Scheduling Algorithms for Input-Queued Cell Switches

by

Nicholas William McKeown

B.Eng (University of Leeds) 1986

M.S. (University of California at Berkeley) 1992

A thesis submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy
in

Engineering-Electrical Engineering
and Computer Sciences

in the
GRADUATE DIVISION
of the
UNIVERSITY of CALIFORNIA at BERKELEY

Committee in charge:

Professor Jean Walrand, Chair
Professor Pravin P. Varaiya
Professor Ronald W. Wolff

1995

This thesis of Nicholas William McKeown is approved:

Chair	Date
	Date
	Date

University of California at Berkeley

1995

Scheduling Algorithms for Input-Queued Cell Switches

by

Nicholas William McKeown

Abstract

Scheduling Algorithms for Input-Queued Cell Switches

by

Nicholas William McKeown

Doctor of Philosophy in Engineering-Electrical Engineering and Computer Sciences

University of California at Berkeley

Professor Jean Walrand, Chair

The algorithms described in this thesis are designed to schedule cells in a very high-speed, parallel, input-queued crossbar switch. We present several novel scheduling algorithms that we have devised, each aims to match the set of inputs of an input-queued switch to the set of outputs more efficiently, fairly and quickly than existing techniques.

In Chapter 2 we present the simplest and fastest of these algorithms: SLIP — a parallel algorithm that uses rotating priority ("round-robin") arbitration. SLIP is simple: it is readily implemented in hardware and can operate at high speed. SLIP has high performance: for uniform i.i.d. Bernoulli arrivals, SLIP is stable for any admissible load, because the arbiters tend to *desynchronize*. We present analytical results to model this behavior. However, SLIP is not always stable and is not always monotonic: adding more traffic can actually make the algorithm operate more efficiently. We present an approximate analytical model of this behavior. SLIP prevents starvation: all contending inputs are eventually served. We present simulation results, indicating SLIP's performance. We argue that SLIP can be readily implemented for a 32x32 switch on a single chip.

In Chapter 3 we present *i*-SLIP, an iterative algorithm that improves upon SLIP by converging on a maximal size match. The performance of *i*-SLIP improves with up to $\log_2 N$ iterations. We show that although it has a longer running time than SLIP, an *i*-SLIP scheduler is little more complex to implement.

In Chapter 4 we describe maximum or maximal *weight* matching algorithms based on the occupancy of queues, or waiting times of cells. These algorithms are stable over a wider range of traffic loads. We describe two algorithms, *longest queue first* (LQF) and *oldest cell first* (OCF) and consider their performance. We prove that LQF, although too complex to implement in hardware, is stable under all admissible i.i.d. offered loads. We consider two implementable, iterative algorithms *i*-LQF and *i*-OCF which converge on a maximal weight matching. Finally, we present two interesting implementations of the Gale-Shapley algorithm, designed to solve the *stable marriage problem*.

To my parents,

and

My Wife,
My Love,
My Le.

Table of Contents

Acknowledgements viii

CHAPTER 1
Introduction 1

1 Problem Statement 1
2 Motivation 4
2.1 Datapath for an Input-Queued Switch 4
2.2 Controlling the Datapath 5
3 Background 7
3.1 Input vs. Output Queueing 7
3.2 Overcoming Head-of-Line Blocking 8
3.3 Previous Scheduling Work 9
3.3.1 Maximum Size Matching 9
3.3.2 Neural Network Algorithms 10
3.3.3 Scheduling into the Future 11
3.3.4 Parallel Iterative Matching 12
3.4 Simple Comparison of Previous Techniques 14
4 Outline of Thesis 16

CHAPTER 2
The SLIP Algorithm with a Single Iteration 18

1 Introduction 18
2 Basic Round-Robin Matching Algorithm 20
2.1 Performance of RRM for Bernoulli Arrivals 20
3 The SLIP Algorithm 23
4 Simulated Performance of SLIP 24
4.1 Bernoulli Traffic 24
4.2 “Bursty” Traffic 26
4.3 As a Function of Switch Size 28
4.4 Burst Reduction 30
5 Analysis of SLIP Performance 32

5.1 Convergence to Time-Division Multiplexing Under Heavy Load 32
5.2 Desynchronization of Arbiters 32
5.3 Stability of SLIP 35
5.3.1 Drift Analysis of a 2x2 SLIP Switch: First Approximation 38
5.3.2 Drift Analysis of a 2x2 SLIP Switch: Second Approximation ...41
5.4 Approximate Delay Model for 2x2 SLIP Switch 41
6 Variations on SLIP 44
6.1 Prioritized SLIP 44
6.2 Threshold SLIP 45
6.3 Weighted SLIP 46
6.4 Least Recently Used 46
7 Implementing SLIP 49
7.1 Prioritized SLIP 51

CHAPTER 3
The SLIP Algorithm with Multiple Iterations 53

1 Introduction 53
2 The Iterative SLIP Matching Algorithm 54
2.1 Description 54
2.2 Updating Pointers 55
2.3 Properties 56
3 Simulated Performance of Iterative SLIP 57
3.1 How Many Iterations? 57
3.2 Bernoulli Traffic 58
3.3 Bursty Traffic 63
3.4 As a Function of Switch Size 65
4 Variations of Iterative SLIP 66
4.1 Iterative SLIP with LRU Accept Arbiters 66
4.2 Separate Pointers for each Iteration 69
5 Implementing Iterative SLIP 69

CHAPTER 4
Weighted Matching Algorithms 72

1 Introduction 72
2 Maximum Weight Matching 73
2.1 Starvation with LQF 74
2.2 Performance of LQF and OCF Algorithms 75
2.2.1 Uniform Workload 75
2.2.2 Non-Uniform Workload 75
2.2.3 Stability of 2x2 Switch 77
2.2.4 Stability of NxN Switch 80
3 Iterative Maximal Weight Matching Algorithms 81
3.1 i-LQF 81
3.2 Properties 82
3.3 i-OCF 82
3.4 Properties 83
3.5 Performance of i-LQF and i-OCF 83
3.5.1 Uniform Workload 83
3.5.2 Nonuniform Workload 83
3.5.3 Stability for 2x2 Switch 83
3.6 Implementation of i-LQF and i-OCF 84
4 Stable Marriages 87
4.1 The Stable Marriage Problem 88
4.2 The Gale-Shapley Algorithm 88
4.3 Analogy to Switch Scheduling 89
4.3.1 The GS-LQF Algorithm 90
4.3.2 The GS-OCF Algorithm 90
4.4 Performance of GS-LQF and GS-OCF Algorithms 91
4.5 Implementation of GS-LQF and GS-OCF 91
4.6 Egalitarian Stable Marriage 92

References 94

APPENDIX 1
Arbiter Synchronization for Single-Iteration SLIP 97

APPENDIX 2
Stability of Single-Iteration SLIP Algorithm 101

1 Single-Step Drift Analysis of 2x2 Switch with 1 Queue 101
1.1 First Approximation.....101
1.2 Second Approximation 103
2 Matrix Geometric Solution for 2x2 Switch with 1 Queue.....106

APPENDIX 3
Stability of 2x2 Switch 109

1 Stability of 2x2 Switch with 3 Active Flows (Theorem 4.2).....109
1.1 Definitions.....109
1.2 Problem statement.....110
1.3 Solution.....110
1.4 Stable Algorithms 111
2 Relative Queue Sizes (Theorem 4.3) 111

APPENDIX 4
Stability of NxN Switch with i.i.d. Arrivals 115

1 Definitions.....115
2 Main Theorem.....116
3 Proof.....116

Acknowledgements

For their continued guidance, support and encouragement throughout my time at Berkeley, I would like to thank my adviser Jean Walrand and Pravin Varaiya. I greatly appreciate the freedom and collegial respect you have given me and your other students.

Numerous discussions with Richard Edell lead to the design of the datapath that provided the main motivation for this thesis. Richard, you are a truly gifted engineer and it has been a pleasure to be your colleague.

I am grateful to Professor Tom Anderson for discussions about the iterative properties of the SLIP algorithm, and to Professor Venkat Anantharam for suggesting the proof in Appendix 4. I also wish to acknowledge the helpful feedback and suggestions of Chuck Thacker (DEC SRC), the inventor of parallel iterative matching. I thank Dana Randall for introducing me to the stable marriage problem, and Matthew J. Salzman (CMU) for kindly donating his code to implement the maximum match algorithms used in many of my simulations.

I am extremely grateful to John Limb, for whom I worked at Hewlett-Packard Labs in Bristol, England. John, you have been a constant source of inspiration to me; and without your encouragement I would not have gone back to school to pursue my Ph.D.

I would like to give thanks to the numerous other people at Hewlett-Packard Labs who supported me over the years, in particular John Taylor, Daniel Pitt, Steve Wright and Gwenda Ward.

Last, and definitely most, I want to thank my family. Words cannot express my thanks to my wife and parents for all your love and encouragement. I dedicate this thesis to you.

CHAPTER 1

Introduction

1 Problem Statement

Consider the “input-queued cell switch” in Figure 1.1 connecting m inputs to n outputs. The arrival process $A_i(t)$ at input i, $1 \le i \le m$, is a discrete-time process of fixed sized packets, called cells.[1] At the beginning of each time slot, either zero or one cells arrive at each input. Each cell contains an identifier that indicates which output j, $1 \le j \le n$, it is destined for. When a cell destined for output j arrives at input i it is placed in the FIFO queue $Q(i,j)$ which has occupancy $L_{i,j}(t)$. We shall define the arrival process $A_{i,j}(t)$ as the process of arrivals at input i for output j at rate $\lambda_{i,j}$, and the set of arrival processes $A(t) = \{A_i(t); 1 \le j \le m\}$. A(t) is considered *admissible* if no input or output is oversubscribed, i.e. $\sum_i \lambda_{i,j} < 1, \sum_j \lambda_{i,j} < 1$, otherwise it is *inadmissible*.

The FIFO queues are served as follows. A scheduling algorithm selects a conflict-free match M between the set of inputs and outputs such that each input is connected to at most one output and each output is connected to at most one input. At the end of the time slot, if input i is connected to output j, one cell is removed from $Q(i,j)$ and sent to output j. Clearly, the departure process from output j, $D_j(t)$, rate μ_j is also a discrete-time process with either zero or one cell

1. Unless otherwise stated, we will assume that A_i(t) is stationary and ergodic.

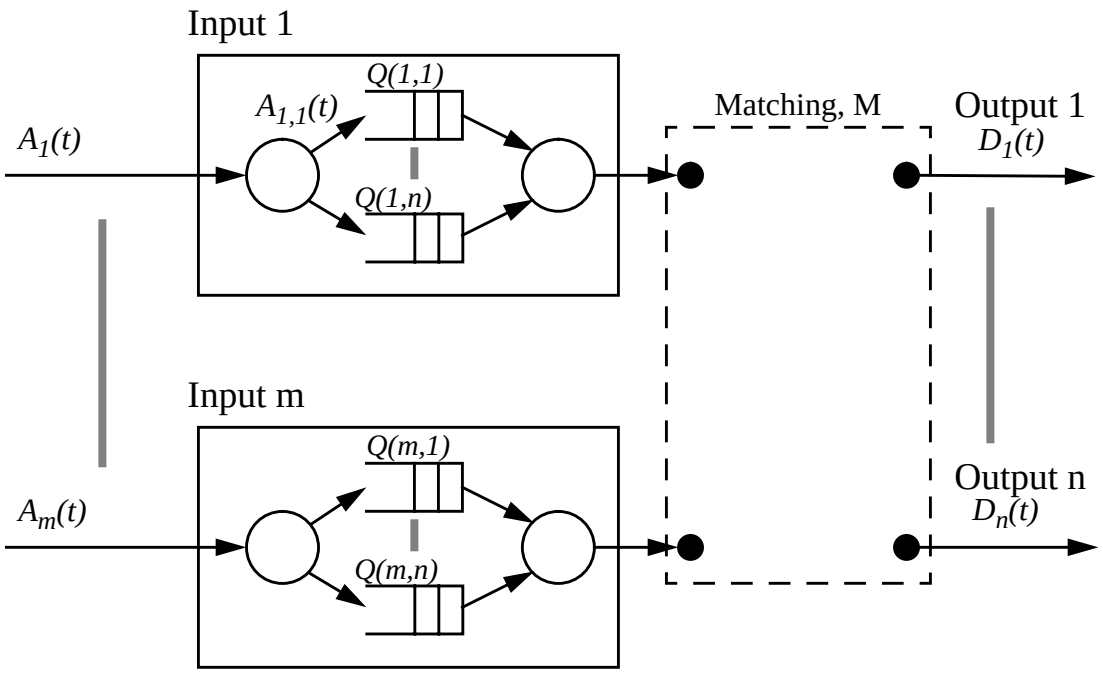


Figure 1.1 Components of an Input-Queued Cell-Switch.

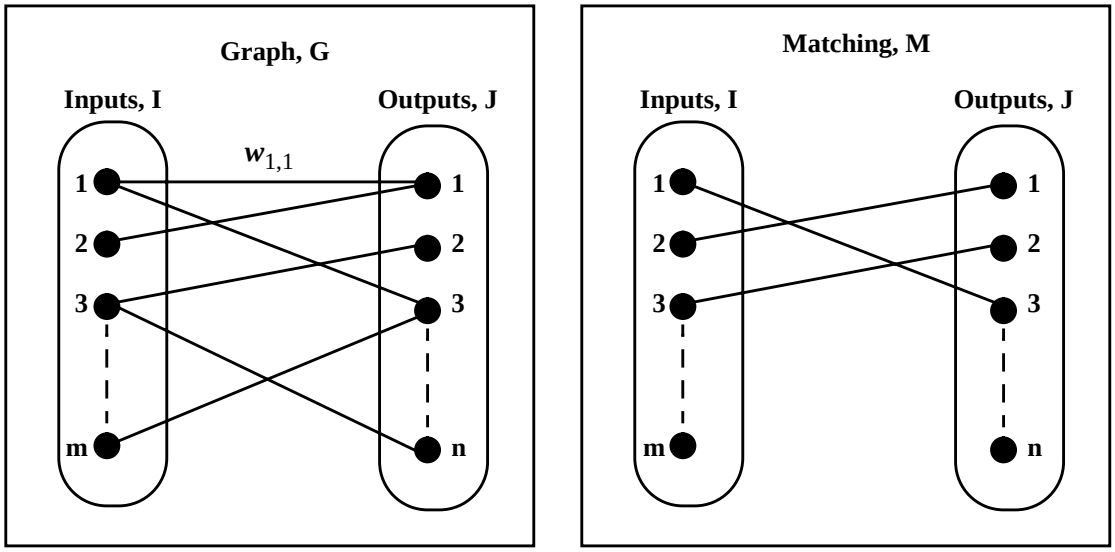
departing from each output at the end of each time slot. We shall define the departure process $D_{i,j}(t)$, rate $\mu_{i,j}$, as the process of departures from output j that were received from input i.

To find a matching M, the scheduling algorithm solves a bipartite graph matching problem. An example of a bipartite graph is shown in Figure 1.2.

All of the scheduling algorithms described in this thesis attempt to match the set of inputs I, of an input-queued switch, to the set of outputs J. For our application, we will assume that $|\mathrm{I}| = m = |\mathrm{J}| = n = \mathrm{N}$, where N is the number of ports. In each algorithm, if the queue $Q(i,j)$ is non-empty, $L_{i,j}(t) > 0$ and there is an edge in the graph G between input i and output j. The meaning of the weights depend on the algorithm. For example, in some algorithms the weight is always equal to one, indicating whether the queue is empty or non-empty. In other algorithms, the weight $w_{i,j}$ may be integer-valued, equalling for example $L_{i,j}(t)$.

There are a number of properties that we desire for all scheduling algorithms:

- *Efficiency* — An efficient algorithm is one that serves as many input-queues as possible in each match. In general, the maximum matching problem does not have a solution that



a) Example of G for $|I| = m$ and $|J| = n$. **b) Example of matching M on G.**

Figure 1.2 Define G = [V,E] as an undirected graph connecting the set of vertices V with the set of edges E. The edge connecting vertices *i*, $1 \le i \le m$ and *j*, $1 \le j \le n$ has an associated weight denoted $w_{i,j}$. Graph G is bipartite if the set of inputs I = {i: $1 \le i \le m$} and outputs J = {*i:* $1 \le j \le n$} partition V such that every edge has one end in I and one end in J. Matching M on G is any subset of E such that no two edges in M have a common vertex. A *maximum matching algorithm* is one that finds the matching M_{max} with the maximum total size or total weight.

can be calculated quickly in hardware, and so each of the algorithms that we describe finds a sub-maximum match M_{sub}, where: $|M_{sub}| \le |M_{max}|$.

- *Stability* — For a given admissible traffic pattern, we define an algorithm as *stable* if the expected occupancy of every input queue *Q(i,j)* is finite, i.e. $E[L_{i,j}(t)] < \infty$. For a given algorithm, we call a stationary traffic pattern *sustainable* if it does not cause the switch to become unstable.
- *No Starvation* — We shall describe a non-empty input-queue as *starved* if, for a given traffic pattern and scheduling algorithm, it remains unserved indefinitely.
- *Fast* — To achieve the highest bandwidth switch, it is important that the scheduling algorithm does not become the performance bottleneck. The algorithm should therefore find a match as quickly as possible.
- *Simple to implement* — If the algorithm is to be fast in practice, it must be implemented in special-purpose hardware. The implementation complexity includes the amount of state that the scheduler must maintain, the amount of logic required to make a decision based on the state, and the amount of communication required to update the state at the beginning and end of each cell time.

2 Motivation

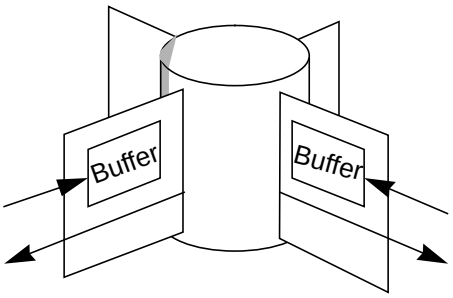
The scheduling algorithms described in this thesis are applicable to all input-queued switches. However, the work was motivated by a single goal: to find a simple algorithm that can schedule cells in a high-speed, parallel input-queued crossbar switch. We may partition such a switch into two main components: the datapath and the scheduler. Designing a high-bandwidth datapath is straightforward; it is the scheduling algorithm that is complex. To illustrate this point we begin with the example of an extremely high-bandwidth datapath that we have devised. We will then describe how this datapath can be controlled by a central scheduler. We then discuss the problem of scheduling cells for such a datapath.

2.1 Datapath for an Input-Queued Switch

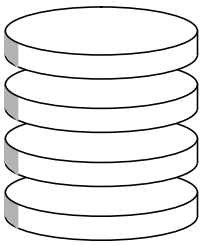
An example of a high-speed datapath is shown in Figure 1.3. This switch is shown to illustrate that it is feasible to build a small switch with extremely high aggregate bandwidth in current CMOS technology. Figure 1.3(a) shows the general structure of the switch: switch port cards connect to a central switching hub; when cells arrive at the switch port card, they are buffered while waiting to be transferred through the hub. As shown in Figure 1.3(b), the switching hub is composed of a stack of identical bit-slices and for a small number of ports (for example, 32 ports or less) is readily implemented using a crossbar switch. Plan and side elevations of each bit-slice are shown in detail in Figure 1.3(c). Each bit-slice is a single layer printed circuit board containing a 1-bit NxN switching chip. The switching chip is connected to every port card via a two-bit connector: one bit to receive from and one bit to transmit to the port card.

The main advantages of this switch architecture are:

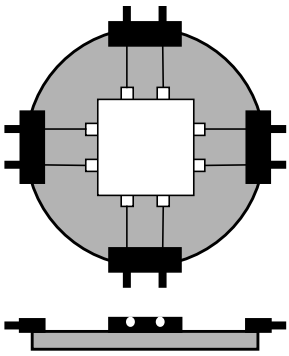
- The bit-slice is extremely simple. No leads need to cross, reducing crosstalk and allowing the slice to be constructed from a single layer printed circuit board.
- The lead lengths connecting each port card to the central switch are all of identical and minimum length. This reduces skew and the effect of reflections, enabling high data rates and means that each bit-slice can be small. For example, a slice for a 32-port switch could be just 2 inches in diameter.
- By switching multiple bits in parallel, extremely high aggregate bandwidths are achievable. For example, for a 32-port switch with 32 bit-slices and a clock-rate of 100MHz for



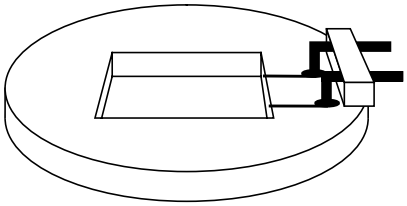
a) The switch consists of a central, vertical hub. Each interface card connects radially into the hub. This example shows a 4-port switch.



b) The central hub consists of multiple, identical bit-slices stacked vertically. This example shows a stack of 4 bit-slices.



c) Each bit-slice contains a single switch bit-slice chip, a 2-bit connector to each port card (one bit for each direction). This example shows a single 4x4 bit chip.



