) + \Delta$, or when $\sum_i x(i,j_1) = ST$ or when $\sum_i x(i,j_i) = \sum_i B(i,j_1)$, whichever occurs first. If either of the last two conditions ocurs, remove j_1 from the list. Otherwise, move it down the list so as to maintain the ordering criterion.*

When the list has been reduced to a single output j, the algorithm increase $\sum_i x(i,j)$ until $\sum_i x(i,j) = min(ST, \sum_i B(i,j))$ or until all inputs with cells for output j have scheduled all they can (ST).

The number of steps performed by the algorithm is at most $\frac{nST}{\Delta}$. It can be implemented to run in $O(m + \frac{ST}{\Delta}n^2)$ time, where m is the number of non-empty VOQs. This can be improved to $O(m + \frac{ST}{\Delta}n \log n)$, if the list is replaced by a *heap*. If $\Delta = 1$, the algorithm computes an OLA schedule (regardless of the input selection criterion). For larger values of Δ, it implements a Δ-OLA schedule, which is defined as any maximal schedule for which

$$B(j_1) + \sum_i x(i,j_1) < B(j_2) + \sum_i x(i,j_2) - \Delta \tag{4.43}$$

implies that $\sum_i x(i,j_1) = min(ST, \sum_i B(i,j_1))$. That is, a Δ-OLA schedule allows the output queue differences at the end of a transfer phase to exceed Δ, only if there is no way to transfer more cells to the outputs with smaller queues. Δ-OLA schedulers, like OLA schedulers are work-conserving when the speedup is at least 2 (a slight variation of the proof used for OLA can be used to show this). For smaller speedups, we can tradeoff scheduling performance against running time by adjusting Δ.

The criterion used to select the next input to use to effect an increase in $\sum_i x(i, j_1)$ does not effect the work-conservation condition. However, different choices can effect the performance when the speedup is less than 2. In the performance results reported in this chapter, we distribute the load approximately evenly among all inputs with traffic for output j_1 using a round-robin technique. We maintain a list of inputs that can still send to j_1 (i.e, they have both cells for j_1 and uncommitted bandwidth) and use the first input on the list to increase the flow to j_1. To obtain an even distribution, we take at most Δ from an input at a time and then move that input to the end of the list. This method can be implemented without increasing the time complexity of the algorithm.

4.10.1 Example

Fig. 4.13 presents an example of the operation of Δ-OLA. The initial state of the system (after the arrival phase, as shown in Fig. 4.13(a)) and the notation used is the same as in Fig. 4.6 which demonstrated the performance of Distributed BLOOFA. Fig. 4.13(b), Fig. 4.13(c) and Fig. 4.13(d) show the various stages of execution of the distributed OLA algorithm (with $\Delta = 1$). As shown in Fig. 4.13(e), which shows the state of the system after the transfer phase, Δ-OLA transfers more cells to outputs compared to DBL.

4.10.2 Distributed OLA

To convert a Δ-OLA scheduler to a practical distributed scheduler (Distributed OLA (DOLA)), we use the *backlog proportional allocation* technique introduced earlier to allow inputs to divide the responsibility for supplying traffic to the different outputs. This allows each input to operate independently of others, once the initial exchange of information takes place. As with DBL, this initial exchange supplies input i with the values of $B(j)$ and $\sum_i B(i, j)$ for every output j. Input i also has the values $B(i, j)$ for all j and it uses these to compute values $\rho(i, j) = \frac{B(i,j)}{\sum_i B(i,j)}$. Given this information, input i makes its scheduling decisions in a way that is similar to the centralized algorithm. In particular, input i maintains a list of outputs for which it has cells, sorted in increasing order of $B(j) + \frac{x(i,j)}{\rho(i,j)}$. It then repeats the following steps so long as the list has at least two elements.