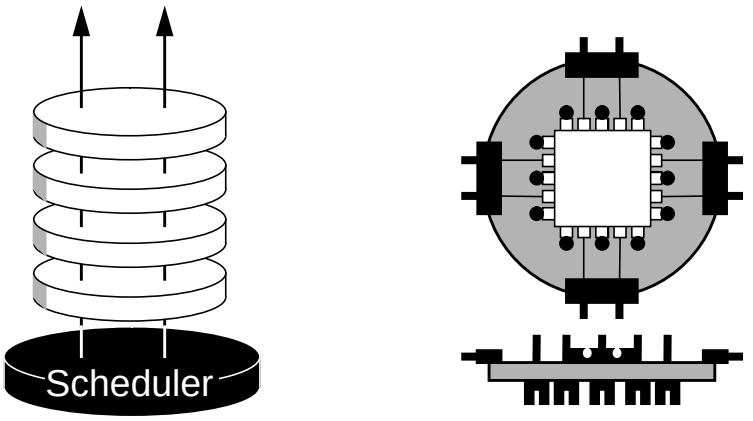
d) Detail of construction of bit-slice. This example only shows a single edge connector.

Figure 1.3 The datapath for a parallel, bit-sliced input-queued switch.

each switching chip (easily achievable in current CMOS technology), an aggregate bandwidth of 100Gbps is achievable. In the extreme, if the parallel path is as wide as a single ATM cell (424 bits), a 32 port switch operating at 100MHz would have an aggregate bandwidth in excess of 1 terabit per second!

2.2 Controlling the Datapath

It is necessary for the datapath to be configured at the beginning of each cell time. In this design, we assume that a centralized scheduler examines the state of the input queues and selects a conflict-free match between inputs and outputs. This configuration is then loaded into all bit-slices in parallel, as shown in Figure 1.4.



a) A centralized scheduler determines the configuration of the crossbar and loads it vertically through the hub.

b) Each slice is connected to the slice above and below so that the switch configuration can be loaded in parallel to all slices.

Figure 1.4 Extension of datapath to control configuration.

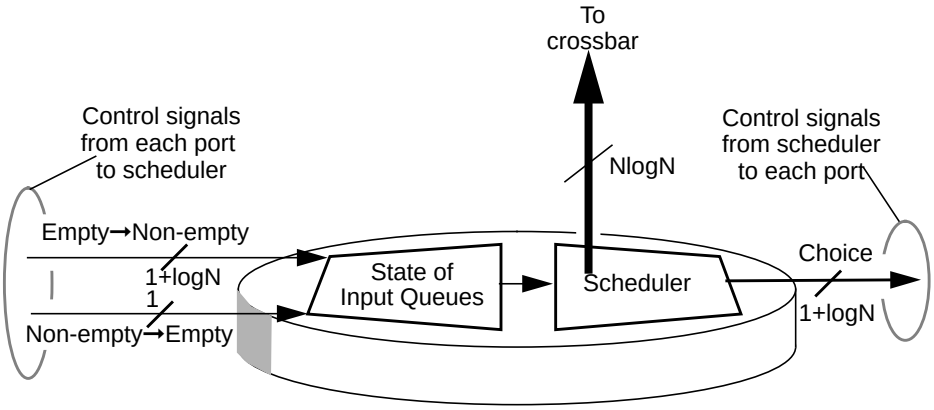


Figure 1.5 Connections to and from each port and a centralized scheduler.

In Figure 1.5 we consider the number of connections to and from each port and the centralized scheduler, assuming that the scheduler maintains N^2 state bits indicating whether each input queue is empty or non-empty. At the beginning of each cell time, each input port may receive at most one new arrival. If as a result of the arrival input queue *Q(i,j)* changes from empty to non-empty, then input *i* must notify the scheduler, passing the value *j*. It may do this with $\log N$ bits, and one extra

bit to indicate that the value is valid. At the end of the arbitration time the scheduler must notify each input at most one output that it may transmit a cell to, once again requiring $1 + \log N$ bits. The scheduling decision may result in *Q(i,j)* changing from non-empty to empty, requiring input *i* to indicate this to the scheduler. Because the scheduler knows which output *j* was scheduled, input *i* requires only 1 bit to indicate this information.

The scheduler must also indicate its decision to the switch datapath. It may do this by notifying each output which input it is connected to, requiring a total of NlogN bits.[1] The crossbar loads this configuration by turning on or off each switching element.

3 Background

3.1 Input vs. Output Queueing

The long-standing view has been that input-queued switches are impractical because of poor performance. If FIFO queues are used to queue cells at each input, only the first cell in each queue is eligible to be forwarded. As a result, FIFO input queues suffer from *head of line* (HOL) blocking; if the cell at the front of the queue is blocked, other cells in the queue cannot be forwarded to other unused inputs. It is well known that for an input-queued switch with Bernoulli i.i.d. arrivals with destinations uniformly distributed over all outputs the maximum achievable throughput is limited to just 58% when the number of ports is large [22]. For periodic traffic, HOL blocking can lead to even worse performance [33] and as a result the standard approach has been to abandon input queueing and instead use output queueing.

With output queueing the bandwidth of the internal interconnect is increased, allowing multiple cells to be forwarded at the same time to the same output, and queued there for transmission on the output link. The main advantage of output queueing is that all cells are delayed by a fixed amount making it possible to control delay through the switch. This is why schemes that schedule cells to provide absolute or statistical performance guarantees assume output queueing [8], [10],

1. Alternatively, the scheduler may notify the datapath for each input which output it is connected to. For unicast traffic this would be sufficient and equivalent. However, if the datapath is used for multicast traffic, an input may be connected to *multiple* outputs requiring a list of outputs to configure an input. Because an output can still receive a cell from at most one input, it is still sufficient to indicate to each output which input is connected to.

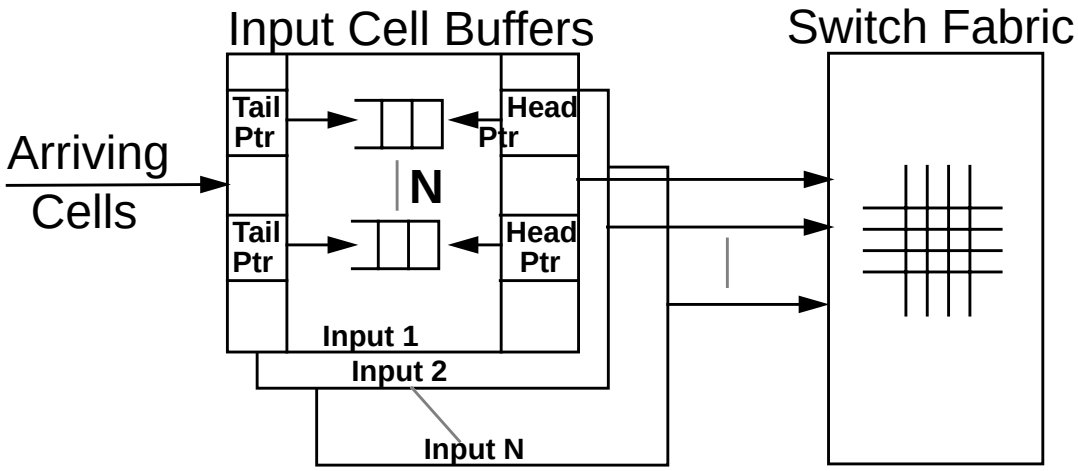


Figure 1.6 Head of line blocking can be eliminated by using a separate queue for each output at each input.

[29], [30], [31], [44], [45]. This is not generally possible with input-queued switches due to variations in delay caused by contention for the switching fabric and queueing at the input.The main disadvantage of output queueing is that for a N-port switch, the internal interconnect and output queues must operate at N times the line rate. In applications where the number of ports is large or the line rate is high, this makes output queueing impractical.

3.2 Overcoming Head-of-Line Blocking

Our work is motivated by the desire to achieve the highest data rate for a given technology. This forces us to consider only input-queued switches and to try and overcome the limitations of HOL blocking. Many techniques have been suggested for reducing HOL blocking, for example by considering the first K cells in the FIFO queue, where K>1 [6], [19], [23]. Although these schemes can improve throughput, they are highly sensitive to traffic arrival patterns and perform no better than regular FIFO queueing when the traffic is bursty.

But HOL blocking can be eliminated entirely by using a simple buffering strategy at each input port. Rather than maintain a single FIFO queue for all cells, each input maintains a separate queue for each output [3], [24], [40], as shown in Figure 1.6. HOL blocking is eliminated because

a cell cannot be held up by a cell queued ahead of it that is destined for a different output. This implementation is slightly more complex, requiring N FIFOs to be maintained by each input buffer. But no additional speedup is required: at most one cell can arrive and depart from each input in a cell time.

3.3 Previous Scheduling Work

In this section we summarize a selection of scheduling algorithms for input-queued switches described in the literature. All of these algorithms are for switches that avoid HOL blocking using the scheme described above. Each algorithm attempts to find either a maximum *size*[1] matching, or attempts to schedule a cell on arrival at the earliest possible time in the future.

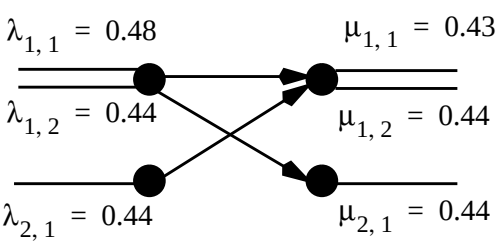
3.3.1 Maximum Size Matching

The maximum size matching for a bipartite graph can be found by solving an equivalent network flow problem [41]. We will call this algorithm *maxsize*. There exist many algorithms for solving these problems, the most efficient currently known converges in $O(n^{5/2})$ time and is described in [17].[2] The problem with this algorithm is that although it is guaranteed to find a maximum match, for our application it is too complex to implement and takes too long to complete.

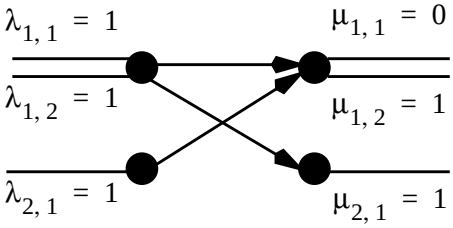
It is important to note that a maximum *size* matching is not necessarily desirable. First, under *admissible* traffic it can lead to instability and unfairness, particularly for non-uniform traffic patterns. An example of this behavior for a 2x2 switch is shown in Figure 1.7(a). Arrivals to the switch are i.i.d. Bernoulli arrivals and the performance was obtained using simulation. Even though the traffic is admissible, it cannot be sustained by the maximum size matching algorithm.[3] Second, under *inadmissible* traffic, the maximum size matching algorithm can lead to *starvation.* An example of this behavior is shown in Figure 1.7(b). It is clear that because all three queues are permanently occupied, the algorithm will always select the “cross” traffic: input 1 to output 2 and input 2 to output 1.

---

1. In some literature, the maximum *size* matching is called the maximum *cardinality* matching or just the maximum bipartite matching.
2. This algorithm is equivalent to Dinic’s algorithm [9].
3. Later, we will look at particular values under which the maximum size matching algorithm is unstable.



a) Even under an *admissible* workload of Bernoulli arrivals, the maximum size match can be unstable.



b) Under an *inadmissible* workload, the maximum size match will always serve just two queues, *starving* the flow from input 1 to output 1.

Figure 1.7 Example of *instability* using a maximum size matching algorithm for a 2x2 switch with 3 offered flows.

3.3.2 Neural Network Algorithms

Hopfield neural networks have also been used to approximate maximum size matchings for bipartite graphs [2], [5], [34], [42]. For an N-port switch, the neural network comprises N^2 neurons; each neuron is implemented by an analog amplifier and RC circuit. At the beginning of each cell time, the neural net is loaded with the state matrix, V = [$v_{i,j}$] where $v_{i,j} = 1$ if $L_{i,j}(t) > 0$, else $v_{i,j} = 0$. For example, in [2] the circuit is designed to minimize the following quadratic energy function [18]:

$$E = \frac{A}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\sum_{\substack{l=1\\ l\neq j}}^{N} v_{ij}v_{il} + \frac{B}{2}\sum_{j=1}^{N}\sum_{i=1}^{N}\sum_{\substack{k=1\\ k\neq i}}^{N} v_{ij}v_{kj} + \frac{C}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}(N - v_{ij}) \qquad (1.1)$$

A, B, C are positive parameters selected by simulation to ensure convergence of the network. The first term in Equation 1.1 is minimized when a solution has at most a single non-zero element per row and ensures that at most one cell is chosen per input. Likewise, the second term is minimized when a solution has at most a single non-zero element per column and ensures that at most one cell is chosen per output. The third term is minimized when the number of non-zero elements in the solution is maximized.

The Hopfield neural network will usually, but not always, converge on a maximum size match. Occasionally, the network will find a suboptimal match, settling on a local minimum of the energy function. Our results from a simulation of [2] suggest that for a 16x16 switch the match never differs from the maximum size match by more than one connection and that the algorithm converges rapidly. We will call this algorithm *neural*. Results from the simulation of an almost identical scheme, designed in 2µm CMOS, reported a maximum convergence time of 200ns when N=8 [42]. The main problem with this approach is that it is analog, requiring careful design of amplifiers and RC circuits to ensure that the network will converge and that it will not favor some connections over others. However, although we shall not consider this method further in this thesis, we believe that this method is promising for prioritized and multicast traffic.

3.3.3 Scheduling into the Future

Several schemes have been proposed in which the time that a cell will be transmitted across the switch is decided when the cell arrives [1], [24], [35], [37], [38], [39]. We will describe two of these schemes, which are representative of the others.

The first scheme described by Obara in [37], consists of two phases: request and arbitration. We will call this scheme *Future 0*. The scheduler for output j consists of a counter, T_j representing the next time in the future that this output is not scheduled. When the output receives a request at time t, it returns the current value T_j to the requesting input and increments T_j by one. This ensures that the output is reserved at time T_j for the input. The input buffers the cell in an ordered list of departure times, tagging the cell for departure at time T_j. However, the input may have already received a value $T_k = T_j$, $j \neq k$, from some other output k at some time $t' \leq t$. In this case, the input must attempt to schedule this cell again in the next cell time. The advantage of this scheme is that the implementation complexity of the output scheduler is low, requiring only a counter that can be incremented by up to N per cell time. As described in [37], it is straightforward to pipeline this scheme for very high-bandwidth or large switches.

But even for Bernoulli i.i.d. arrivals with destinations uniformly distributed over outputs, this scheme achieves a throughput of just 65%, only slightly higher than for FIFO queueing. This is

because under high load, many reservations made by the output schedulers are not used by any input.

In an attempt to improve the throughput of this scheme, the authors in [24], propose a second scheme which we will call *Future 1*. An enhancement of *Future 0*, this scheme returns unusable reservation times to the outputs for recycling. Each output maintains a list of recycled time slots. When it receives a request, an output first considers its list of recycled time slots; if there is a time slot on the list that has not been previously granted to the requesting input, the slot is returned. If there is no suitable slot time on the list, the output returns the value of a counter T_j as before and increments the counter by one.

Under simulation, the authors find a dramatic increase in throughput; with Bernoulli i.i.d. arrivals, a throughput in excess of 95% can be achieved even if the recycling list is limited to just one cell. But the scheme is difficult to implement in hardware, requiring counters and lists that can be accessed by up to N requesting inputs in parallel.

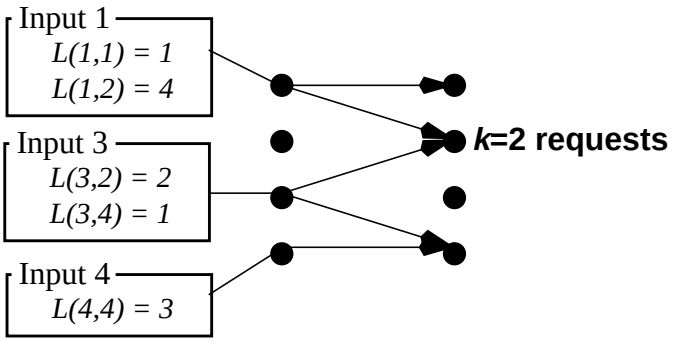
3.3.4 Parallel Iterative Matching

Parallel Iterative Matching (PIM) was developed by DEC Systems Research Center for the 16-port, 1Gbps AN2 switch [3]. Because it forms the basis of the novel algorithms described later, we will describe the scheme in detail and consider some of its performance characteristics.

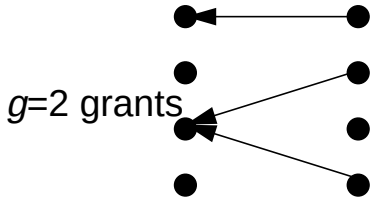
PIM uses *randomness* to avoid starvation, and to reduce the number of iterations needed to converge on a maximal matching. PIM attempts to quickly converge on a conflict-free match in multiple iterations, where each iteration consists of three steps. All inputs and outputs are initially unmatched and only those inputs and outputs not matched at the end of one iteration are eligible for matching in the next. The three steps of each iteration operate in parallel on each output and input and are shown in Figure 1.8. The steps are:

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

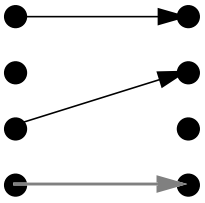
Step 2. *Grant*. If an unmatched output receives any requests, it grants to one by *randomly* selecting a request uniformly over all requests.



a) Step 1: *Request*. Each input makes a request to each output for which it has a cell. This is shown here as a graph with all weights, $w_{i,j} = 1$.



b) Step 2: *Grant*. Each output selects an input uniformly among those that requested it. In this example, inputs 1 and 3 both requested output 2. Output 2 chose to grant to input 3.



c) Step 3: *Accept*. Each input selects an output uniformly among those that granted to it. In this example, outputs 2 and 4 both granted to input 3. Input 3 chose to accept output 2.

Figure 1.8 An example of the three steps that make up one iteration of the PIM scheduling algorithm [3]. In this example, the first iteration does not match input 4 to output 4, even though it does not conflict with other connections. This connection would be made in the second iteration.

Step 3. *Accept*. If an input receives a grant, it accepts one by selecting an output among those that granted to this output.

By considering only unmatched inputs and outputs, each iteration only considers connections not made by earlier iterations.

Note that in step (2) above the independent output schedulers *randomly* select a request among contending requests. This has three effects: first the authors in [3] show that each iteration will match or eliminate on average at least $\frac{3}{4}$ of the remaining possible connections and thus the algorithm will converge to a maximal match in $O(\log N)$ iterations. Second, it ensures that all requests will eventually be granted. As a result, no input queue is starved. Third, it means that no memory or state is used to keep track of how recently a connection was made in the past. At the beginning

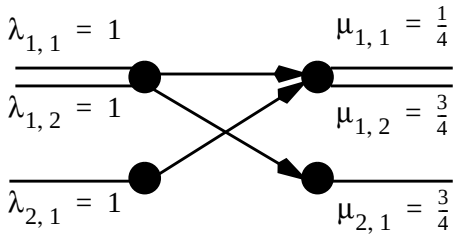


Figure 1.9 Example of unfairness for PIM under heavy, inadmissible load with more than one iterations.

of each cell time, the match begins over, independently of the matches that were made in previous cell times. Not only does this simplify our understanding of the algorithm, but it also makes analysis of the performance straightforward: there is no time-varying state to consider, except for the occupancy of the input queues.

But using randomness comes with its problems. First, it is difficult and expensive to implement at high speed: each scheduler must make a random selection among the members of a varying set. Second, for unsustainable traffic it can lead to unfairness between connections. An extreme example of unfairness for a 2x2 switch under an inadmissible load is shown in Figure 1.9. We will see examples later for which PIM and some other algorithms are unfair for admissible but unsustainable traffic. Finally, PIM does not perform well for a single iteration: it limits the throughput to just 63%, only slightly higher than for a FIFO switch. This is because the probability that an input will remain ungranted is $\left(\frac{N-1}{N}\right)^N$, hence as N increases, the throughput tends to $1-\frac{1}{e} \approx 63\%$. Although the algorithm will often converge to a good match after several iterations, the time to converge may affect the rate at which the switch can operate. We would prefer an algorithm that performs well with just a single iteration.

3.4 Simple Comparison of Previous Techniques

We conclude this chapter with a simple comparison of the performance under simulation[1] for the algorithms described above. We present results for each algorithm when the arrival process

1. All of the simulation results presented in this there were obtained using a slotted-time ATM simulator, written in C.

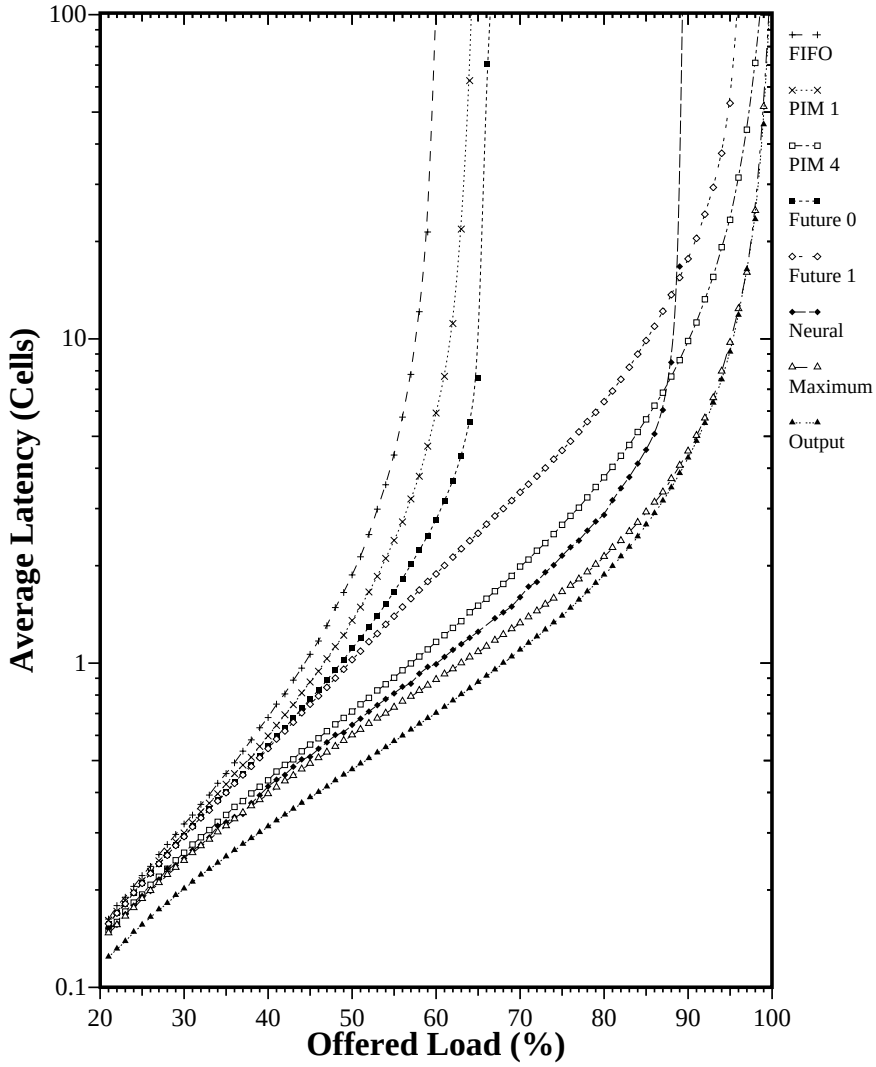


Figure 1.10 Comparison of latency as a function of offered load for several scheduling algorithms, using simulation. 16x16 switch, arrivals at each input are Bernoulli i.i.d. trials for each cell time. Cell destinations are uniformly distributed over all outputs. All arrival processes are independent.

$A_i(t)$ at each input consists of independent Bernoulli trials. Figure 1.10 indicates the latency as a function of offered load for each algorithm as well as for FIFO and pure output queueing.

The worst performance is given by FIFO queueing, with the input queues becoming unbounded for an offered load greater than 60%[1]. At the other extreme, output queueing represents the best performance and is stable for an offered load arbitrarily close to 100%.

1. As shown in [23], the throughput tends to 58% from above, as N tends to infinity.

Among the algorithms that attempt to achieve a maximum size match, the highest throughput is achieved unsurprisingly by the *maxsize*[1] algorithm. It is interesting to note that under high offered load, the performance of *maxsize* is indistinguishable from output queueing. This is because the input-queues are almost invariably occupied, resulting in a perfect match between inputs and outputs on every iteration. At the other extreme, PIM 1 (PIM with a single iteration) performs poorly, as expected. But with just four iterations, PIM 4 is a significant improvement remaining stable with an offered load in excess of 95%.

Future 0 performs slightly better than FIFO queueing saturating at just 65%, while the recycling of *Future 1* (with a list size of just 1) enables it to sustain an offered load in excess of 95%.

4 Outline of Thesis

Now that we have described the main features and limitations of existing scheduling algorithms we will, in the next three chapters, present several novel scheduling algorithms that we have devised. It is the objective of each algorithm to match the set of inputs of an input-queued switch to the set of outputs more efficiently, fairly and quickly than existing techniques.

Chapter 2 presents the simplest and fastest of these algorithms: SLIP. The SLIP algorithm is similar to PIM, but uses rotating priority (“round-robin”) arbitration to schedule each active input and output in turn. The main characteristic of SLIP is its simplicity: it is readily implemented in hardware and can operate at high speed. For uniform i.i.d. Bernoulli arrivals, SLIP has the appealing property that it is stable for any admissible load. We explain how this property arises from the tendency of the arbiters to *desynchronize* with respect to each other, and present some analytical results to model this behavior. SLIP, however, is not stable for all admissible arrival processes. Surprisingly, we also find that its behavior is not always monotonic: under specific conditions, adding more traffic can actually make the algorithm operate more efficiently. We examine this at length, presenting an approximate analytical model to describe this behavior. We present numerous simulation results, indicating how SLIP’s performance varies as a function of switch size and

1. *maxsize* was implemented using a randomized version of the $O(N^3)$ augmenting path algorithm [41].

traffic “burstiness”. Finally, we argue that a SLIP scheduler for a 32x32 switch can be readily implemented at high speed on a single VLSI chip with current technology.

Chapter 3 presents an iterative version of SLIP. Called *i*-SLIP, this algorithm attempts in each iteration to add connections not made by earlier iterations. The resulting match converges on a maximal match — the largest achievable match without rearranging connections. We find that the performance of *i*-SLIP increases significantly with the number of iterations, but only up to a point. Beyond $\log_2 N$ iterations, there is on average negligible improvement in performance. To avoid starvation careful attention must be paid to the way that the pointers are updated in *i*-SLIP and so we examine several variations of the algorithm, all designed to prevent starvation. Finally, we show that although it has a longer running time, an *i*-SLIP scheduler is little more complex than a single-iteration SLIP scheduler.

We conclude in Chapter 4 by describing algorithms that consider more information per queue, for example the occupancy of the queue, or the waiting time of queued cells. These algorithms find the maximum or maximal *weight* matching. Each algorithm gives preference to queues with a larger occupancy or to cells that have been waiting longest. We find these algorithms to be stable over a wider range of traffic loads. In particular, we describe two maximum weight match algorithms, *longest queue first* (LQF) and *oldest cell first* (OCF) and consider their performance. We prove that the LQF algorithm is stable for all admissible i.i.d. arrival, and conjecture that both algorithms, although too complex to implement in hardware, are stable under all admissible, ergodic arrival processes. We consider two implementable, iterative algorithms *i*-LQF and *i*-OCF which, with sufficient iterations, converge on a maximal weight matching. Implementations of both algorithms are presented. Finally, we present two interesting implementations of the Gale-Shapley algorithm, designed to solve the *stable marriage problem*.

CHAPTER 2

The SLIP Algorithm with a Single Iteration

1 Introduction

In this chapter we introduce, describe and evaluate the SLIP algorithm — a novel algorithm for scheduling cells in input-queued switches. This chapter concentrates on the behavior of SLIP with just a single iteration per cell time. In the next chapter we consider SLIP with multiple iterations.

The SLIP algorithm uses rotating priority ("round-robin") arbitration to schedule each active input and output in turn. The main characteristic of SLIP is its simplicity: it is readily implemented in hardware and can operate at high speed.

Before describing SLIP, we begin this chapter with a description of the basic round-robin matching (RRM) algorithm. We show that RRM performs poorly and demonstrate this with some examples. In Section 3 we introduce the SLIP algorithm as a variation of RRM. We show that the performance of SLIP for uniform traffic is surprisingly good; in fact, for uniform i.i.d. Bernoulli arrivals, SLIP with a single iteration is stable for any admissible load. This is the result of a phenomenon that we encounter repeatedly in this chapter: the arbiters in SLIP have a tendency to *desynchronize* with respect to one another.

As was observed for the *maxsize* algorithm in Chapter 1, SLIP can become unstable for admissible non-uniform traffic. In Section 5 we illustrate this with a 2x2 switch. For non-uniform i.i.d. Bernoulli arrivals we find offered loads for which SLIP performs *worse* than the *maxsize* algorithm and offered loads for which SLIP performs *better.* We examine in detail a region of operation in which SLIP behaves non-monotonically: increasing offered load can actually decrease the average queueing delay. We develop an analytical model describing this behavior, based on a simplified version of the switch. We expand this model in Section 5.4 to analyze the delay performance of a 2x2 SLIP switch.

In Section 6 we propose some variations on the basic SLIP algorithm, suitable for a number of different applications. Finally, in Section 7 we describe the implementation of a centralized SLIP scheduler, arguing that with current technology it is feasible to implement a 32x32 port scheduler on a single chip.

2 Basic Round-Robin Matching Algorithm

The basic round-robin (RRM) algorithm is designed to overcome two problems in PIM: *complexity* and *unfairness*. Implemented as priority encoders, the round-robin arbiters are much simpler and can perform faster than random arbiters. The rotating priority aids the algorithm in assigning bandwidth equally and more fairly among requesting connections.

The RRM algorithm, like PIM, consists of three steps. But rather than arbitrate *randomly*, the input and output arbiters for RRM make their selection according to a deterministic round-robin schedule. As shown in Figure 2.1, for an NxN switch each round-robin schedule contains N ordered elements. The three steps of arbitration are:

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

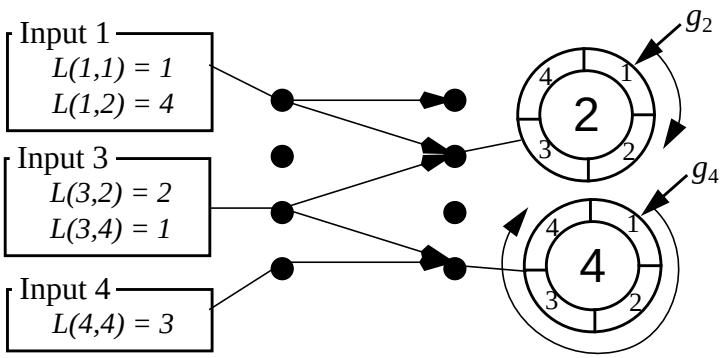
Step 2. *Grant*. If an output receives any requests, it chooses the one that appears next in a fixed, round-robin schedule starting from the highest priority element. The output notifies each input whether or not its request was granted. The pointer g_i to the highest priority element of the round-robin schedule is incremented (modulo N) to one location beyond the granted input.

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

2.1 Performance of RRM for Bernoulli Arrivals

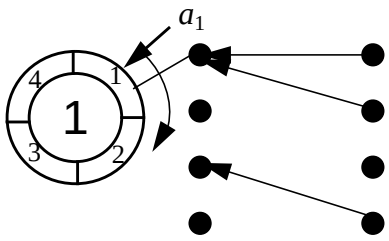
As an introduction to the performance of the RRM algorithm, Figure 2.2 shows the average delay as a function of offered load for uniform i.i.d. Bernoulli arrivals. For an offered load of just 63% the round-robin algorithm becomes unstable. This is similar to but worse than the PIM algorithm with a single iteration.

The reason for the poor performance of RRM lies in the rules for updating the pointers at the output arbiters. We illustrate this with an example, shown in Figure 2.3. Both inputs 1 and 2 are under heavy load and receive a new cell for both outputs during every cell time. But because the output schedulers move in lock-step, only one input is served during each cell time. The sequence of requests, grants, and accepts for four consecutive cell times are shown in Figure 2.4. Note that

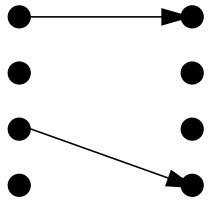


a) Step 1: *Request*. Each input makes a request to each output for which it has a cell.

Step 2: *Grant*. Each output selects the next requesting input at or after the pointer in the round-robin schedule. Arbiters are shown here for outputs 2 and 4. Inputs 1 and 3 both requested output 2. Since $g_2 = 1$ output 2 grants to input 1. g_2 and g_4 are updated to favor the input after the one that is granted.



b) Step 3: *Accept*. Each input selects at most one output. The arbiter for input 1 is shown. Since $a_1 = 1$ input 1 accepts output 1. a_1 is updated to point to output 2.



c) When the arbitration has completed, a matching of size two has been found. Note that this is less than the maximum sized matching of three.

FIGURE 2.1 Example of the three steps of the RRM matching algorithm.

the grant pointers change in lock-step: in cell time 1 both point to input 1 and during cell time 2 both point to input 2 *etc*. This synchronization phenomenon leads to a maximum throughput of just 50%.

As an example of the effect of synchronization under a random arrival pattern, Figure 2.5 shows the number of synchronized output arbiters as a function of offered load for a 16x16 switch with i.i.d Bernoulli arrivals. The graph plots the number of non-unique g_i's, i.e. the number of out-

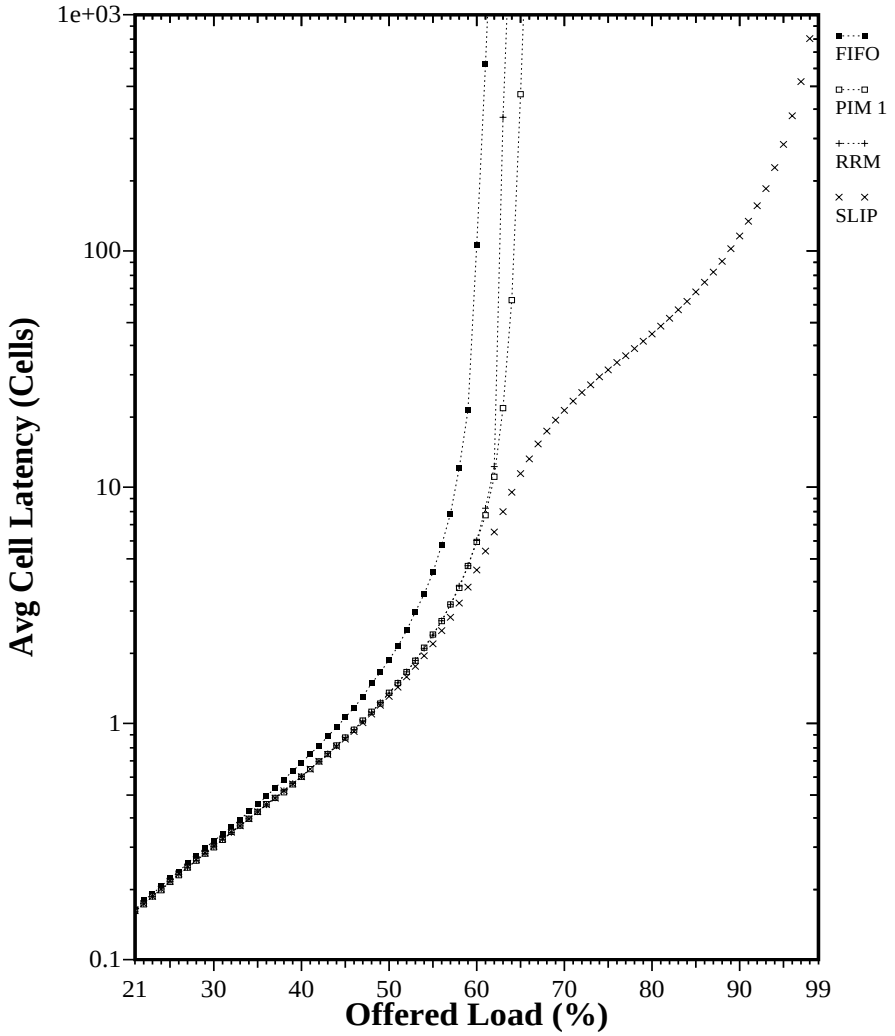


FIGURE 2.2 Performance of RRM and SLIP compared with PIM for i.i.d Bernoulli arrivals with destinations uniformly distributed over all outputs. Results obtained using simulation for a 16x16 switch. The graph shows the average delay per cell, measured in cell times, between arriving at the input buffers and departing from the switch.

put arbiters that clash with another arbiter. Under low offered load cells arriving for output j will find g_j in a random position, equally likely to grant to any input. The probability that $g_j \neq g_k$ for all $k \neq j$ is $\left(\frac{N-1}{N}\right)^{N-1}$ which for N=16 implies that the expected number of arbiters with the same highest-priority value is 9.9. This agrees well with the simulation result for RRM in Figure 2.5. As the offered load increases, synchronized output arbiters tend to move in lock-step and the degree of synchronization changes only slightly.

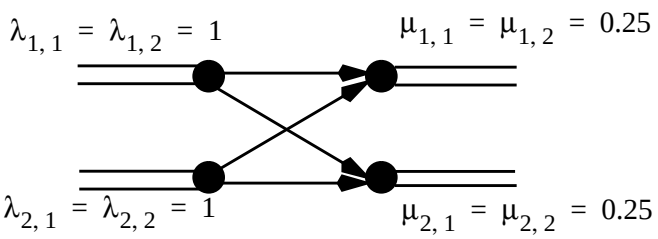


FIGURE 2.3 2x2 switch with RRM algorithm under heavy load. Synchronization of output arbiters leads to a throughput of just 50%.

3 The SLIP Algorithm

The SLIP algorithm is a variation on RRM designed to reduce the synchronization of the output arbiters. SLIP achieves this by not moving the grant pointers unless the grant is accepted leading to a desynchronization of the arbiters under high load. SLIP is identical to RRM except for a condition placed on updating the grant pointers. The *Grant* step of RRM is changed to:

> **Step 2.** *Grant*. If an output receives any requests, it chooses the one that appears next in a fixed, round-robin schedule starting from the highest priority element. The output notifies each input whether or not its request was granted. *The pointer g_i to the highest priority element 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.*

This small change to the algorithm leads to the following properties of SLIP:

> **Property 1.** Lowest priority is given to the most recently made connection. This is because when the arbiters move their pointers, the most recently granted (accepted) input (output) becomes the lowest priority at that output (input). If input *i* successfully connects to output *j*, both a_i and g_j are updated and the connection from input *i* to output *j* becomes the lowest priority connection in the next cell time.
>
> **Property 2.** No connection is starved. This is because an input will continue to request an output until it is successful. The output will serve at most N-1 other inputs first, waiting at most N cell times to be accepted by each input. Therefore, a requesting input is always served in less than N^2 cell times.
>
> **Property 3.** Under heavy load, all queues with a common output have the same throughput. This is a consequence of Property 2: the output pointer moves to each requesting input in a fixed order, thus providing each with the same throughput.

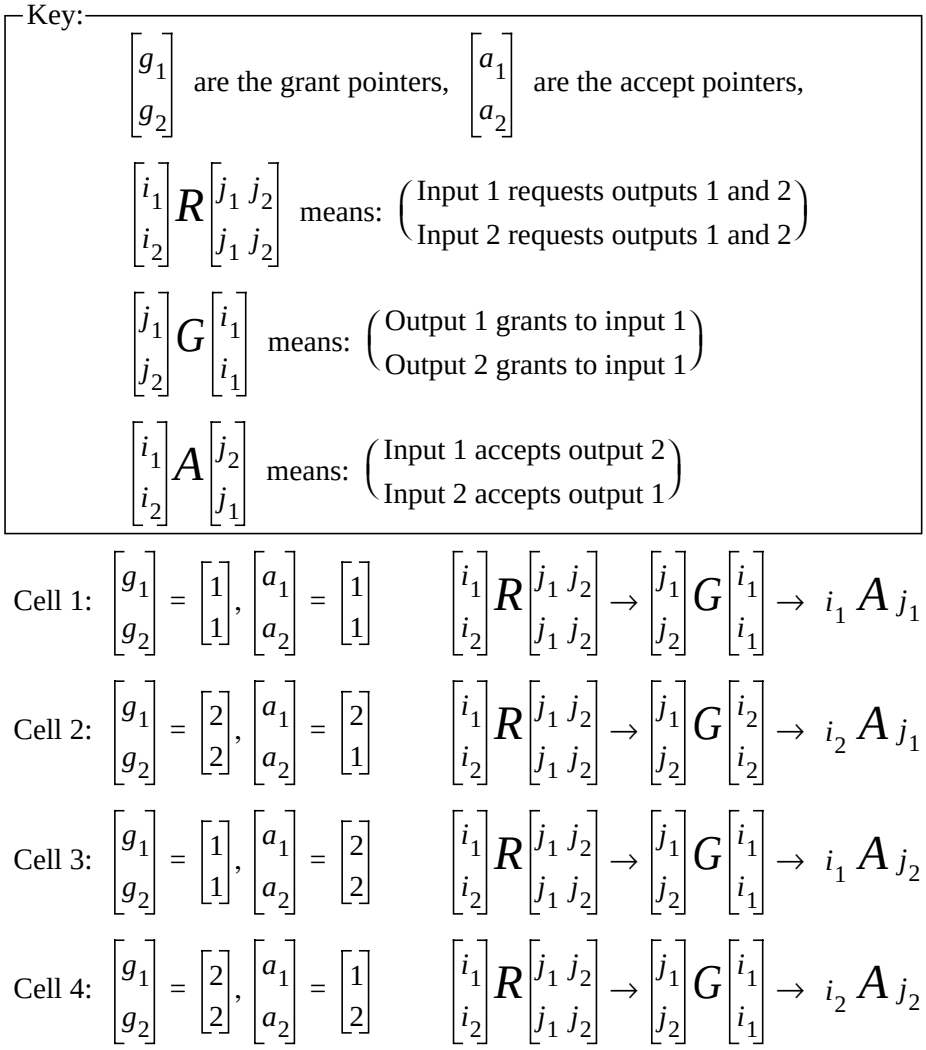


FIGURE 2.4 Illustration of low throughput for RRM caused by synchronization of output arbiters. Note that pointers $[g_i]$ stay synchronized, leading to a maximum throughput of just 50%.

But most importantly, this small change prevents the output arbiters from moving in lock-step leading to a dramatic improvement in performance.