	a–4	b–3	c–2	d–1
0	3	3	1	1
1	0	3	2	1
2	5	0	2	0
3	0	1	2	4

(a) $B(i,j)$ at the end of the arrival phase

	a–4	b–4	c–4	d–4
0	0	1	1	1
1	0	0	1	1
2	0	0	0	0
3	0	0	0	1

(b) $x_3(i,j)$ with outputs level at 4

	a–7	b–7	c–7	d–7
0	1	3	1	1
1	0	1	2	1
2	2	0	2	0
3	0	0	0	4

(c) $x_6(i,j)$ with outputs level at 7

	a–9	b–9	c–8	d–7
0	1	3	1	1
1	0	3	2	1
2	4	0	2	0
3	0	0	1	4

(d) Final values of $x(i,j)$

	a–9	b–9	c–8	d–7
0	2	0	0	0
1	0	0	0	0
2	1	0	0	0
3	0	1	1	0

(e) $B(i,j)$ at the end of the transfer phase

Figure 4.13: Example operation of the Δ-OLA algorithm (with $\Delta = 1$).

Let j_1 and j_2 be the indices of the first two outputs on the list. Increase $x(i, j_1)$ until one of the following conditions is satisfied.

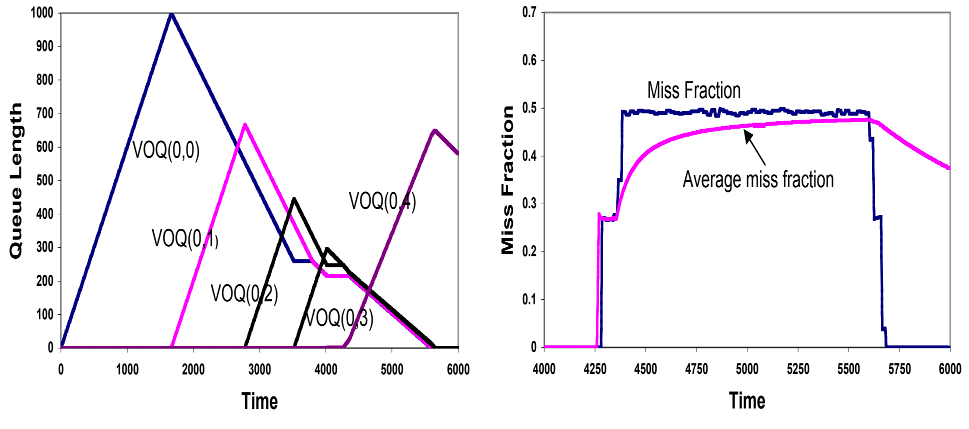
1. $\sum_j x(i,j) = ST$,
2. $x(i, j_1) = hi(i, j_1) = \rho(i, j_1)ST$,
3. $x(i, j_1) = B(i, j_1)$ *or,*
4. $B(j_1) + \frac{x(i,j_1)}{\rho(i,j_1)} = B(i, j_2) + \Delta + \frac{x(i,j_2)}{\rho(i,j_2)}$.

If condition 1 occurs, the algorithm terminates. If either of the conditions 2 or 3 occurs, remove j_1 from the list. Otherwise, move j_1 down the list so as to maintain the ordering criterion.

When the list has been reduced to a single output j, the algorithm increases $x(i, j)$ until $x(i, j) = min(hi(i, j), B(i, j))$ or until $\sum_j x(i, j) = ST$, whichever occurs first.

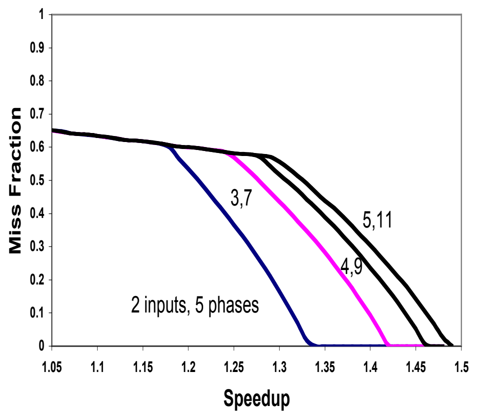
The number of steps performed by the algorithm is at most $\frac{nST}{\Delta}$. It can be implemented to run in $O((\frac{ST}{\Delta})n^2)$ time using a naive list implementation or $O((\frac{ST}{\Delta})n \log n)$ time, if the list is replaced with a heap. Using a hardware implementation of a sorted list, this can be improved to $O((\frac{ST}{\Delta})n)$ at the cost of n registers and associated comparison logic.