4 Simulated Performance of SLIP

4.1 Bernoulli Traffic

To illustrate the improvement in performance of SLIP over RRM, Figure 2.2 shows the performance of the two algorithms under uniform i.i.d. Bernoulli arrivals. Under low load, SLIP's per-

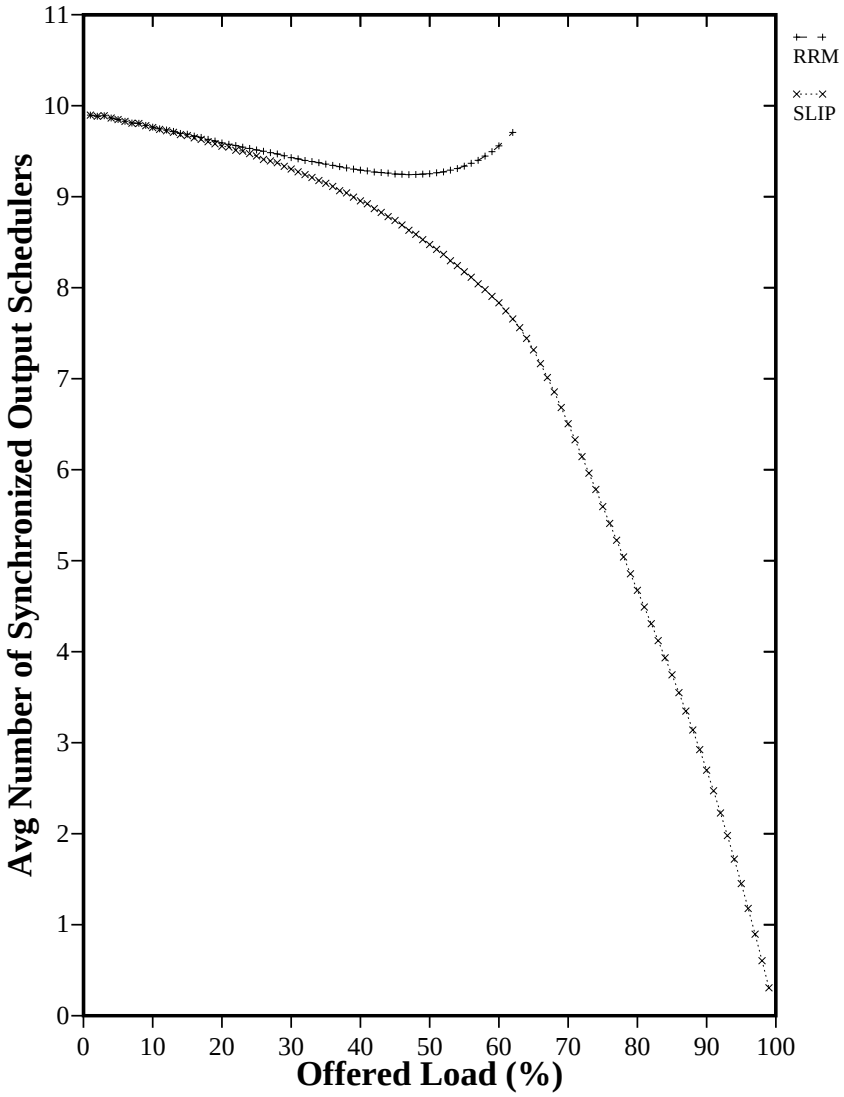


FIGURE 2.5 Synchronization of output arbiters for RRM and SLIP for i.i.d Bernoulli arrivals with destinations uniformly distributed over all outputs. Results obtained using simulation for a 16x16 switch.

formance is almost identical to RRM and FIFO; arriving cells usually find empty input queues, and on average there are only a small number of inputs requesting a given output. As the load increases, the number of synchronized arbiters decreases (see Figure 2.5), leading to a large sized match. In fact, under uniform 100% offered load the SLIP arbiters adapt to a time-division multiplexing scheme, providing a perfect match and 100% throughput.

$$\text{Cell 1: } \begin{bmatrix} g_1 \\ g_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \qquad \begin{bmatrix} i_1 \\ i_2 \end{bmatrix} R \begin{bmatrix} j_1 & j_2 \\ j_1 & j_2 \end{bmatrix} \rightarrow \begin{bmatrix} j_1 \\ j_2 \end{bmatrix} G \begin{bmatrix} i_1 \\ i_1 \end{bmatrix} \rightarrow \; i_1 \, A \, j_1$$

$$\text{Cell 2: } \begin{bmatrix} g_1 \\ g_2 \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}, \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \end{bmatrix} \qquad \begin{bmatrix} i_1 \\ i_2 \end{bmatrix} R \begin{bmatrix} j_1 & j_2 \\ j_1 & j_2 \end{bmatrix} \rightarrow \begin{bmatrix} j_1 \\ j_2 \end{bmatrix} G \begin{bmatrix} i_2 \\ i_1 \end{bmatrix} \rightarrow \begin{bmatrix} i_1 \\ i_2 \end{bmatrix} A \begin{bmatrix} j_2 \\ j_1 \end{bmatrix}$$

$$\text{Cell 3: } \begin{bmatrix} g_1 \\ g_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}, \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix} \qquad \begin{bmatrix} i_1 \\ i_2 \end{bmatrix} R \begin{bmatrix} j_1 & j_2 \\ j_1 & j_2 \end{bmatrix} \rightarrow \begin{bmatrix} j_1 \\ j_2 \end{bmatrix} G \begin{bmatrix} i_1 \\ i_2 \end{bmatrix} \rightarrow \begin{bmatrix} i_1 \\ i_2 \end{bmatrix} A \begin{bmatrix} j_1 \\ j_2 \end{bmatrix}$$

$$\text{Cell 4: } \begin{bmatrix} g_1 \\ g_2 \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}, \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \end{bmatrix} \qquad \begin{bmatrix} i_1 \\ i_2 \end{bmatrix} R \begin{bmatrix} j_1 & j_2 \\ j_1 & j_2 \end{bmatrix} \rightarrow \begin{bmatrix} j_1 \\ j_2 \end{bmatrix} G \begin{bmatrix} i_2 \\ i_1 \end{bmatrix} \rightarrow \begin{bmatrix} i_1 \\ i_2 \end{bmatrix} A \begin{bmatrix} j_2 \\ j_1 \end{bmatrix}$$

FIGURE 2.6 Illustration of 100% throughput for SLIP caused by desynchronization of output arbiters. Note that pointers [g_i] become desynchronized at the end of Cell 1 and stay desynchronized, leading to an alternating cycle of 2 cell times and a maximum throughput of 100%.

Figure 2.6 is an example for a 2x2 switch showing how under heavy traffic the arbiters adapt to an efficient time-division multiplexing schedule.

4.2 “Bursty” Traffic

Real network traffic is highly correlated from cell to cell [32] and so in practice, cells tend to arrive in bursts, corresponding perhaps to a packet that has been segmented or a packetized video frame. Many ways of modeling bursts in network traffic have been proposed [16], [21], [4], [32]. Recently, Leland *et al.* [32] have demonstrated that measured network traffic is bursty at every level making it important to understand the performance of switches in the presence of bursty traffic.

We illustrate the effect of burstiness on SLIP using an on-off arrival process modulated by a 2-state Markov-chain. The source alternately produces a burst of full cells (all with the same destination) followed by an idle period of empty cells. The bursts and idle periods contain a geometrically distributed number of cells.

Figure 2.7 shows the performance of SLIP under this arrival process for a 16x16 switch, comparing it with the performance under uniform i.i.d. Bernoulli arrivals. As we would expect, the

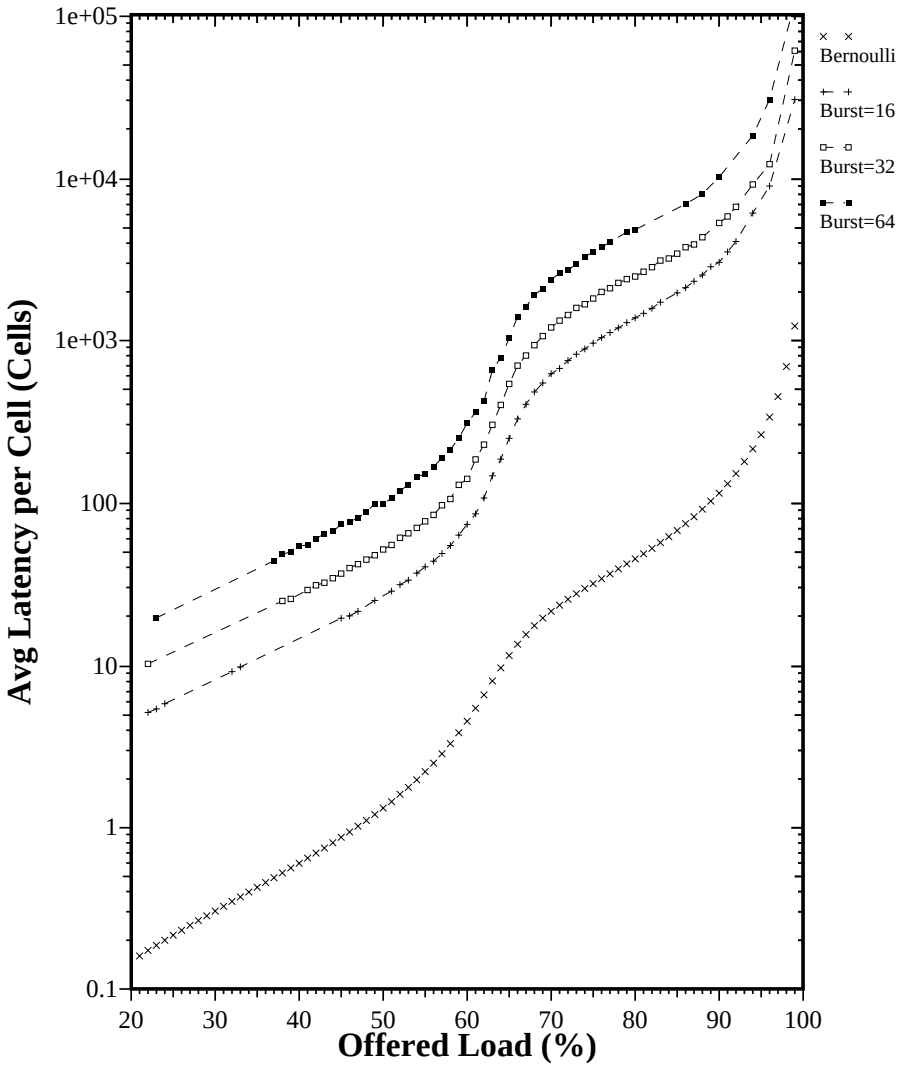


FIGURE 2.7 The performance of SLIP under 2-state Markov-modulated Bernoulli arrivals. All cells within a burst are sent to the same output. Destinations of bursts are uniformly distributed over all outputs.

increased burst size leads to a higher queueing delay. In fact, the average latency is *proportional* to the expected burst length.

Although not shown here, we have also compared the performance of SLIP with other algorithms for this traffic model. Our results suggest that for all the algorithms described in this thesis, the increase in average queueing delay for input-queued switches is approximately proportional to the expected burst length. In fact, the performance of the input-queued switch scheduling algorithms become more and more alike and can become similar to the performance of an output-

queued switch. This similarity indicates that the performance for bursty traffic is not heavily influenced by the queueing policy. Burstiness tends to concentrate the conflicts on outputs rather than inputs: each burst contains cells destined for the same output and each input will be dominated by a single burst at a time. As a result, the performance is limited by output contention.

4.3 As a Function of Switch Size

The *maxsize* algorithm described in Chapter 1 is known to have a running time of $O(N^{2.5})$ and the PIM algorithm is known to converge to a maximal match in a (serial) running time of O(NlogN)[1]. In the next chapter we will consider the improvement in performance of SLIP when we allow more iterations per cell time. But for a *single* iteration in which the running time is fixed, we can expect the performance to degrade as the number of ports is increased.

Figure 2.8 shows the average latency imposed by a SLIP scheduler as a function of offered load for switches with 4, 8, 16 and 32 ports. As expected, the performance degrades with the number of ports. But the performance degrades differently under low and heavy loads. For a fixed low offered load, the queueing delay converges to a constant value. However, for a fixed heavy offered load the increase in queueing delay is *proportional* to N.

The reason for these different characteristics under low and heavy load lies once again in the degree of synchronization of the arbiters. Under low load, arriving cells find the arbiters in random positions and SLIP performs in a similar manner to the single iteration version of PIM. The probability that the cell is scheduled to be transmitted immediately is proportional to the probability that no other cell is waiting to be routed to the same output. Ignoring the (small) queueing delay under low offered load, the number of contending cells for each output is approximately $\lambda\left(1-\left(\frac{N-1}{N}\right)^{N-1}\right)$ which for large N converges to $\lambda\left(1-\frac{1}{e}\right)$. Hence, for constant small λ, the queueing delay converges to a constant. Under heavy load, the algorithm serves each FIFO once every *N* cycles and the queues will behave similarly to an M/D/1 queue with arrival rates $\frac{\lambda}{N}$ and

1. PIM is designed to run on N parallel arbiters for which it has a running time O(logN). Its running time on a single arbiter is therefore O(NlogN).

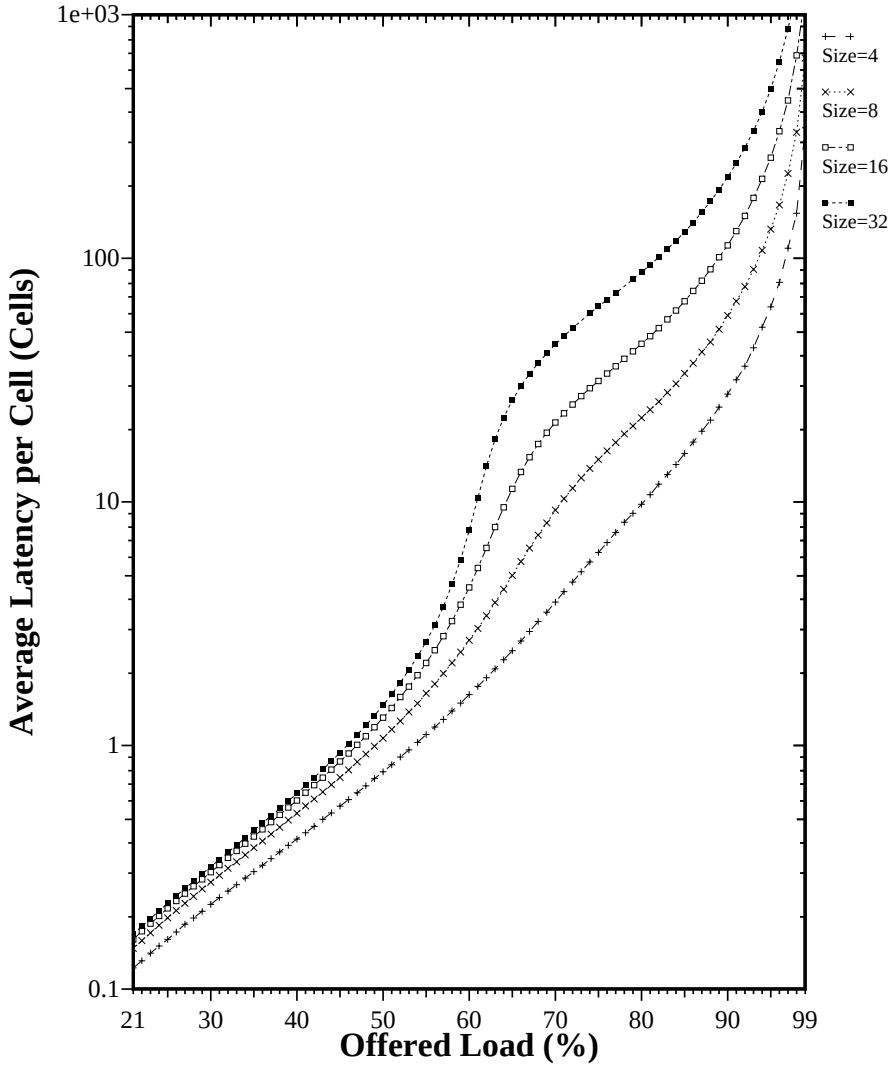


FIGURE 2.8 The performance of SLIP as function of switch size. Uniform i.i.d. Bernoulli arrivals.

deterministic service time N cell times. For an M/G/1 queue with random service times S, arrival rate λ and service rate μ the queueing delay is given by

$$d = \frac{\lambda E(S^2)}{2\left(1 - \frac{\lambda}{\mu}\right)}. \quad (1)$$

So, for the SLIP switch under a heavy load of Bernoulli arrivals the delay will be approximately

$$d = \frac{\lambda N}{2\,(1-\lambda)} \tag{2}$$

which is proportional to N.

4.4 Burst Reduction

In Section 4.2 we saw the not surprising result that burstiness increases queueing delay. In addition to the performance of a single switch for bursty traffic, it is important to consider the effect that the switch has on other switches downstream. Intuitively, if a switch decreases the average burst length of traffic that it forwards, then we can expect it to improve the performance of its downstream neighbor. We next examine the burst-reduction properties of SLIP.

There are many definitions of burstiness, for example the coefficient of variation [43], burstiness curves [28], maximum burst length [7], or effective bandwidth [31]. In this section, we use the same measure of burstiness that we used when generating traffic in Section 4.2: the average burst length. We define a burst of cells at the output of a switch as the number of consecutive cells that entered the switch at the same input.

SLIP is a deterministic algorithm, serving each connection in strict rotation. We therefore expect that bursts of cells at different inputs contending for the same output will become interleaved and the burstiness will be reduced. This is indeed the case, as shown in Figure 2.9. The graph shows the average burst length at the switch output as a function of offered load. Arrivals are on-off processes modulated by a 2-state Markov chain with average burst lengths of 16, 32 and 64 cells, as described in Section 4.2.

Our results indicate that SLIP reduces the average burst length, and will tend to be more burst-reducing as the offered load increases. This is because the probability of switching between multiple connections increases as the utilization increases. When the offered load is low, arriving bursts do not encounter output contention and the burst of cells is passed unmodified. As the load

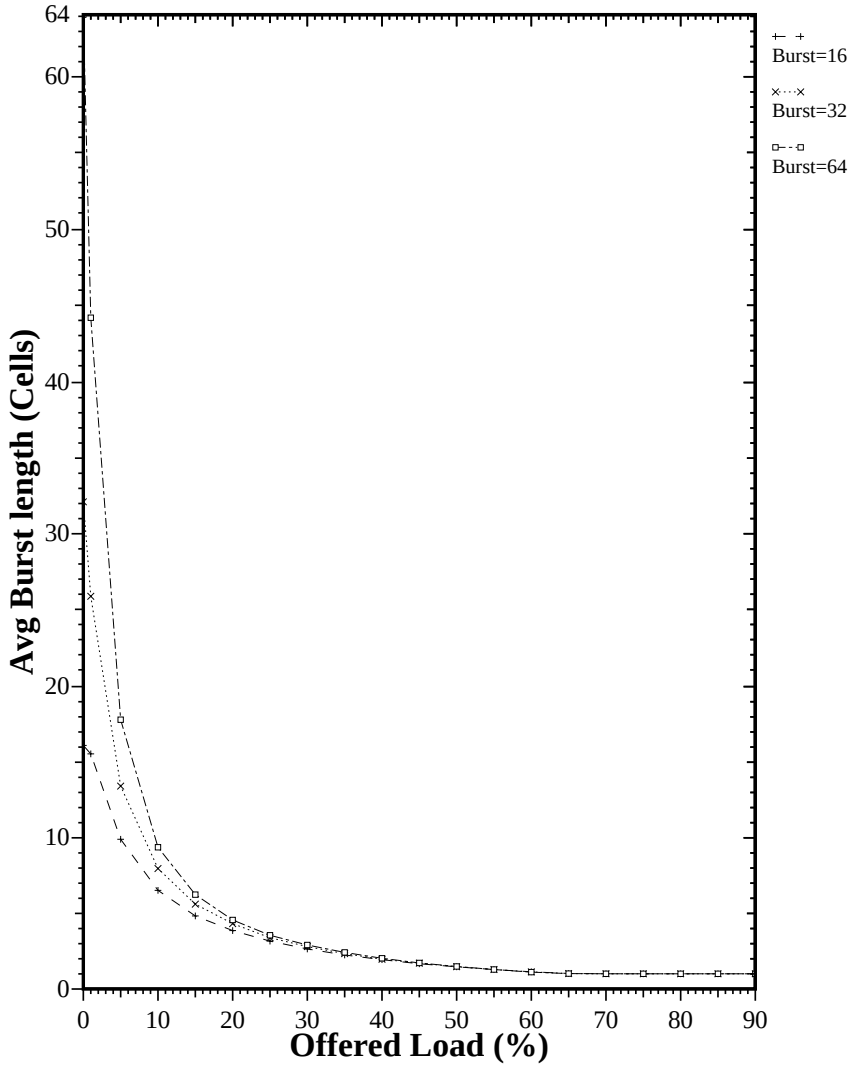


FIGURE 2.9 Average burst length at switch output as a function of offered load. The arrivals are on-off processes modulated by a 2-state DTMC. Results are for a 16x16 switch using the SLIP scheduling algorithm.

increases, the contention increases and bursts are interleaved at the output. In fact, if the offered load exceeds approximately 70%, the average burst length drops to exactly one cell. This indicates that the output arbiters have become desynchronized and are operating as time-division multiplexers, serving each input in turn.

5 Analysis of SLIP Performance

In general, it is difficult to analyze the performance of a SLIP switch, even for the simplest traffic models. Under uniform load and either very low or very high offered load we can readily approximate and understand the way in which SLIP operates. When arrivals are infrequent we can assume that the arbiters act independently and that arriving cells are successfully scheduled with very low delay. At the other extreme, when the switch becomes uniformly backlogged, we can see that desycnhronization will lead the arbiters to find an efficient time division multiplexing scheme and operate without contention. But when the traffic is non-uniform, or when the offered load is at neither extreme, the interaction between the arbiters becomes difficult to describe. The problem lies in the evolution and interdependence of the state of each arbiter and their dependence on arriving traffic.

5.1 Convergence to Time-Division Multiplexing Under Heavy Load

In Section 4.3 we argued that under heavy load, SLIP will behave similarly to an M/D/1 queue with arrival rates $\frac{\lambda}{N}$ and deterministic service time N cell times. So, under a heavy load of Bernoulli arrivals the delay will be approximated by Equation 2.

To see how close SLIP becomes to time-division multiplexing under heavy load, Figure 2.10 compares the average latency for both SLIP and an M/D/1 queue (Equation 2). Above an offered load of approximately 70%, SLIP behaves very similarly to the M/D/1 queue, but with a higher latency. This is because the service policy is not constant: when a queue changes between empty and non-empty, the scheduler must adapt to the new set of queues that require service. This adaption takes place over many cell times while the arbiters desynchronize again. During this time, the throughput will be worse than for the M/D/1 queue and the queue length will increase. This in turn will lead to an increased latency.

5.2 Desynchronization of Arbiters

We have argued that the performance of SLIP is dictated by the degree of synchronization of the output schedulers. In this section we present a simple model of synchronization for a stationary and sustainable uniform arrival process.

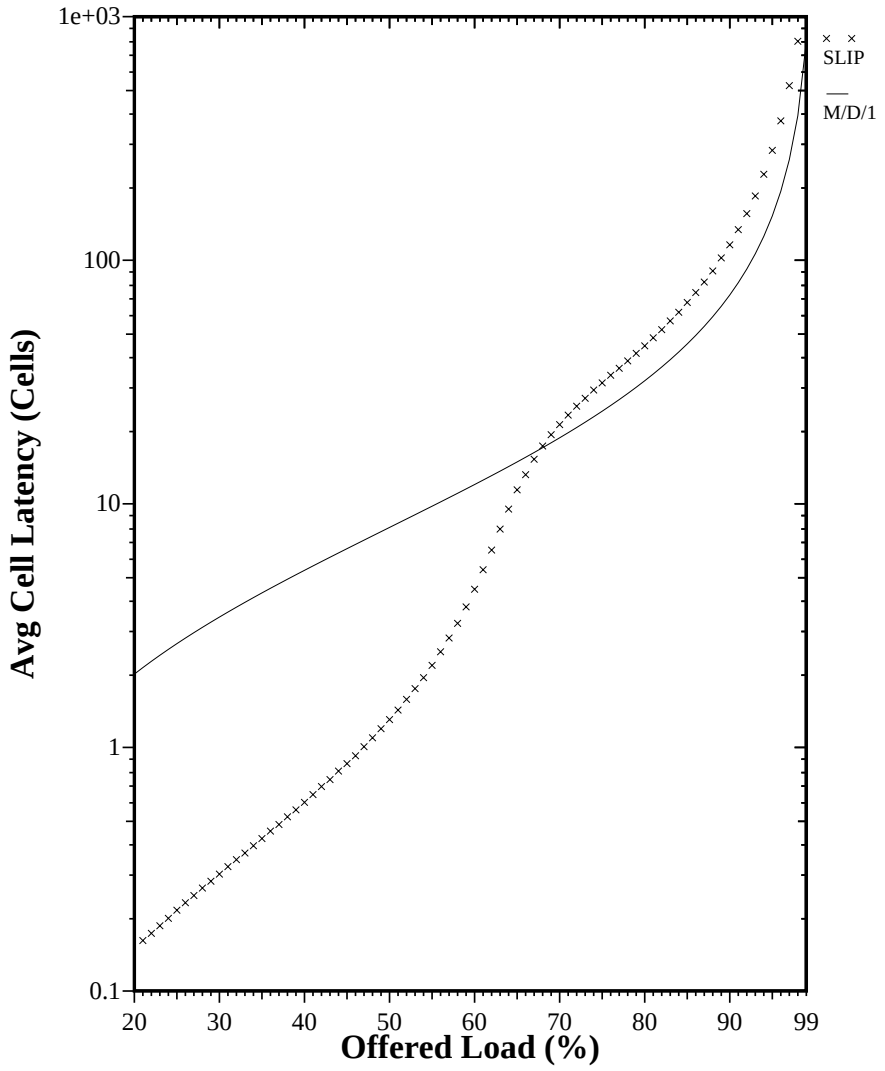


FIGURE 2.10 Comparison of average latency for the SLIP algorithm and an M/D/1 queue. The switch is 16x16 and, for the SLIP algorithm, arrivals are uniform i.i.d. Bernoulli arrivals.

In Appendix 1 we find an approximation for $\mathrm{E}[S(t)]$, the expected number of synchronized output schedulers at time *t*. The approximation is based on two assumptions:

1. Inputs that are unmatched at time *t* are uniformly distributed over all inputs.
2. The number of unmatched inputs at time *t* has zero variance.

This leads to the approximation

$$\mathrm{E}[S(t)] \approx N - \lambda N\left(\frac{\lambda N - 1}{\lambda N}\right)^{\lambda\bar{\lambda}N} - \bar{\lambda}^2 N\left(\frac{\bar{\lambda}N - 1}{\bar{\lambda}N}\right)^{\bar{\lambda}^2 N - 1} \tag{3}$$

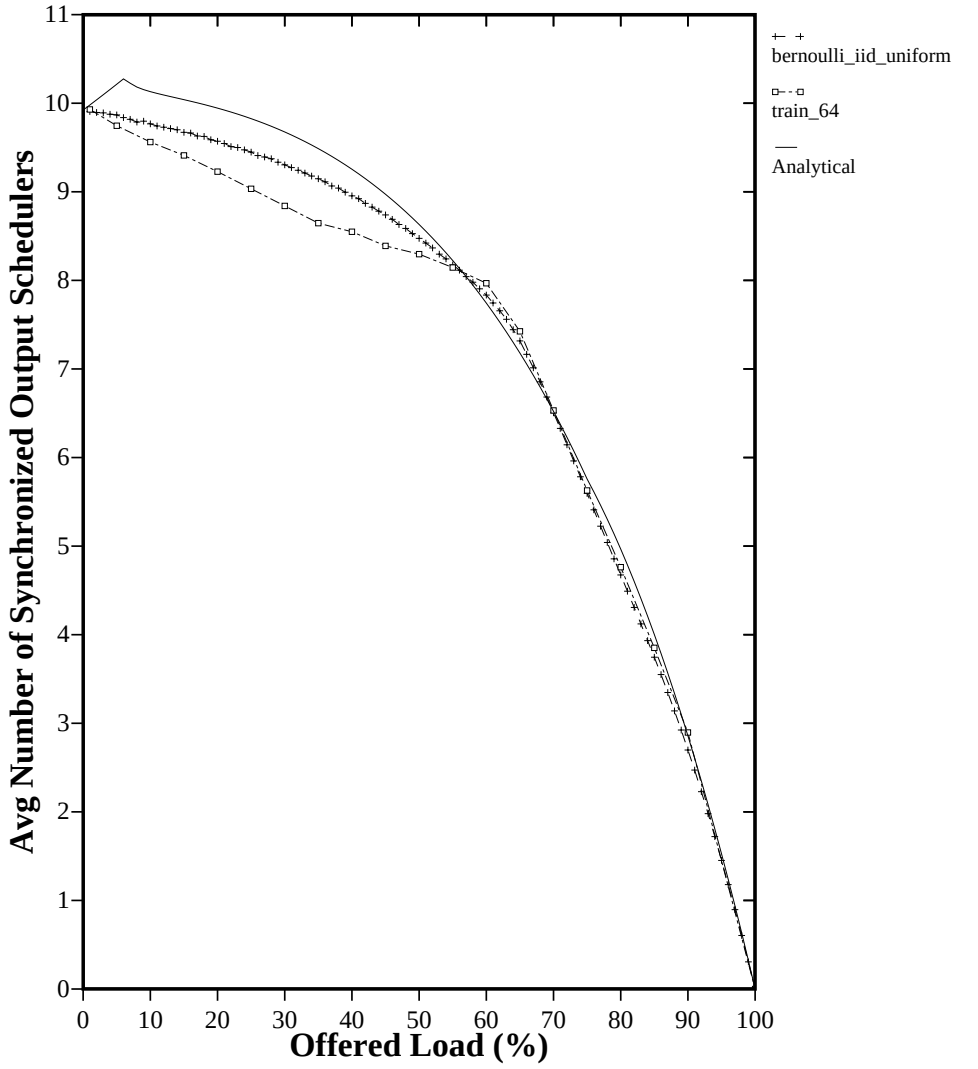


FIGURE 2.11 Comparison of analytical approximation and simulation results for the average number of synchronized output schedulers. Simulation results are for a 16x16 switch with i.i.d Bernoulli arrivals and an on-off process modulated by a 2-state Markov chain with an average burst length of 64 cells. The analytical approximation is shown in Equation 3.

where,

N = number of ports,

λ = arrival rate averaged over all inputs,

$\bar{\lambda} = (1 - \lambda)$.

This approximation is quite accurate over a wide range of uniform workloads. Figure 2.11 compares the approximation in Equation 3 with simulation results for both i.i.d. Bernoulli arrivals and for an on-off arrival process modulated by a 2-state Markov-chain (described in Section 4.2).

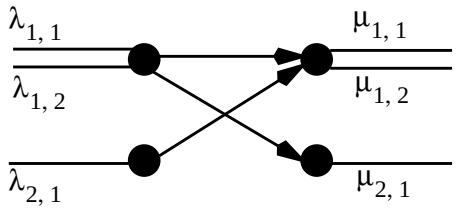


FIGURE 2.12 2x2 Switch with 3 active flows.

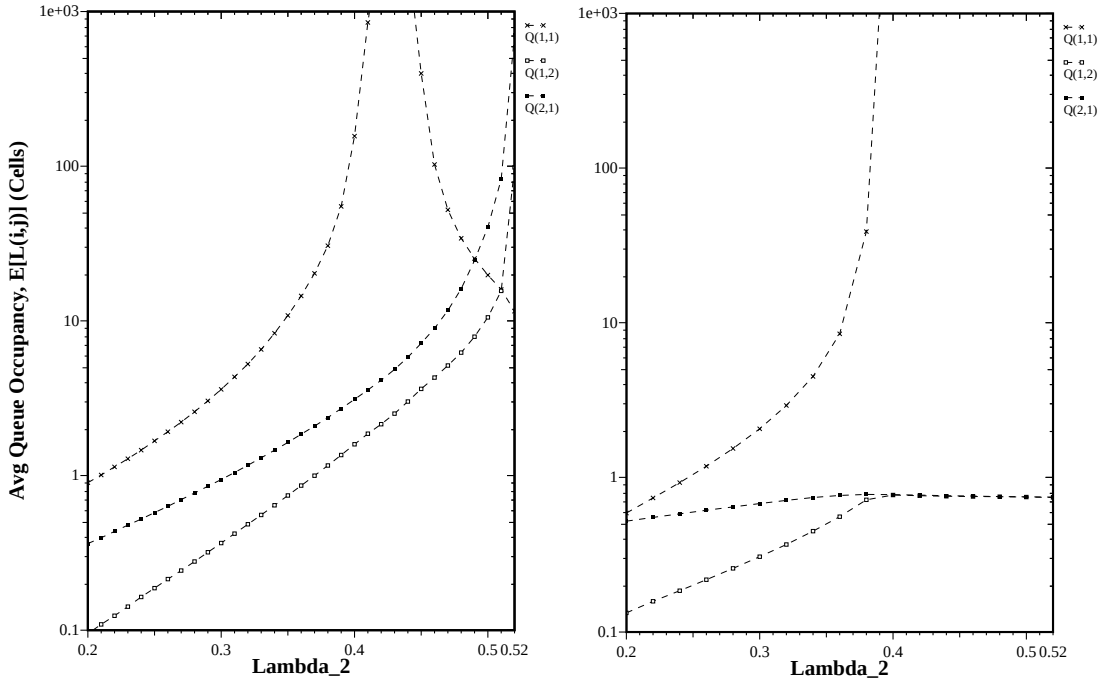
5.3 Stability of SLIP

Figure 2.2 shows that the SLIP algorithm is stable for all admissible uniform i.i.d. Bernoulli traffic. In practice, however, traffic tends to be concentrated among a small number of ports that have quite asymmetric transmit and receive behavior, making the traffic non-uniform. In this section we consider the stability of SLIP under non-uniform traffic.

In Chapter 1 we saw that a 2x2 switch can be unstable for the maximum sized matching algorithm for admissible i.i.d. Bernoulli arrivals, *when the traffic pattern is non-uniform*. SLIP operates efficiently by mimicking the behavior of the maximum matching algorithm under heavy load. It is therefore not surprising that a 2x2 switch using the SLIP algorithm can also be unstable under non-uniform traffic.

We illustrate the region of instability for SLIP using the 2x2 switch shown in Figure 2.12. With i.i.d. Bernoulli arrivals, we find that the SLIP algorithm is not only unstable for certain arrival rates, but also that its behavior is non-monotonic: increasing the arrival rate can actually *reduce* the expected occupancy of the input queues.

Figure 2.13(a) illustrates this surprising effect: fixing $\lambda_1 (= \lambda_{1,1}) = 0.48$ and varying $\lambda_2 (= \lambda_{1,2} = \lambda_{2,1})$ we see that SLIP becomes unstable in the region $0.41 \leq \lambda_2 \leq 0.44$, but becomes stable again for $0.44 < \lambda_2 < 1 - \lambda_1$. It is also interesting to note that the behavior of Q(1,2) and Q(2,1) is unaffected by Q(1,1), increasing monotonically even through the region of instability for Q(1,1).



a) SLIP. Note that Q(1,1) becomes unstable for $0.41 \le \lambda_2 \le 0.44$, but is stable again as traffic is increased.

b) *Maxsize* algorithm.

FIGURE 2.13 Example of instability for SLIP and maximum sized matching algorithms for 2x2 switch. Traffic pattern as shown in Figure 2.12, $\lambda_1 = 0.48$.

In contrast, *maxsize* behaves quite differently: as shown in Figure 2.13(b) Q(1,1) becomes unstable for all $0.4 \le \lambda_2 \le 1 - \lambda_1$.

The region in which SLIP behaves non-monotonically is small. Figure 2.14 compares the region of instability for both SLIP and *maxsize*. Whereas *maxsize* has a stable region over most of the region of admissible traffic bounded by $\lambda_1 + \lambda_2 = 0.88$, the region for SLIP is more complex. Over most of the region of admissible traffic, increasing λ_1 or λ_2 cannot change SLIP from unstable to stable. However, this is not the case for $0.48 \le \lambda_1 \le 0.51$, highlighted by the rectangle in Figure 2.14.

Before trying to model this behavior, let us consider the intuition to be drawn from Figure 2.14. First, the region of instability is not symmetric in λ_1 and λ_2: the switch is more susceptible to instability for small λ_2 when λ_1 is large than vice-versa. This is also true for *maxsize*. Both

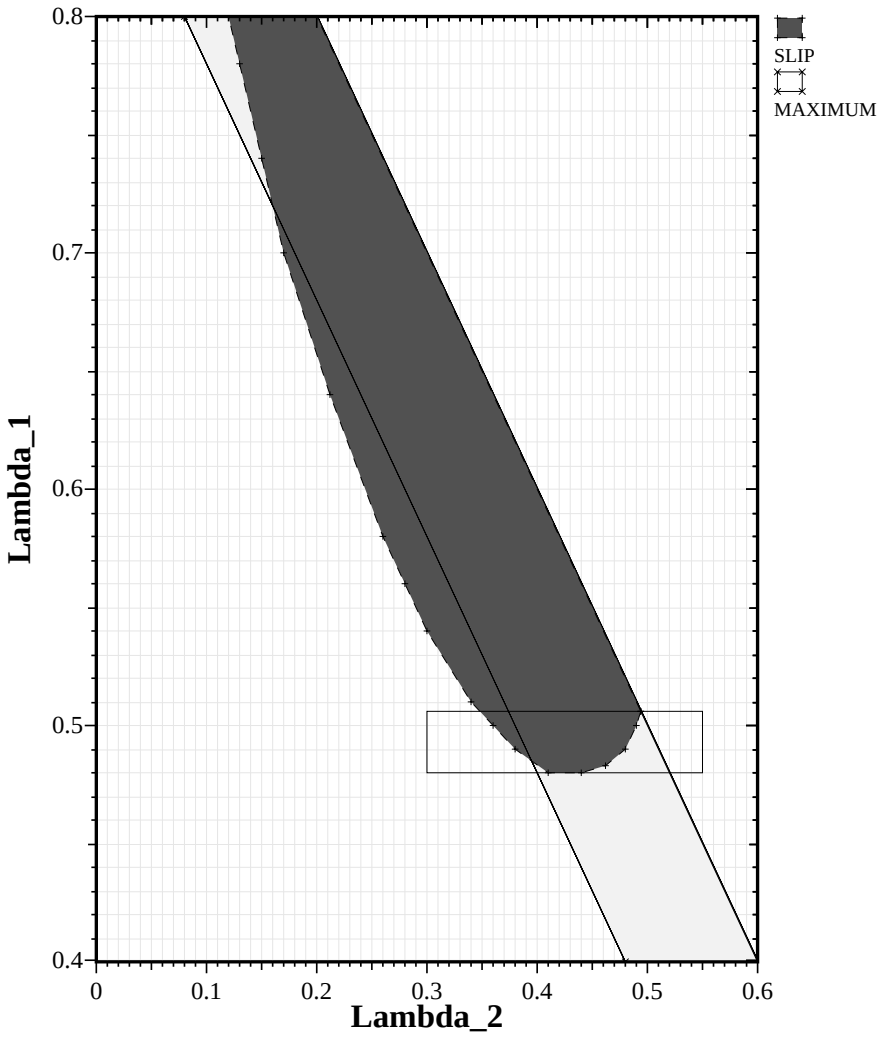


FIGURE 2.14 Region of instability for SLIP and *maxsize* for a 2x2 switch under i.i.d Bernoulli arrivals and the traffic pattern of Figure 2.12. In this example, $\text{Lambda\_1} = \lambda_{1,1}$, $\text{Lambda\_2} = \lambda_{1,2} = \lambda_{2,1}$ and $\text{Lambda\_1+Lambda\_2} < 1$. For each algorithm, the shaded area represents *unsustainable* traffic patterns.

algorithms favor large sized matches (*maxsize* does this statically, whereas SLIP does so over several cell times) and so will favor the “cross” traffic, which clears two cells simultaneously from both inputs in a cell time, over the “parallel” traffic that clears a cell from only Q(1,1). The second characteristic to be noted is that SLIP performs at its worst compared to *maxsize* when $0.2 < \lambda_1 < 0.4$ and $\lambda_1 + \lambda_2$ is close to 1. The reason for this is that the offered load is high, yet the input queues Q(1,2) and Q(2,1) receive preferential service over Q(1,1). Frequently, either Q(1,2)

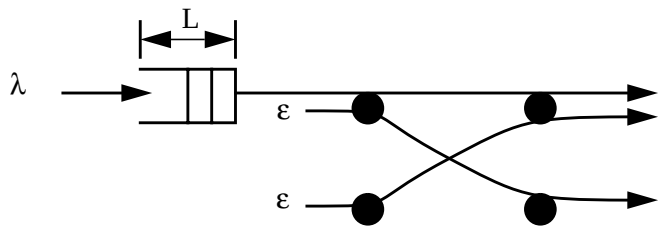


FIGURE 2.15 Simplified 2x2 switch with a single queue, Q(1,1).

or Q(2,1) will change between empty and non-empty, requiring SLIP to adapt to the new traffic pattern. This inhibits the tendency of the arbiters to desynchronize.

5.3.1 Drift Analysis of a 2x2 SLIP Switch: First Approximation

To try and understand the non-monotonic behavior of SLIP, we examine the more tractable, simplified switch with only one queue Q(1,1) shown in Figure 2.15. This switch behaves similarly to the 2x2 switch with 3 queues in Figure 2.12, except that cells arriving at input 1 and destined for output 2 are not queued. A cell arrives at the beginning of the time slot with probability ε; if the cell is not scheduled to be transmitted in the same cell time, it is discarded. Similarly for cells arriving at input 2 destined for output 1.

In Appendix 2 Section 1 we analyze this switch to determine values of λ and ε for which the switch is unstable. By considering the expected increase in L, the occupancy of Q(1,1), at each cell time, we find that the switch is unstable for

$$\lambda > \frac{1}{1 + 2\varepsilon + \varepsilon^2 - 2\varepsilon^3} \ . \tag{4}$$

This result is confirmed in Appendix 2 Section 2 where the distribution function for the occupancy of Q(1,1) is found using the matrix geometric method of Neuts [36].

Equation 4 is plotted in Figure 2.16 along with the admissibility constraint $\lambda + \varepsilon < 1$. The area between the curves is the region for which $\mathrm{E}[L(t)] \to \infty$. Comparing Figure 2.16 with the region of stability for the full 2x2 switch in Figure 2.14, we see that they are quite different.