4.10.3 Performance Analysis

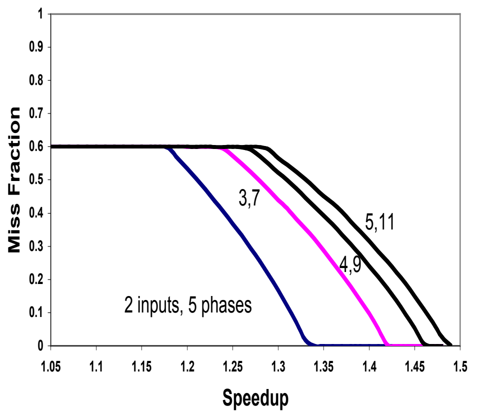


(a) Virtual Output Queue Lengths (b) Miss fraction

Figure 4.14: Sample stress test (3 inputs, 5 phases) on DOLA with speedup=1.2



(a) Delta = 0.02



(b) Delta = 0.2

Figure 4.15: Miss fractions for Distributed OLA and under a variety of stress tests with varying delta.

Fig. 4.14 shows the performance of distributed OLA on a sample stress test with 3 participating inputs and 5 phases. This example uses $\Delta = 0.1$. Comparing this to Fig. 4.7(a), we see that distributed OLA reduces the miss fraction during the critical period of the last phase by about 20% relative to DBL. For this situation, distributed OLA delivers nearly ideal performance, distributing the misses evenly among the different outputs experiencing misses. Fig. 4.15 shows how distributed OLA performs on a large number of different stress tests. Comparing these results with those in Fig. 4.8, we see that distributed OLA provides the largest improvement for small speedups. The speedups needed to reduce the misses to zero are the same for both DBL and distributed OLA.

4.11 Practical Considerations

Though the main focus of the results presented in this chapter has been on establishing the theoretical foundation for robust distributed scheduling, we believe that the results are of direct practical value. Firstly, it's important to discuss the significance of the idealized assumptions made to facilitate the analysis; specifically, the assumption that the system operation is structured in discrete phases (the three phases of a T-CIOQ switch: arrival, transfer and departure). While systems could certainly be built that adhere to this assumption, this would imply a period during which data forwarding was suspended, while scheduling was being performed. Pipelining could be used to eliminate this inefficiency. During each update period (T), a pipelined implementation would perform the scheduling needed to handle traffic received up to the start of the current update period. This traffic would then be allowed to proceed to the outputs in the next update period. This implies that all cells would experience a delay of between one and two update periods. This was our intent in defining T-work conservation. In other words, a switching system is said to be T-work conserving, if an output link is never allowed to be idle, so long as there are no cells that arrives at least $2T$ time units earlier.

In practice, it may not be preferable to adhere to this strict pipelining discipline, but to allow scheduling to proceed more or less on a continuous basis, with ports sending their status information and *asynchronously* updating the forwarding rates of their VOQ in response to the data received from other inputs. **The asynchronous update capability is a very important feature of the distributed scheduling mechanism.** This eliminates delays that are artificially imposed by the scheduling algorithm. Delays will still occur when the rate at which traffic arriving at an input for a given output increases suddenly, but during periods of relative (arrival) rate stability there would be no unnecessary delays. It should be noted however, that while the results given here provide strong evidence that such systems can be work-conserving, they do not specifically apply to them. It would be interesting to see if one could formalize such asynchronously scheduled systems so as to enable rigorous statements about work-conservation.

Another important practical issue for distributed scheduling is the overhead of the message exchanges required by the scheduling algorithms. The practical variants of the distributed scheduling algorithms described here require that each port send and receive $2n$ values, each update period (where n is the number of ports). Using a compact floating point representation, these can be encoded with accuracy in $4n$ bytes. If the update period is chosen so that the amount of data a port can send to or receive from the interconnection network per update period is much larger than $4n$, the overhead required to communicate these values can be kept acceptably small. For a system with $n = 1000$ an update period of $50\mu s$ is enough to keep the overhead below 5%.