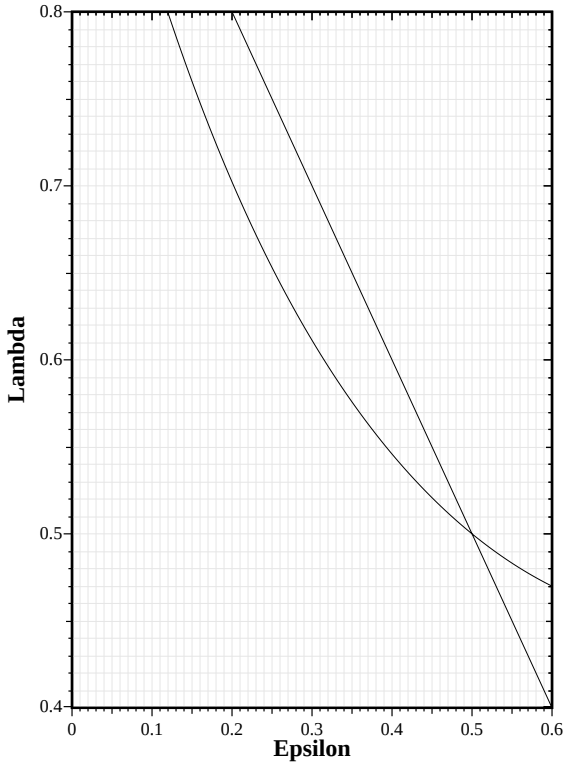


FIGURE 2.16 The area between the two curves is the region of instability for the switch in Figure 2.15.

Although the model captures the asymmetry between λ and ε, and the fact that the switch performs worst when λ is small and $\lambda + \varepsilon \approx 1$, it does *not* capture the non-monotonic behavior of SLIP. In fact, we should expect the behavior to be different: cells that arrive at one of the unbuffered inputs of our simplified switch can only affect the switch for a single cell time. Cells arriving at the full 2x2 switch of Figure 2.12 that are unscheduled when they first arrive will still be there in the next cell time, reducing the likelihood that Q(1,1) will be serviced.

We can substantially improve the accuracy of our model by estimating the number of cell times that an arriving cell will affect the scheduling algorithm and increase the arrival rate, ε to compensate.

Our claim is that the arrival rate in the approximate model should be *doubled*, i.e. $\varepsilon = 2\lambda_2$. Our argument in support of this is a heuristic one: when a cell arrives at an empty queue in the exact model, it is either successfully scheduled immediately or it is queued. If it is queued, the

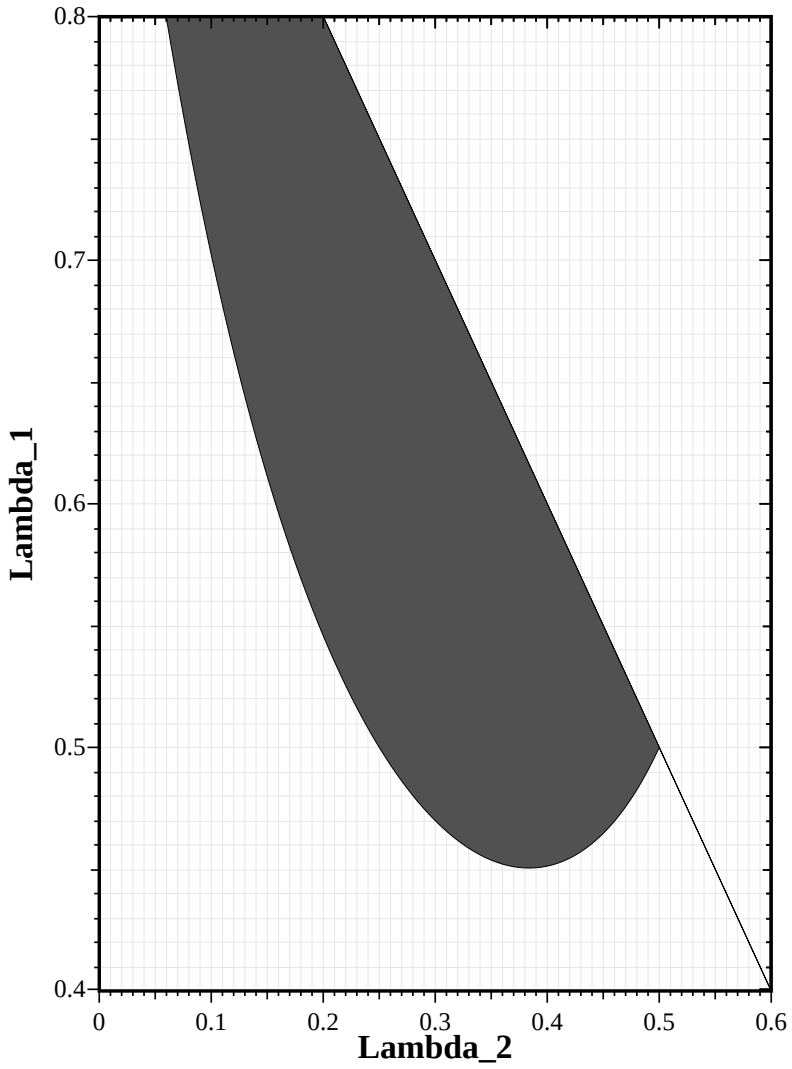


FIGURE 2.17 Region of instability for simplified 2x2 switch model, using the approximation $\lambda_2 \approx \frac{1}{2}\varepsilon$, $\lambda_1 \approx \lambda$.

SLIP scheduler must service this queue in the next cell time. Hence, the cell has affected the scheduler for two cell times.

With the approximation $\lambda_2 \approx \frac{1}{2}\varepsilon$, we obtain a more accurate model. Figure 2.17 shows the region of instability with this approximation, modeled in the same way as before. Comparing this with the region of instability in Figure 2.14 obtained using simulation, we see that the characteris-

tics are very similar. The approximate model captures the non-monotonic behavior of SLIP close to $\lambda_1 = 0.5$.

5.3.2 Drift Analysis of a 2x2 SLIP Switch: Second Approximation

In our first model we found that modeling the arrival process as unqueued i.i.d. Bernoulli arrivals was inaccurate. This was because arriving cells in the real switch are queued and affect the scheduler for multiple cell times. In this section we try and improve upon this approximation by modeling the arrival process more accurately.

In our second approximation, we model arrivals as an on-off process, modulated by a 2-state discrete-time Markov chain (DTMC). The DTMC is used to model the *busy* and *idle* cycles of input queues Q(1,2) and Q(2,1) in the real switch. When the DTMC is in the *busy* state, cells are arrive at rate 1, and when it is the *idle* state, cells arrive at rate 0. Using this model we attempt to capture the correlation between successive cell times.

In Appendix 2 Section 1.2, we analyze such a switch to determine values of λ_1 and λ_2 for which the switch is unstable. As before, by considering the expected increase in L, the occupancy of Q(1,1), at each cell time, we find an expression for the stable region of the switch. Unfortunately the stability expression is the ratio of two 10th degree polynomials in λ_1 and λ_2 and we have been unable to find a closed form expression for this region in the desired form $\lambda_1 > f(\lambda_2)$.

Instead, we find the stable region numerically, as shown in Figure 2.18 along with admissibility constraint $\lambda_1 + \lambda_2 < 1$. The approximate model captures the non-monotonic behavior of SLIP well. But although the *shape* of the stability region is accurate, its values are not. The exact position of the region is very sensitive to the expressions for the busy and idle cycles. Several of the poles of the 10th degree polynomials are close to the admissible region: moving these only slightly has a large affect on the position and rotation of the stability region.

5.4 Approximate Delay Model for 2x2 SLIP Switch

As described in Section 5.3.1, we can model the *simplified* switch in Figure 2.15 for i.i.d. Bernoulli arrivals as an infinite dimension DTMC. This can be solved using the matrix-geometric

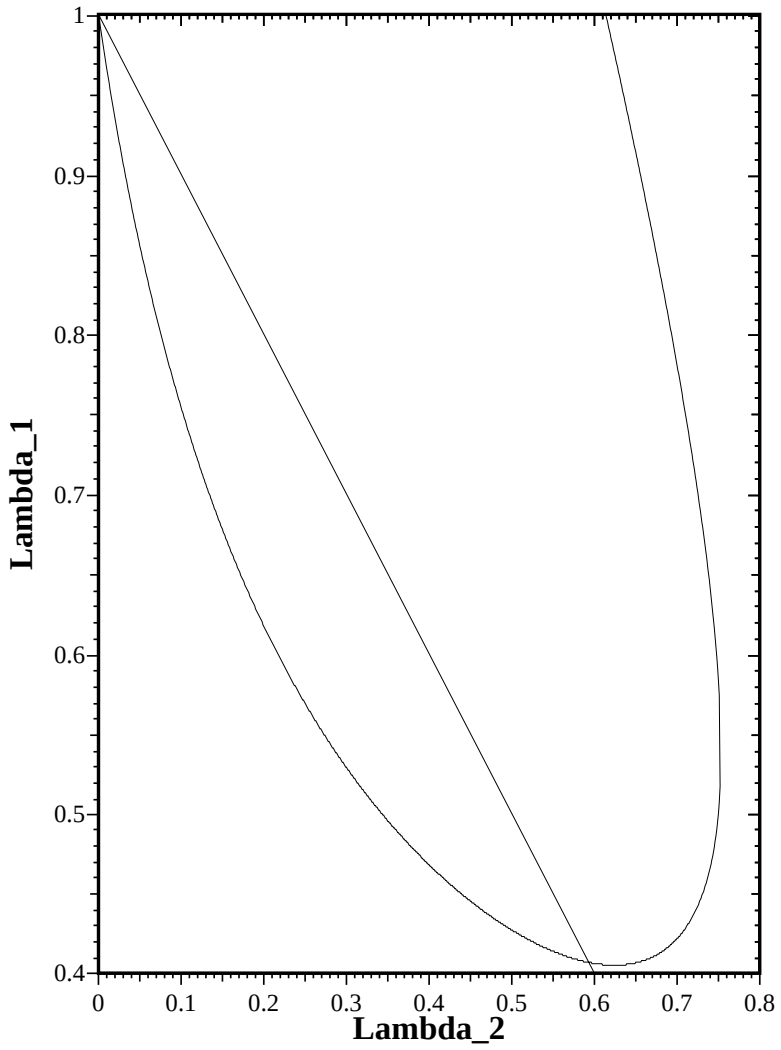
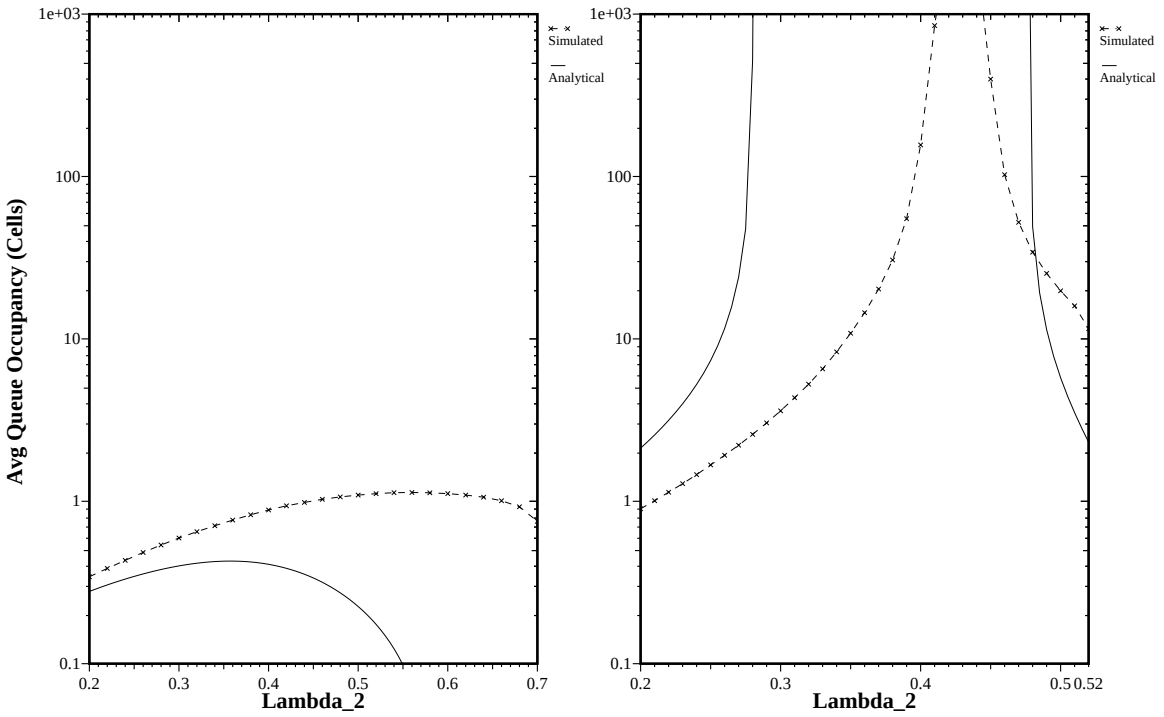


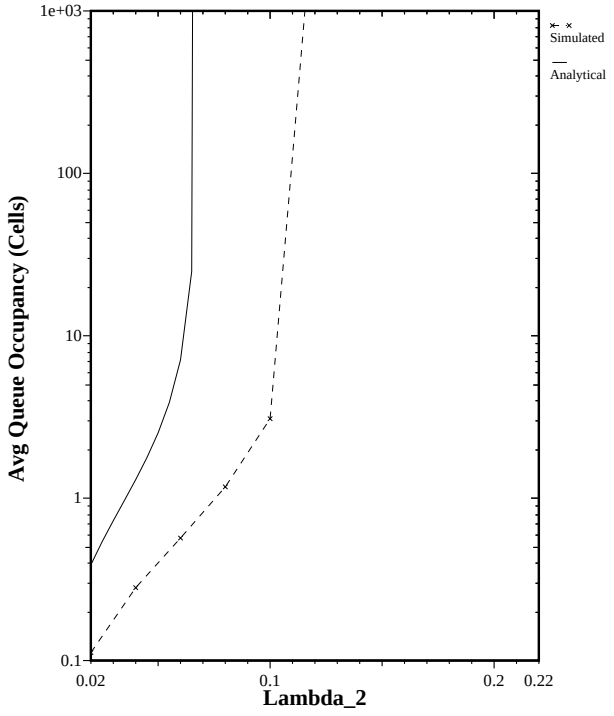
FIGURE 2.18 Region of stability for the approximate model of the switch in Figure 2.12 as a function of λ_1 and λ_2. The admissibility constraint $\lambda_1 + \lambda_2 < 1$ is shown. The stable region lies between the two curves and below the admissibility constraint.

method of Neuts [36], and its solution is described in Appendix 2 Section 2. From the steady-state distribution, $\Pi = [\underline{\Pi}_0, \underline{\Pi}_1, \underline{\Pi}_2, \ldots]$ (Appendix 2, Equation 20) we can evaluate the expected occupancy of Q(1,1).

To determine how good our simplified model is, Figure 2.19 compares the expected delay of the *simplified* switch model to the simulated average delay of the *actual* switch with three queues in Figure 2.12. We make the assumption introduced in Section 5.3.1 that $\varepsilon = 2\lambda_2$. Graphs are



(a) $\lambda_1 = 0.30$ (b) $\lambda_1 = 0.48$



(c) $\lambda_1 = 0.78$

FIGURE 2.19 Average delay as a function of offered load for 2x2 switch with SLIP scheduling algorithm. Comparison of solution of simplified model with simulated results for exact model.

shown for $\lambda_1 = 0.3, 0.4$ and 0.78 representing respectively regions in which Q(1,1) is always stable, non-monotonic and unstable.

As we found with the analytical solution for the stability region, the model of the *simplified* switch exhibits the same behavior as the actual switch, but the values for delay are quite different.

6 Variations on SLIP

6.1 Prioritized SLIP

Many applications use multiple classes of traffic with different priority levels. The basic SLIP algorithm can be extended to include requests at multiple priority levels with only a small performance and complexity penalty. We call this the Prioritized SLIP algorithm.

In Prioritized SLIP each input now maintains a separate FIFO *for each priority level* and for each output. This means that for an NxN switch with P priority levels, each input maintains PxN FIFOs. We shall label the queue between input *i* and output *j* at priority level *l*, $Q_l(i, j)$ where $1 \le i, j \le N$, $1 \le l \le P$. As before, only one cell can arrive in a cell time, so this does not require a processing speedup by the input.

The Prioritized SLIP algorithm gives *strict* priority to the highest priority request in each cell time. This means that $Q_l(i, j)$ will only be served if all queues $Q_m(i, j)$, $l < m \le P$ are empty.

The SLIP algorithm is modified as follows:

Step 1. *Request*. Input *i* selects the highest priority non-empty queue for output *j*. The input sends the priority level l_{ij} of this queue to the output *j*.

Step 2. *Grant*. If output *j* receives any requests, it determines the highest level request. i.e. it finds $L(j) = \max_i(l_{ij})$. The output then chooses one input among only those inputs that have requested at level $L(j)$. The output arbiter maintains a separate pointer, g_{jl} for each priority level. When choosing among inputs at level *L(j)*, the arbiter uses the pointer $g_{jL(j)}$ and chooses using the same round-robin scheme as before. The output notifies each input whether or not its request was granted. The pointer $g_{jL(j)}$ is incremented (modulo N) to one location beyond the granted input if and only if input *i* accepts output *j* in step 3.

Step 3. *Accept*. If input *i* receives any grants, it determines the highest level grant. i.e. it finds $L'(i) = \max_j(l_{ij})$. The input then chooses one output among only those that have requested at level $l_{ij} = L'(i)$. The input arbiter maintains a separate pointer, a_{il} for each priority level. When choosing among outputs at level $L'(i)$, the arbiter uses the pointer $a_{iL'(i)}$ and chooses using the same round-robin scheme as before. The input notifies each output whether or not its grant was accepted. The pointer $a_{iL'(i)}$ is incremented (modulo N) to one location beyond the accepted output.

Implementation of the Prioritized SLIP algorithm is more complex than the basic SLIP algorithm, but can still be fabricated from the same number of arbiters. This is because each arbiter only selects an input (output) among those requesting (granting) at the highest priority level. The arbiter now consists of two parts: the first part determines the level *l* of the highest priority request (grant) and removes those requests (grants) with levels *m*<*l*; the second part of the arbiter is the same round-robin arbiter as before. An implementation of Prioritized SLIP is described in Section 7.

6.2 Threshold SLIP

As we shall see in Chapter 4, scheduling algorithms that find a maximum *weight* match outperform those that find a maximum *sized* match. In particular, if the weight of the edge between input *i* and output *j* is the occupancy $L_{i,j}(t)$ of input queue *Q(i,j)* then we will conjecture that the algorithm is stable for all admissible i.i.d. Bernoulli arrival patterns. But maximum weight matches are significantly harder to calculate than maximum sized matches [41] and to be practical, must be implemented using an upper limit on the number of bits used to represent the occupancy of the input queue.

In the Threshold SLIP algorithm we make a compromise between the maximum sized match and the maximum weight match by quantizing the queue occupancy according to a set of threshold levels. The threshold level is then used to determine the priority level in the Priority SLIP algorithm. Each input queue maintains an ordered set of threshold levels $\mathbf{T} = \{t_1, t_2, \ldots, t_T\}$, where $t_1 < t_2 < \ldots < t_T$. If $t_a \le Q(i,j) < t_{a+1}$ then the input makes a request of level $l_i = a$.

6.3 Weighted SLIP

In some applications, the strict priority scheme of Prioritized SLIP may be undesirable, leading to starvation of low-priority traffic. The Weighted SLIP algorithm can be used to divide the throughput to an output non-uniformly among competing inputs. The bandwidth from input *i* to output *j* is now a ratio $f_{ij} = \frac{n_{ij}}{d_{ij}}$ subject to the admissibility constraints $\sum_i f_{ij} < 1, \sum_j f_{ij} < 1$.

In the basic SLIP algorithm each arbiter maintains an ordered circular list, $\boldsymbol{S} = \{1, \ldots, N\}$. In the Weighted SLIP algorithm the list is expanded at output *j* to be the ordered circular list $\boldsymbol{S}_j = \{1, \ldots, W_j\}$ where W_j = LowestCommonMultiple(d_{ij}) and input *i* appears $\frac{n_{ij}}{d_{ij}} \times W_j$ times in $\boldsymbol{S}_j$.

6.4 Least Recently Used

When an output arbiter in SLIP successfully selects an input, that input becomes the *lowest priority* in the next cell time. This is intuitively a good characteristic: the algorithm should least favor connections that have been served recently. But which input should now have the *highest* priority? In SLIP, it is the next input that happens to be in the schedule. But this is not necessarily the input that was served *least* recently. By contrast, the Least Recently Used (LRU) algorithm gives highest priority to the least recently used and lowest priority to the most recently used.

LRU is identical to SLIP except for the ordering of the elements in the arbiter list: they are no longer in ascending order of input number but rather are in an ordered list starting from the least recently to most recently selected. If a grant is successful, the input that is selected is moved to the end of the ordered list. Similarly, an LRU list can be kept at the inputs for choosing among competing grants.