A related issue is the computational overhead of distributed scheduling. Since, the update period is necessarily a constant multiple of the number of ports, there is time to perform even moderately complex algorithms. For a system with $n = 1000$ and a clock frequency of $200Mhz$, the DBL algorithm can be executed at each port in

$5\mu s$, a small fraction of the requried update period. While more complex algorithms such as the distributed OLA algorithm are more challenging to implement in the required time, even these are within the scope of practical implementation if Δ is at least, say $\frac{ST}{10}$.

In this thesis, we have not addressed the interconnection network itself and how it might interact with a distributed scheduler. The performance of multi-stage interconnection networks with buffered switch elements has been studied in great detail, using both analysis and simulations. The general conclusion of these studies has been that these systems can provide excellent performance when carrying traffic that does not cause sustained overloads on any output links. The use of distributed scheduling can ensure that this condition is met, allowing one to consider interconnection network performance as a largely independent issue. Most performace studies of these networks have been done assuming switch element chips that provide buffering for just a small number of cells per port (the typical range is 2-12) and these systems are capable of throughputs exceeding 90% for switch element buffer sizes of eight or more per port. Modern ICs allow the construction of switch elements with over four thousand cells, allowing system throughputs to approach 100%. With current technology, a three stage, multi-plane, Clos-type network using dynamic routing requires roughly n switch element ICs to support n OC-192 links (for values of n ranging from 100 to several 1000). Such a network can buffer several thousand cells per external link, allowing it to effectively smooth out any rate variations that may occur within an update period. Since rate-controlled VOQs feed traffic to the network in a smooth, rather than a bursty fashion, the magnitude of such variations can be expected to be quite limited, allowing the network to deliver cells to the outputs with very modest queueing delays.

4.12 Future Work

In this chapter, we have presented a comprehensive study and analysis of distributed scheduling algorithms for switches with buffered, multistage interconnection networks. There are some interesting ways in which these results can be further extended. Firstly, using simulations, we've shown that algorithms like DBL and distributed OLA are work-conserving for modest speedups. The stress test (with its high contention factor) suggests that any traffic pattern that can cause these algorithms to not be work conserving will have to be a very demanding and extreme traffic pattern. It seems possible that these algorithms (DBL and distributed OLA) are provably work-conserving for smaller speedups. However, proving such results seems to require either extensions to the proof techniques used here (adapted largely from earlier work on crossbar scheduling), or entirely new techniques. Establishing such results would be of great interest from both a theoretical and a practical perspective.

In reference [49], we described distributed scheduling algorithms that support weighted-fair queuing and algorithms that seek to guarantee that packets that arrive at the same time for the same output link are forwarded at approximately the same time on that output link. The results developed here can likely be extended to allow rigorous statements about the performance of these distributed schedulers.

Finally, as noted before, whereas crossbar schedulers must match inputs to outputs in a one-to-one fashion, distributed schedulers can subdivide the bandwidth at inputs and outputs arbitrarily. It seems likely that this difference may allow the construction of distributed schedulers with speedups smaller than 2. Our failure to prove such a result may just be a consequence of our reliance on proof methods adapted from crossbar scheduling. Our simulation studies sugest that speedups close to 1.5 may be sufficient for work-conservation in distributed schedulers. The establishment of such a result would be of considerable practical value and would also be interesting from a purely analytical standpoint, as it would likely require different proof techniques than those that have been employed so far.

Chapter 5

Concluding Remarks

In this thesis, we have reviewed the literature related to the problem of scheduling in Combined Input and Output Queued (CIOQ) switches and have identified several important issues requiring further research. We addressed the important problem of studying and improving the performance of these systems in demanding but realistic traffic conditions.

The stress test introduced in this thesis has proven to be a useful tool in these studies. Algorithms which are provably work-conserving at speedup of 2 or more require lesser speedups to perform well on the stress tests. Given the high contention factor of the stress test, this suggests that perhaps, the theoretical results include traffic patterns that are a lot more demanding and even infrequently encountered in real networks. It would be very interesting to see if a traffic pattern more demanding than the stress test (and equally likely to occur in real networks) can be designed to test these algorithms.