We might expect LRU to perform as well as, if not better than SLIP. But as we can see from Figure 2.20, it performs significantly worse when the offered load is greater than 65%. This is because the output arbiters do not tend to *desynchronize* and several may grant to the same input, as shown in Figure 2.21. Each schedule can become re-ordered at the end of each cell time which, over many cell times, leads to a random ordering of the schedules. This in turn leads to a high probability that the pointers at two or more outputs will point to the same input: the same problem

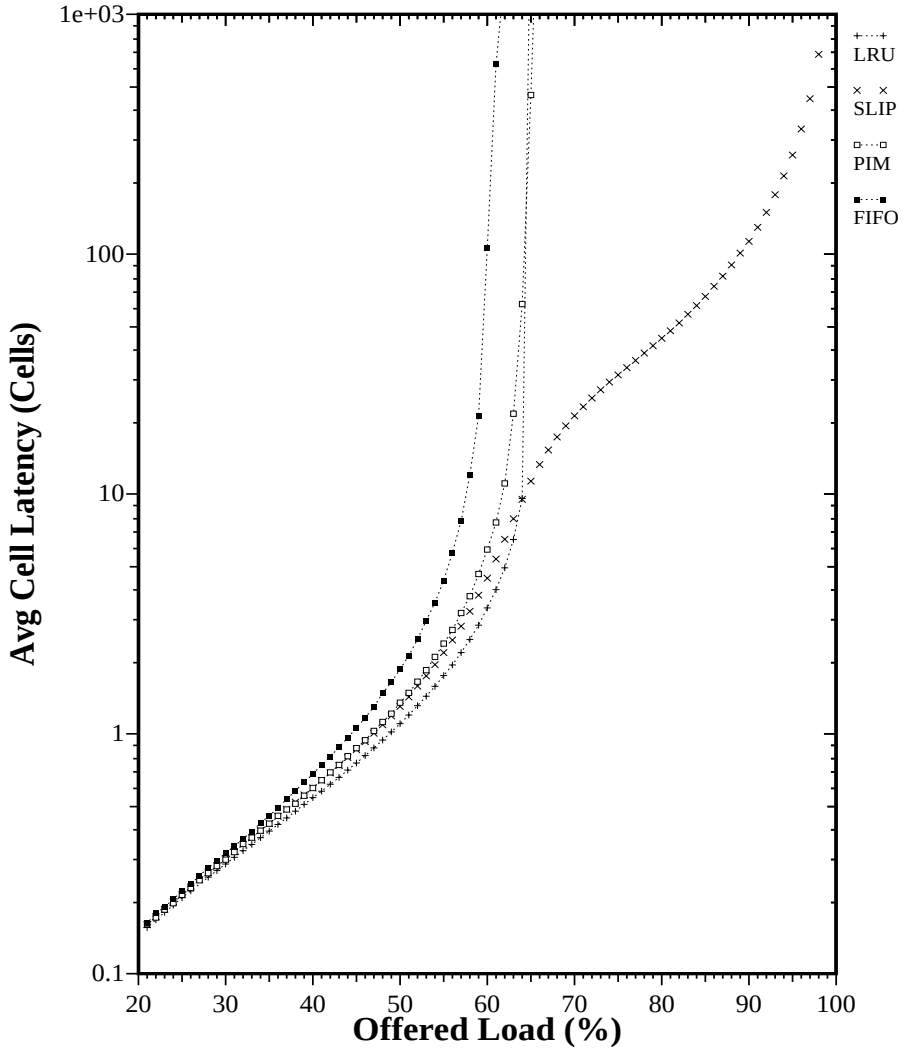


FIGURE 2.20 LRU performs no better than PIM for a single iteration. Results shown for 16x16 switch with i.i.d. Bernoulli arrivals.

encountered by RRM and PIM with a single iteration. This explains why the performance for PIM and LRU are very similar.

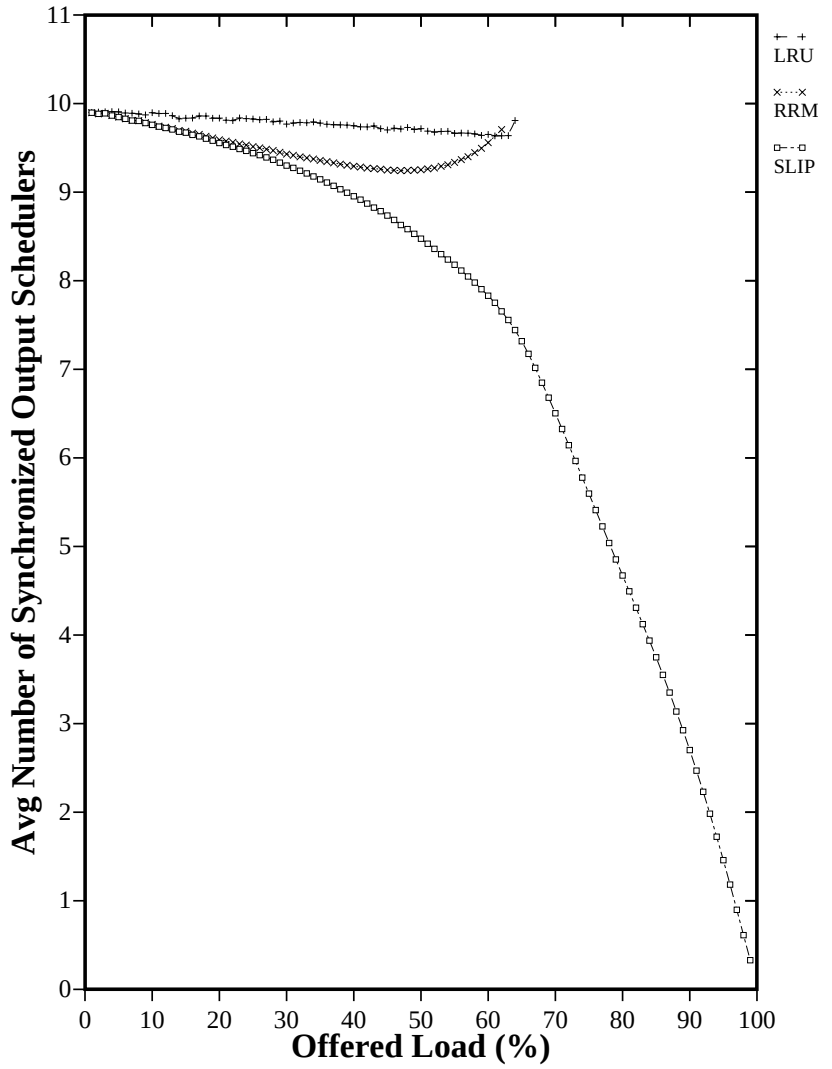


FIGURE 2.21 LRU performs poorly because of the synchronization between the output arbiters. Results shown for 16x16 switch with i.i.d. Bernoulli arrivals.

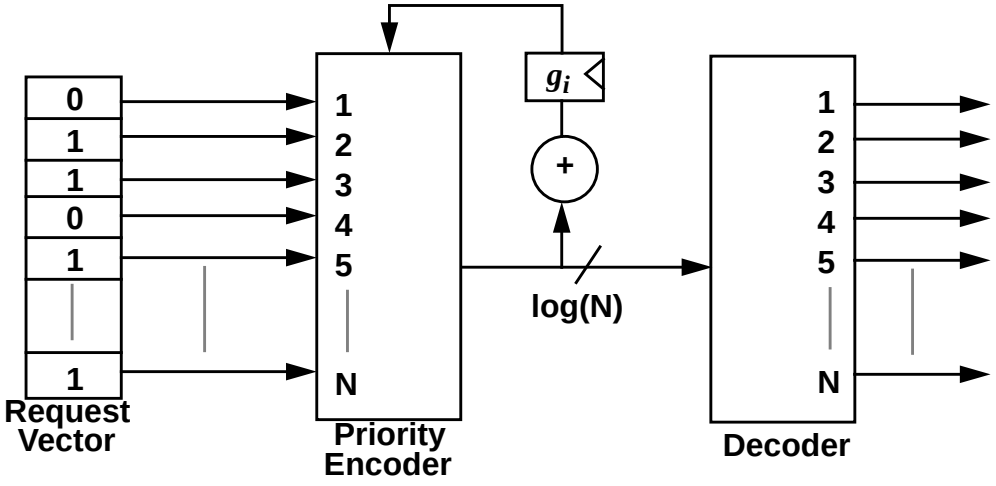


FIGURE 2.22 Round-robin *grant* arbiter for SLIP and RRM algorithms. The priority encoder has a programmed highest-priority, g_i. The *accept* arbiter at the input is identical.

7 Implementing SLIP

One of the objectives of this work was to design a scheduler that is simple to implement. To conclude our description of SLIP, in this section we consider the complexity of implementing SLIP in hardware, arguing that with current technology it is feasible to implement a centralized scheduler for a 32x32 switch on a single chip.

As illustrated in Figure 2.22, each SLIP arbiter consists of a priority encoder with a programmable highest priority, a register to hold the highest priority value, and an incrementer to move the pointer after it has been updated. The decoder is necessary to provide a decision line for each bit in the request vector.

Figure 2.23 shows how 2N arbiters and an N^2-bit memory are interconnected to construct a SLIP scheduler for an NxN switch. The state memory records whether an input queue is empty or non-empty. From this memory, an N^2-bit wide vector presents N bits to each of N *grant* arbiters. The grant arbiters select a single input among the contending requests. The grant decision from each grant arbiter is then passed to the N *accept* arbiters. Each arbiter selects at most one output on

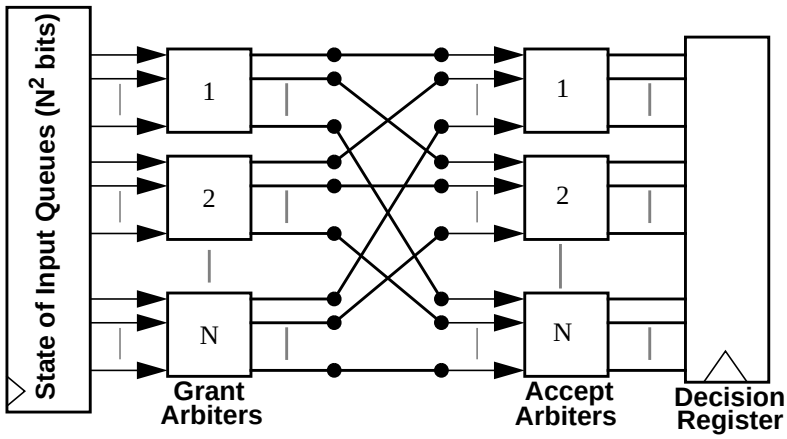


FIGURE 2.23 Interconnection of 2N arbiters to implement SLIP for an NxN switch.

behalf of an input. The final decision is then saved in a decision register and the values of the g_i and a_i pointers are updated. The decision register is used to notify each input which cell to transmit and to configure the crossbar switch.

The area required to implement the scheduler in silicon is dominated by the priority encoders.

Switch Size (N)	Number of 2-input gates per arbiter	Total number of 2-input gates for N arbiters
4	44	176
8	280	2,240
16	1,637	29,192
32	13,169	421,408

Table 2.1 Estimate of number of 2-input gates required to implement 1 and N arbiters for a SLIP scheduler.

An estimate of the number of 2-input gates required to implement the programmable priority using a PAL structure is shown in Table 2.1[1]. This table shows that the number of gates per arbiter grows approximately with N^3 and hence with N^4 for the full scheduler. In some implementations, it may

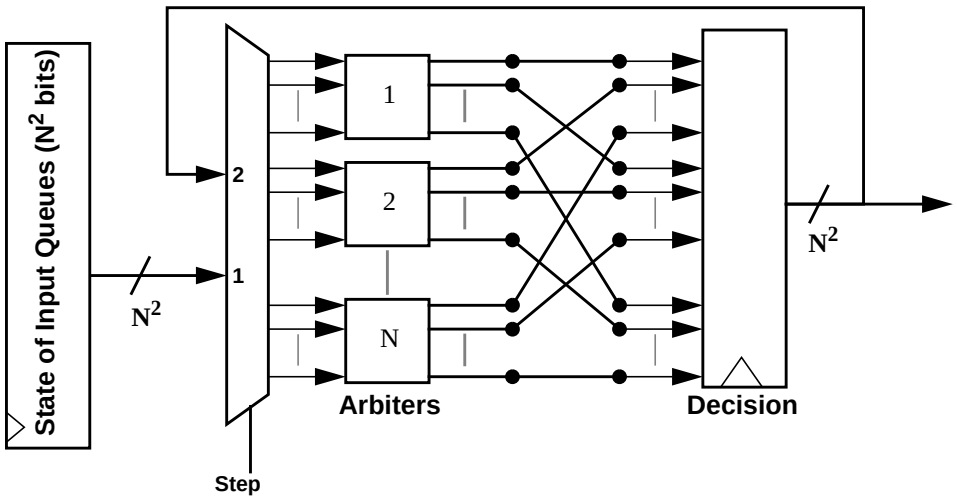


FIGURE 2.24 Interconnection of N arbiters to implement SLIP for an NxN switch. Each arbiter is used for both input and output arbitration. In this case, each arbiter contains *two* registers to hold pointers g_i and a_i.

be desirable to reduce the number of arbiters, sharing them among both the grant and accept steps of the algorithm. Such an implementation requiring only N arbiters[1] is shown in Figure 2.24. When the results from the grant arbiter have settled, they are registered and fed back to the input for the second step. Obviously each arbiter must maintain a separate register for the g_i and a_i pointers, selecting the correct pointer for each step.

Assuming that the design is dominated by the arbiters, Table 2.1 indicates that fewer than 500,000 gates are required for a 32x32 switch. This is easily feasible in current gate-array technology.

7.1 Prioritized SLIP

The Prioritized SLIP algorithm was described in Section 6.1 and is also the basis of the Threshold SLIP algorithm in Section 6.2.

1. These values were obtained using *espresso* and *misII* from the Berkeley Octtools VLSI design package. No attempt was made to manually optimize the design.

1. A slight performance penalty is introduced by registering the output of the grant step and feeding back the result as the input to the accept step. This is likely to be small in practice.

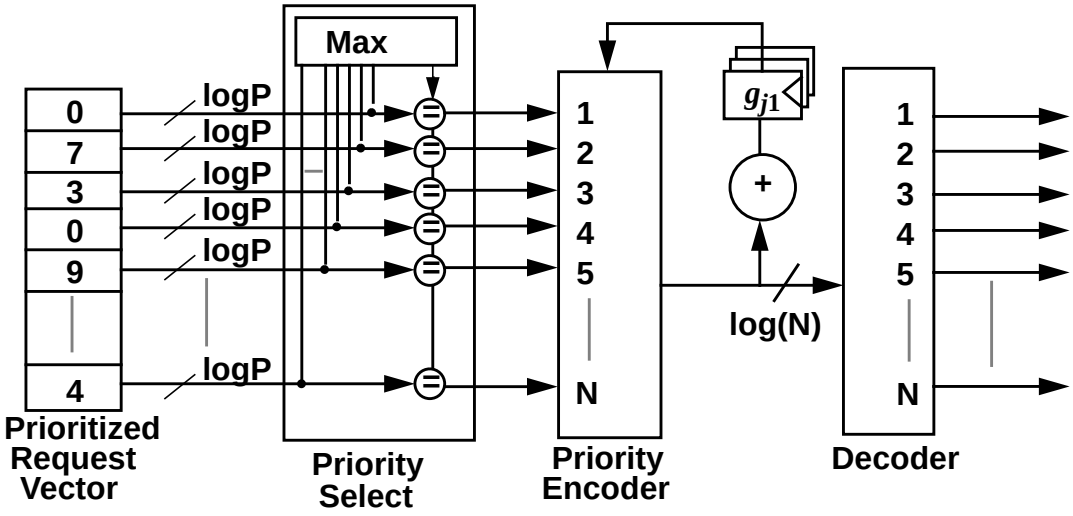


FIGURE 2.25 *Grant* arbiter for Prioritized SLIP with P priority levels. The "Priority Select" block selects only those requests at the highest requested priority level, L(*j*). The priority encoder uses pointer $g_{jL(j)}$.The decoder determines which input to send L(*j*) to.

For a small number of priority levels (e.g. 2 or 3), a *grant* arbiter for Prioritized SLIP can be implemented using a separate request vector for each priority level. The arbiter selects the highest priority, non-zero request vector and from that vector selects a single input as before. The arbiter makes its decision using a single priority encoder, but must maintain a separate pointer for each priority level. When the grant arbiter has made its selection, the result and priority level is fed to each *accept* arbiter, which operate in an identical manner to the grant arbiters.

If the number of priority levels is large, it is more efficient for an input (output) to supply the grant (accept) arbiter with just the highest priority request (grant). An example of an arbiter for Prioritized SLIP is shown in Figure 2.25. The arbiter selects the highest priority request and considers only those requests at this level. Although the arbiter does not require any additional priority encoders, for P priority levels it requires P pointer registers, a combinatorial circuit with N inputs, each logP bits wide to determine the maximum requested priority level and N 2-input comparators, each logP bits wide.

CHAPTER 3

The SLIP Algorithm with Multiple Iterations

1 Introduction

In this chapter we consider the SLIP algorithm with multiple iterations per cell time. We shall call the generic algorithm “iterative SLIP” (*i*-SLIP). When the number of iterations *n* is known, we shall call the algorithm *n*-SLIP. For example, Chapter 2 focussed on 1-SLIP.

With more than one iteration, the iterative SLIP algorithm improves upon the performance of 1-SLIP: each iteration attempts to add connections not made by earlier iterations.

We begin this section with a description of *i*-SLIP, emphasizing the differences from non-iterative SLIP. In particular, we pay careful attention to the way that the arbiter pointers are updated. In Section 3 we present some results from a simulation study of *i*-SLIP. As we found for 1-SLIP, iterative SLIP is stable for all admissible uniform i.i.d. Bernoulli arrivals. We find that the performance improves as we increase the number of iterations up to about log_2N, for an NxN switch. Once again, we shall see that *desynchronization* of the output arbiters with respect to each other plays an important rôle in achieving low latency. However, we will also see that the basic *i*-SLIP algorithm tends to do a worse job of desynchronizing the arbiters as the number of iterations increase.

In Section 4 we try to improve upon the basic iterative SLIP algorithm by changing the rules for updating the pointers so that desynchronization is improved when the number of iterations is increased. We consider the extra complexity that these changes introduce.

Finally, in Section 5 we describe an implementation of iterative SLIP, showing that although it may take longer to execute, the implementation is only slightly more complex than the implementation of non-iterative SLIP.

2 The Iterative SLIP Matching Algorithm

2.1 Description

The *i*-SLIP algorithm is an enhancement of the SLIP algorithm described in Chapter 2, but has a number of differences specific to its iterative behavior. As before, at the beginning of each cell time, the match process begins over. All inputs and outputs are initially unmatched and only those inputs and outputs not matched at the end of one iteration are eligible for matching in the next. Connections made in one iteration are never removed by a later iteration, even if a larger sized match would result. The three steps of each iteration operate in parallel on each output and input and are as follows:

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

Step 2. *Grant*. If an unmatched output receives any requests, it chooses the one that appears next in a fixed, round-robin schedule starting from the highest priority element. The output notifies each input whether or not its request was granted. The pointer g_i to the highest priority element 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.*

Step 3. *Accept*. If an unmatched input receives a grant, it accepts the one that appears next in a fixed, round-robin schedule starting from the highest priority element. The pointer a_i to the highest priority element of the round-robin schedule is incremented (modulo N) to one location beyond the accepted output *only if this input was matched in the first iteration.*

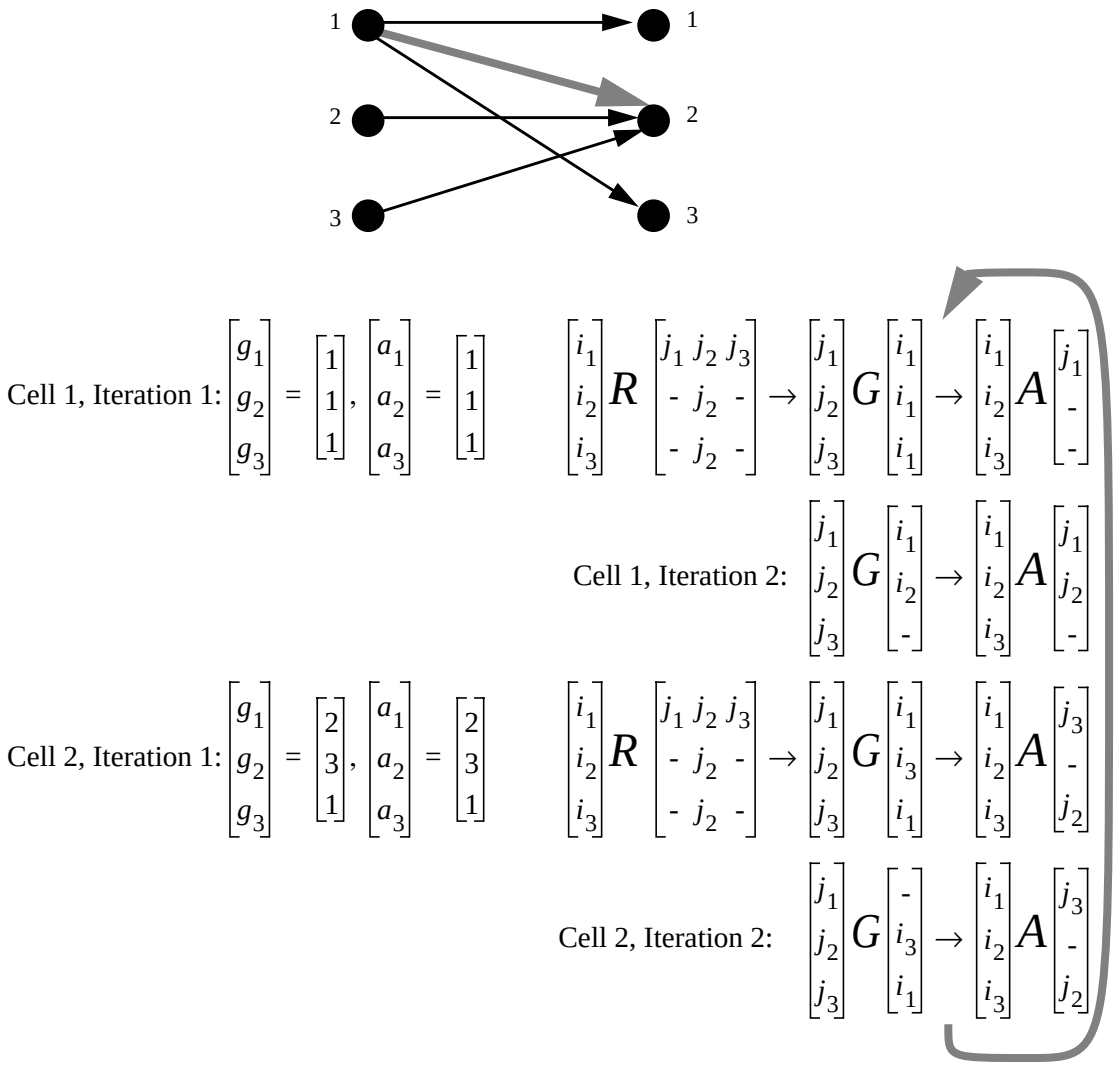


FIGURE 3.1 Example of starvation, if pointers are updated after every iteration. The 3x3 switch is heavily loaded, i.e. all active connections have an offered load of 1 cell per cell time. The sequence of grants and accepts repeats after 2 cell times, even though the (highlighted) connection from input 1 to output 2 has not been made. Hence, this connection will be starved indefinitely.