In this thesis, we have also asserted the importance of using the output backlog information in the scheduling decisions, whenever the switch speedups are greater than 1. Many original algorithms were all designed to work on switches with speedup of 1, when the output queue lengths are zero. We've shown that these algorithms do not perform well under extreme traffic conditions even when used in switches with speedup greater than 1. Our work has demonstrated, how the output backlog information can be included in the scheduling decision to greatly improve the performance of existing schedulers. We've used this insight to improve broad classes of schedulers including iterative, maximal matching algorithms (PIM and *i*SLIP), heursitic maximum weight matching algorithms (APSARA) and even work conserving algorithms

(LOOFA). The new algorithms, LLS-R, LLS-S, SOLIF-A and A-LOOFA are practical variants of these algorithms, that are simpler to implement, retain the desirable properties of the original algorithms and most importantly have vastly improved performance in extreme traffic conditions.

In this thesis, we have also introduced and comprehensively studied distributed scheduling. We belive that system architectures that combine distributed scheduling and buffered, multistage interconnection networks are among the most scalable and cost-effective architectures for implementing high performance routers. These architectures make it feasible today to build systems with aggregate capacities from 1 to 100 Tb/s [2]. Continued improvements in Moore's Law will allow them to contiue to scale in both line speed and total capacity. The one drawback that such systems have suffered from is that their performance can degenerate when they are subjected to extreme traffic conditions that can occur in Internet routers. While, various ad-hoc flow control techniques have been used to address this issue, it has not been possible up to this point, to make rigorous statements about the performance of such systems under extreme traffic conditions. The theoretical results developed in this thesis show that the performance of these systems can be directly comparable to the performance of unbuffered crossbars, controlled by centralized schedulers. While, in both contexts, the scheduling algorithms with the strongest theoretical guarantees are not practical to implement, these algorithms provide the insight needed to design practical variants capable of similar peroformance. The distributed, non-iterative variants DBL and DOLA introduced in this thesis are algorithms that can be readily implemented in buffered, multi-stage switching systems.

References

[1] Apeiro platform datasheet. *http://www.caspiannetworks.com/files/Apeiro_Platform_Datasheet.pdf.*

[2] Cisco carrier routing system. *http://www.cisco.com/application/pdf/en/us/guest/products /ps5763/c1031/cdccont_0900aecd800f818.pdf.*

[3] The essential core: Juniper networks t640 internet routing node with matrix technology. *http://www.juniper.net/solutions/literature/solutionbriefs/351006.pdf*, April 2002.

[4] R. Ahuja, T. Magnati, and J. Orlin. *Network Flows, Theory, Applications and Algorithms.* Prentice-Hall, 1993.

[5] A.M.Odlyzko. Comments on the larry roberts and caspian networks study of internet traffic growth. *The Cook Report on the Internet*, pages 12–15, dec 2001.

[6] T. E. Anderson, S. S. Owicki, J. B. Saxe, and C. P. Thacker. High speed switch scheduling for local area networks. *ACM Transactions on Computer Systems*, 11:319–352, November 1993.

[7] A.V.Goldberg, S.A.Plotkin, and P.M.Vaidya. Sublinear-time parallel algorithms for matching and related problems. *Journal of Algorithms*, 14:180–213, 1993.

[8] Jon Bennett and Hui Zhang. Worst case fair weighted fair queuing. In *Proc. of IEEE INFOCOM 95*, pages 120–128, Boston, MA, April 1995.

[9] Jon C. R. Bennett and Hui Zhang. Hierarchial packet fair queuing algorithms. In *Proc. of ACM SIGCOMM*, pages 43–56, Palo Alto, CA, August 1996. ACM.

[10] Giuseppe Bianchi and Jon Turner. Improved queuing analysis of shared buffer switching networks. *IEEE/ACM Transactions on Networking*, 1(4):482–490, August 1993.

[11] G. Birkhoff. Tres observaciones sobre el algebra lineal. *Univ. Nac. Tucuman Rev. Ser. A*, 5:147–151, 1946.

[12] Josep M. Blanquer and Banu Ozden. Fair queuing for aggregated multiple links. In *Proc. of ACM SIGCOMM 2001*, San Diego, CA, August 2001.

[13] S-T Chuang, A. Goel, Nick McKweon, and B. Prabhakar. Matching output queueing with a combined input output queued switch. *IEEE Journal of Selected Areas in Communication*, 17(6):1030–1039, June 1999.

[14] C. Clos. A study of non-blocking switching networks. *Bell Sys. Tech. J.*, 32(2):406–424, mar 1953.

[15] C.S.Chang, D.S.Lee, and Y.S.Jou. Load balanced birkhoff-von neumann switches, part i: one stage bufefring. In *IEEE HPSR Conference*, Dallas, May 2001.

[16] C.S.Chang, W.J.Chen, and H.Y.Huang. Birkhoff-von neumann input buffered crossbar switches. In *Proc. of IEEE INFOCOM 2000*, Tel Aviv, Israel, March 2000.

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

[18] H. Duan, J. Lockwood, S. Kang, and J. Will. A high performance oc12/oc48 queue design prototype for input buffered atm switches. In *Proc. of IEEE INFOCOM 97*, Kobe, Japan, April 1997.

[19] P. Giaconne, B. Prabhakar, and D. Shah. An implementable parallel scheduler for input-queued switches. *IEEE Micro*, 22(1):19–25, January 2002.

[20] P. Giaconne, B. Prabhakar, and D. Shah. Towards simple, high-performance schedulers for high-aggregate bandwidth schedulers. In *Proc. of the Twenty-First IEEE Conference on Computer Communications (INFOCOM)*, New York, NY, June 2002.

[21] S. Jamaloddin Golestani. A self clocked fair queuing scheme for broadband applications. In *Proc. of IEEE INFOCOM 94*, pages 636–646, Toronto, Canada, June 1994.

[22] J. E. Hopcroft and R. M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM journal of computing*, 2(4):225–231, dec 1973.

[23] J. Hui and E. Arthurs. A broadband packet switch for integrated transport. *IEEE Journal on Selected Areas of Communications*, 5(8):1264–1273, October 1987.

[24] Frank K. Hwang. A survey of nonblocking multicast three-stage clos networks. *IEEE communications magazine*, 41(10):34–37, oct 2003.

[25] Sundar Iyer and Nick Mckeown. Maximum size matching and input queued switches. In *Proc. of the 40thth Allerton Conference on Communication, Control, and Computing*, 2002.

[26] Andrzej Jajszczyk. Nonblocking, repackable and rearrangeable clos networks: fifty years of evolution. *IEEE communications magazine*, 41(10):28–33, oct 2003.

[27] J.G.Dai and B.Prabhakar. The throughout of data switches with and without speedup. In *Proc. of IEEE INFOCOM 2000*, Tel Aviv, Israel, March 2000.

[28] Ying Jiang and M. Hamdi. A fully desynchronized round-robin matching scheduler for a voq packet switch architecture. *IEEE workshop on high performance switching and routing*, pages 407–411, 2001.

[29] M. Karol, K. Eng, and H. Obara. Improving the performance of input-queued atm packet switches. In *Proc. of IEEE INFOCOM 92*, pages 110–115, Florence, Italy, May 1992.

[30] M. Karol, M. Hluchyj, and S. Morgan. Input versus output queuing on a space division switch. *IEEE Trans. Comm*, 35(12):1347–1356, 1987.

[31] Isaac Keslassy, Shang-Tse Chuang, Kyoungsik Yu, David Miller, Mark Horowitz, Olav Solgaard, and Nick McKeown. Scaling internet routers using optics. In *ACM SIGCOMM*, Karlsruhe, Germany, August 2003.

[32] Isaac Keslassy and Nick McKeown. Analysis of a scheduling algorithms that provide 100% throughput in input-queued switches. In *In Proceedings of the 39th Annual Allerton Conference on Communication, Computing and Control*, Monticello, Illinois, oct 2001.

[33] Isaac Keslassy, Rui Zhang-Shen, and Nick McKeown. Maximum size matching is unstable for any packet switch. *IEEE Communications Letters*, 7(10):496–498, oct 2003.

[34] Pattabhiraman Krishna, Naimish S. Patel, Anna Charny, and Robert j. Simcoe. On the speedup required for work-conserving crossbar switches. *IEEE Journal on Selected Areas of Communications*, 17(6):1057–1065, June 1999.

[35] Fred Kuhns, John Dehart, Anshul Kantawala, Ralph Keller, John Lockwood, Prashanth Pappu, David Richard, David Taylor, Jyoti Parwatikar, Ed Spitznagel, Jon Turner, and Ken Wong. Design and evaluation of a high performance dynamically extensible router. *Proceedings of the DARPA Active Networks Conference and Exposition*, 2002.

[36] S.-Y Li. Theory of periodic contention and its application to packet switching. In *Proc. of IEEE INFOCOM 88*, March 1988.

[37] Jing Liu, Hung Chun Kit, Mounir Hamdi, and Chi Ying Tsui. Stable round-robin scheduling algorithms for high-performance input queued switches. In *Proc. of the 10th Symposium on high performance interconnects HOT Interconnects (HotI'02)*, Stanford, CA, August 2002.

[38] John W Lockwood. *Design and implementation of a multicast, input-buffered ATM switch for the iPOINT Testbed.* PhD thesis, University of Illinois at Urbana-Champaign, 1995.

[39] L.Tassiulas. Linear complexity algorithms for maximum throughput in radio networks and input queued switches. In *Proc. of IEEE INFOCOM 98*, San Francisco, CA, April 1998.

[40] M.A.Marsan, A. Bianco, E. Leonardi, and L. Milia. Rpa: A flexible scheduling algorithm for input buffered switches. *IEEE Transactions on Communications*, 47(12):1921–1932, December 1999.

[41] M. Ajmone Marsan, A. Bianco, E. Filippi, P. Giaccone, E. Leonardi, and F. Neri. On the behavious of input queued switch architectures. *Euro. Trans. TeleCommun.*, 10(2):111–124, mar 1999.

[42] N. McKeown. islip: A scheduling algorithm for input queued switches. *IEEE Transactions on Networking*, 7(2), April 1999.

[43] N. Mckeown, V. Anantharam, and J. Walrand. Achieving 100% throughput in an input queued switch. In *Proc. of IEEE INFOCOM 96*, San Francisco, CA, March 1996.

[44] Nick Mckeown. *Scheduling algorithms for input queued cell switches.* PhD thesis, University of California at Berkeley, 1995.

[45] Nick McKeown and Thomas E. Anderson. A quantitative comparison of scheduling algorithms for input-queued switches. *Computer Networks and ISDN Systems*, 30(24):2309–2326, December 1998.

[46] Nick Mckeown, Adisak Mekkittikul, Venkat Anantharam, and Jean Walrand. Achieving 100% throughput in input queued switches. *IEEE Transations on Communications*, 47(8), August 1999.

[47] Adisak Mekkittikul. *Scheduling non-uniform traffic in high speed packet switches and routers.* PhD thesis, Stanford University, 1998.

[48] Adisak Mekkittikul and Nick Mckeown. A practical scheduling algorithm to achieve 100% throughput in input queued switches. In *Proc. of IEEE INFOCOM 98*, San Francisco, CA, April 1998.

[49] Prashanth Pappu, Jyoti Parwatikar, Jonathan Turner, and Ken Wong. Distributed queuing in scalable high performance routers. In *Proc. of IEEE INFOCOM 2003*, San Francisco, CA, March 2003.

[50] Prashanth Pappu and Jon Turner. Stress resistant algorithms for cioq switches. In *International Conference on Network Protocols (ICNP)*, Atlanta, GA, November 2003.

[51] Prashanth Pappu and Tilman Wolf. Scheduling processing resources in programmable routers. In *Proc. of the Twenty-First IEEE Conference on Computer Communications (INFOCOM)*, New York, NY, June 2002.

[52] Abhay K. Parekh and Robert G. Gallager. A generalized processor sharing approach to flow control: The single node case. In *Proc. of IEEE INFOCOM 92*, pages 915–924, Florence, Italy, May 1992.

[53] Abhay K. Parekh and Robert G. Gallager. A generalized processor sharing approach to flow control in integrated service networks: The single node case. *IEEE/ACM Trans. on Computer Networks*, 1(3):344–357, June 1993.

[54] Abhay K. Parekh and Robert G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: The multiple node case. In *Proc. of IEEE INFOCOM 93*, pages 521–530, Florence, Italy, March 1993.

[55] Thomas G. Robertazzi. *Performance evaluation and high speed switching fabrics and networks: ATM, Broadband ISDN and MAN technology.* Wiley-IEEE Press, apr 1993.

[56] T. L. Rodeheffer and J. B. Saxe. An efficient matching algorithm for a high-throughput, low-latency data switch. *Compaq Computer Communications Research Center, Research Report 162.*, 1998.

[57] R.R.Schaller. Moore's law: Past, present and future. *IEEE Spectrum*, 34(6):52–59, June 1997.

[58] Multi stage switching systems using optical WDM grouped links based on dynamic bandwidth sharing. A survey of nonblocking multicast three-stage clos networks. *IEEE communications magazine*, 41(10):56–63, oct 2003.

[59] Ion Stoica, Hui Zhang, and T. S. Eugene Ng. A hierarchical fair service curve algorithm for link-sharing. In *Proc. of ACM SIGCOMM 97*, Cannes, France, August 1997.

[60] Ted Szymanski and Salman Shaikh. Markov chain analysis of packet switched banyan networks with arbitrary switch sizes, queue sizes, link multiplicites and speedups. In *Proc. of IEEE INFOCOM 89*, April 1989.

[61] Y. Tamir and G. Frazier. High performance multi-queue buffers for vlsi communication switches. In *Proc. of the 15th Ann. Symp. on Computer Architecture*, pages 343–354, June 1988.

[62] R. E. Tarjan. *Data structures and network algorithms.* Bell Labs, 1983.

[63] Jon Turner. Queueing analysis of buffered switching networks. *IEEE Transactions on Communications*, 41(2):412–420, February 1993.

[64] Jon Turner and Ricardo Melen. Multirate clos networks. *IEEE Communications Magazine*, 41(10):38–44, oct 2003.

[65] J. von Neumann. A certain zero-sum two-person game equivalent to the optimal assignment problem. *Contributions to the Theory of Games*, 2:5–12, 1953.

[66] Jenq. Yih-Chyun. Performance analysis of a packet switch based on a single-buffered banyan network. *IEEE Journal on Selected Areas of Communications*, 1(6):1014–1021, December 1983.

[67] Hui Zhang. Service disciplines for guaranteed performance service in packet switching networks. *Proc. of the IEEE*, 83(10):1374–96, October 1995.

Vita

Prashanth Pappu

Date of Birth	August 18, 1978
Place of Birth	Hyderabad, AP (INDIA)
Degrees	B.Tech., Computer Science, May 1999 Indian Institute of Technology, Madras, India. M.S., Computer Science, December 2002 Washington University in St. Louis, St. Louis, U.S.A. Ph.D., Computer Engineering, August 2004 Washington University in St. Louis, St. Louis, U.S.A.
Professional Societies	Institute of Electrical and Electronics Engineers
Publications	Prashanth Pappu and Tilman Wolf, "Scheduling Processing Resources in Programmable Routers". In IEEE Infocom 2002, New York, USA Prashanth Pappu, Jonathan Turner and Ken Wong, "Distributed Queuing in Scalable High Performance Routers ". In IEEE Infocom 2003, San Francisco, USA Prashanth Pappu and Jonathan Turner, "Stress Resistant Scheduling Algotihms for CIOQ Switches". In ICNP 2003, Atlanta, USA Prashanth Pappu, Jonathan Turner and Ken Wong, "Work Conserving Distributed Scheduling Algorithms for Terabit Routers". In ACM SIGCOMM 2004, Portland, USA

August 2004