2.2 Updating Pointers

Note that pointers g_i and a_i are only updated for matches found in the first iteration. Connections made in subsequent iterations do not cause the pointers to be updated. This is to avoid starvation. To understand how starvation can occur, we refer to the example of a 3x3 switch with 5 active and heavily loaded connections, shown in Figure 3.1. The switch is scheduled with the *2*-SLIP algorithm, except in this case the pointers are updated after both iterations. The figure shows the sequence of decisions by the grant and accept arbiters; for this traffic pattern, they form a repetitive

cycle in which *the highlighted connection from input 1 to output 2 is never served*. Each time the round-robin arbiter at output 2 grants to input 1, input 1 chooses to accept output 1 instead.

Starvation is eliminated if the pointers are not updated after the first iteration. In the example, output 2 would continue to grant to input 1 with highest priority until it is successful.

2.3 Properties

The *i*-SLIP algorithm has the following properties:

Property 1. Connections matched in the first iteration become the lowest priority in the next cell time. This is the same as Property 1 of 1-SLIP described in Chapter 2 Section 3.

Property 2. No connection is starved. As with 1-SLIP, and because of the requirement that pointers are not updated after the first iteration, an output will continue to grant to the highest priority requesting input until it is successful.

Property 3. For *i*-SLIP with 1 iteration, and under heavy load, queues with a common output all have the same throughput. This is the same as in Chapter 2 Section 3.

Property 4. For *i*-SLIP with more than one iteration, and under heavy load, queues with a common output may each have a different throughput. An example of this property is shown in Figure 3.2 for a heavily loaded 3x3 switch scheduled using 2-SLIP. The state of the grant and accept arbiters forms a cycle that repeats every three cell times. Note that although all non-empty queues are served, Q(2,3) is served *twice* per cycle whereas Q(1,3) is served only *once*.

Property 5. The algorithm will converge in at most N iterations. Each iteration will schedule zero, one or more connections. If zero connections are scheduled in an iteration then the algorithm has converged: no more connections can be added with more iterations. Therefore, the slowest convergence will occur if exactly one connection is scheduled in each iteration. At most N connections can be scheduled (one to every input and one to every output) which means the algorithm will converge in at most N iterations.

Property 6. The algorithm will not necessarily converge to a maximum sized match. At best, it will find a *maximal* match: the largest size match without removing connections made in earlier iterations.

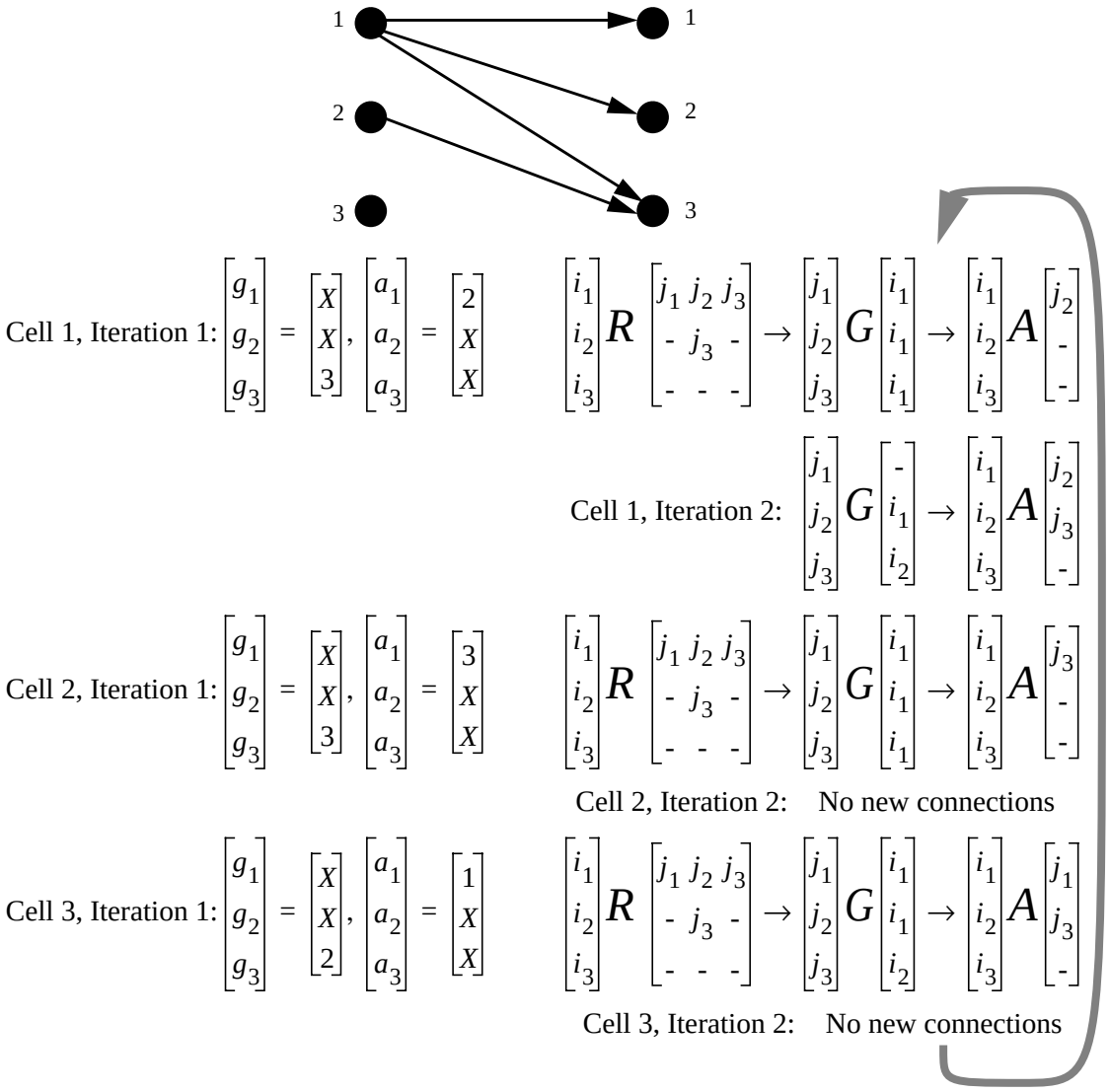


FIGURE 3.2 Example of unequal service under heavy load for 2 inputs that share an output.The 3x3 switch is heavily loaded, i.e. all four active connections have an offered load of 1 cell per cell time. The sequence of grants and accepts for 2-SLIP repeats after 3 cell times. During each cycle, Q(2,3) is served twice whereas Q(1,3) is served only once. Note that for 1-SLIP, only the 1st cell time would be different and Q(2,3) would be served only once per cycle.

3 Simulated Performance of Iterative SLIP

3.1 How Many Iterations?

Now that we have an iterative algorithm, we need to decide how many iterations to perform

during each cell time. Ideally, from Property 5 above we would like to perform N iterations. However, in practice there may be insufficient time for N iterations, and so we need to consider the penalty of performing only i iterations, where $i < \mathrm{N}$. In fact, because of the desynchronization of the arbiters, i-SLIP will usually converge in fewer than N iterations. An interesting example of this is shown in Figure 3.3. In the first cell time, the algorithm takes N iterations to converge, but thereafter converges in one less iteration each cell time. After N cell times, the arbiters have become totally desynchronized and the algorithm will converge in a single iteration.

How many iterations should we use: it clearly doesn't always take N? One option is to always run the algorithm to completion, resulting in a scheduling time that varies from cell to cell. In some applications this may be acceptable. In others, such as in an ATM switch, it is desirable to maintain a fixed scheduling time and to try and fit as many iterations into that time as possible.

Under simulation, we have found that for an NxN switch it takes *about* $\log_2 N$ iterations for i-SLIP to converge. This is similar to the results obtained for PIM in [3], in which the authors prove that

$$\mathrm{E}(I) \leq \log_2 N + \frac{4}{3}, \tag{1}$$

where I is the number of iterations that PIM takes to converge. However, although for all the stationary arrival processes we have considered $\mathrm{E}(I) < \log_2 N$ for i-SLIP, we have not been able to prove that this relation holds in general.

As an example, Figure 3.4 compares the number of iterations required for PIM and i-SLIP to converge under uniform i.i.d. Bernoulli arrivals.

3.2 Bernoulli Traffic

To illustrate the improvement in performance of i-SLIP when the number of iterations is increased, Figure 3.5 shows the average queueing delay for 1, 2 and 4 iterations under uniform i.i.d. Bernoulli arrivals. We find that multiple iterations of i-SLIP significantly increase the size of the match and therefore reduce the queueing delay[1]. In fact, n-SLIP is stable for all n under admis-

1. Although not shown, we find that increasing i above 4 for a 16x16 switch leads to a negligible performance improvement.

Cell 1, Iteration 1:

$$\begin{bmatrix} g_1 \\ g_2 \\ : \\ g_N \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ : \\ 1 \end{bmatrix}, \begin{bmatrix} a_1 \\ a_2 \\ : \\ a_N \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ : \\ 1 \end{bmatrix} \qquad \begin{bmatrix} i_1 \\ i_2 \\ : \\ i_N \end{bmatrix} R \begin{bmatrix} j_1\ j_2\ \cdots\ j_N \\ j_1\ j_2\ \cdots\ j_N \\ :\ :\ \ldots\ : \\ j_1\ j_2\ \cdots\ j_N \end{bmatrix} \rightarrow \begin{bmatrix} j_1 \\ j_2 \\ : \\ j_N \end{bmatrix} G \begin{bmatrix} i_1 \\ i_1 \\ : \\ i_1 \end{bmatrix} \rightarrow \ i_1 \, A \, j_1$$

Iteration N:

$$\begin{bmatrix} j_1 \\ j_2 \\ : \\ j_N \end{bmatrix} G \begin{bmatrix} i_1 \\ i_2 \\ : \\ i_N \end{bmatrix} \rightarrow \begin{bmatrix} i_1 \\ i_2 \\ : \\ i_N \end{bmatrix} A \begin{bmatrix} j_1 \\ j_2 \\ : \\ j_N \end{bmatrix}$$

Cell 2, Iteration 1:

$$\begin{bmatrix} g_1 \\ g_2 \\ : \\ g_N \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \\ : \\ 1 \end{bmatrix}, \begin{bmatrix} a_1 \\ a_2 \\ : \\ a_N \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \\ : \\ 1 \end{bmatrix} \qquad \begin{bmatrix} i_1 \\ i_2 \\ : \\ i_N \end{bmatrix} R \begin{bmatrix} j_1\ j_2\ \cdots\ j_N \\ j_1\ j_2\ \cdots\ j_N \\ :\ :\ \ldots\ : \\ j_1\ j_2\ \cdots\ j_N \end{bmatrix} \rightarrow \begin{bmatrix} j_1 \\ j_2 \\ : \\ j_N \end{bmatrix} G \begin{bmatrix} i_1 \\ i_1 \\ : \\ i_1 \end{bmatrix} \rightarrow \begin{bmatrix} i_1 \\ i_2 \end{bmatrix} A \begin{bmatrix} j_2 \\ j_1 \end{bmatrix}$$

Iteration N-1:

$$\begin{bmatrix} j_1 \\ j_2 \\ : \\ j_N \end{bmatrix} G \begin{bmatrix} i_1 \\ i_2 \\ : \\ i_N \end{bmatrix} \rightarrow \begin{bmatrix} i_1 \\ i_2 \\ i_3 \\ : \\ i_N \end{bmatrix} A \begin{bmatrix} j_2 \\ j_1 \\ j_3 \\ : \\ j_N \end{bmatrix}$$

:

:

Cell N, Iteration 1:

$$\begin{bmatrix} g_1 \\ g_2 \\ : \\ g_N \end{bmatrix} = \begin{bmatrix} N \\ N-1 \\ : \\ 1 \end{bmatrix}, \begin{bmatrix} a_1 \\ a_2 \\ : \\ a_N \end{bmatrix} = \begin{bmatrix} N \\ N-1 \\ : \\ 1 \end{bmatrix} \qquad \begin{bmatrix} i_1 \\ i_2 \\ : \\ i_N \end{bmatrix} R \begin{bmatrix} j_1\ j_2\ \cdots\ j_N \\ j_1\ j_2\ \cdots\ j_N \\ :\ :\ \ldots\ : \\ j_1\ j_2\ \cdots\ j_N \end{bmatrix} \rightarrow \begin{bmatrix} j_1 \\ j_2 \\ : \\ j_N \end{bmatrix} G \begin{bmatrix} i_N \\ i_{N-1} \\ : \\ i_1 \end{bmatrix} \rightarrow \begin{bmatrix} i_1 \\ i_2 \\ : \\ i_N \end{bmatrix} A \begin{bmatrix} j_N \\ j_{N-1} \\ : \\ j_1 \end{bmatrix}$$

FIGURE 3.3 Example of the number of iterations required to converge for a heavily loaded NxN switch. All input queues remain non-empty for the duration of the example. In the first cell time, the arbiters are all synchronized. During each cell time, one more arbiter is desynchronized from the others. After N cell times, all arbiters are desynchronized and a maximum sized match is found in a single iteration.

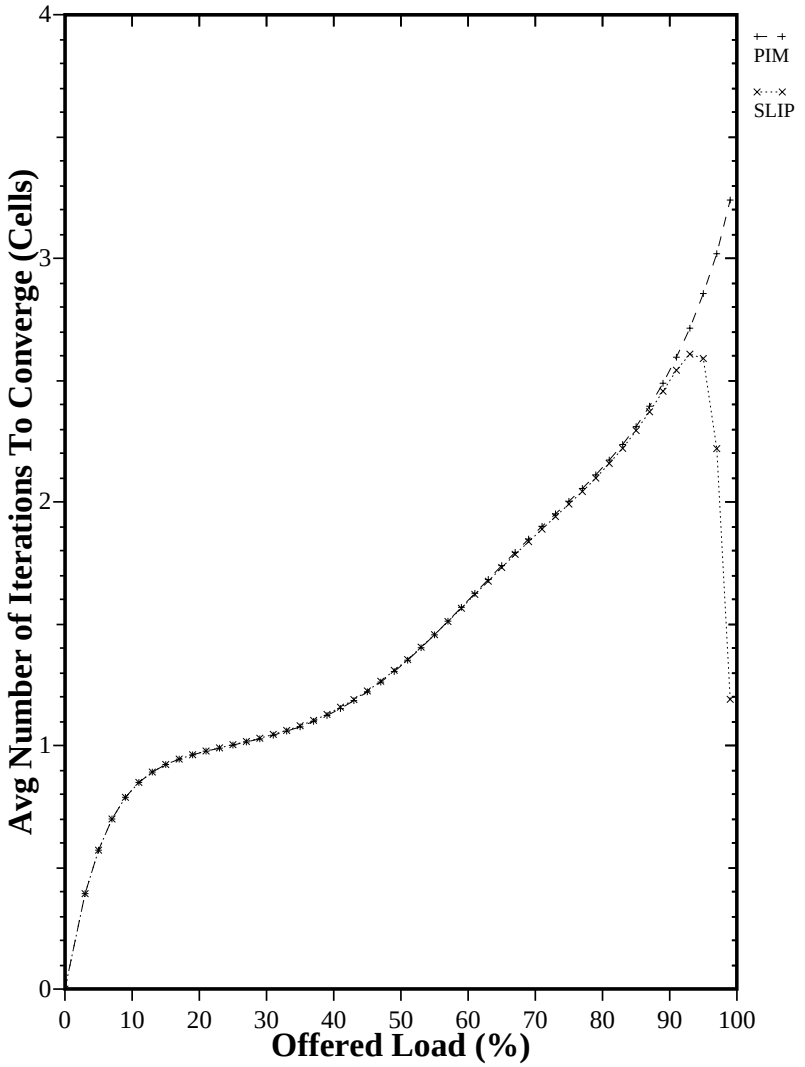


FIGURE 3.4 An example of the number of iterations for *i*-SLIP and PIM to converge for uniform i.i.d. Bernoulli traffic as a function of the offered load for a 16x16 switch. Each algorithm is run to completion during each cell time to determine how many iterations are required before no more connections can be added.

sible uniform i.i.d. Bernoulli arrivals. This should come as no surprise: in Chapter 2 we saw that 1-SLIP is stable under these conditions. Intuitively, the size of the match increases with the number of iterations: each new iteration potentially adds connections not made by earlier iterations. As a result, for a given set of queue occupancies *(n+1)*-SLIP can provide an instantaneous match closer to the maximum sized match than *n*-SLIP. This is illustrated in Figure 3.6 which compares the size of each *i*-SLIP matching with the size of the maximum matching for the same instantaneous queue occupancies. Under low offered load, the 1-SLIP arbiters move randomly and the ratio of the

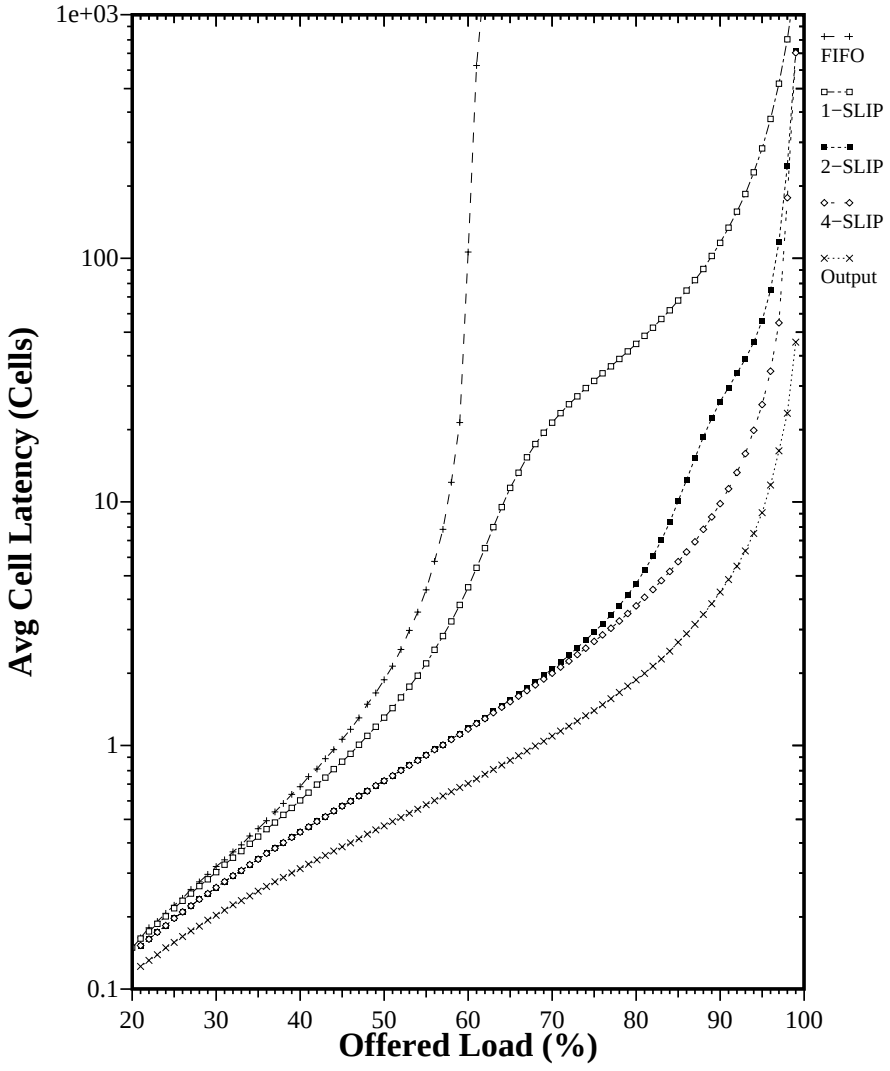


FIGURE 3.5 Performance of *i*-SLIP for 1,2 and 4 iterations compared with FIFO and output queueing for i.i.d Bernoulli arrivals with destinations uniformly distributed over all outputs. Results obtained using simulation for a 16x16 switch. The graph shows the average delay per cell, measured in cell times, between arriving at the input buffers and departing from the switch.

match size to the maximum match size decreases with increased offered load. But when the load exceeds approximately 65%, the ratio begins to increase linearly. This is the result of desynchronization of the output arbiters which leads to a better and better match as the load increases. 2-SLIP and 4-SLIP behave similarly and, as expected, the ratio increases with the number of iterations indicating that the matching gets closer to the maximum sized match. But only up to a point: for an 16x16 switch under this traffic load, increasing the number of iterations beyond four does not measurably increase the average match size.

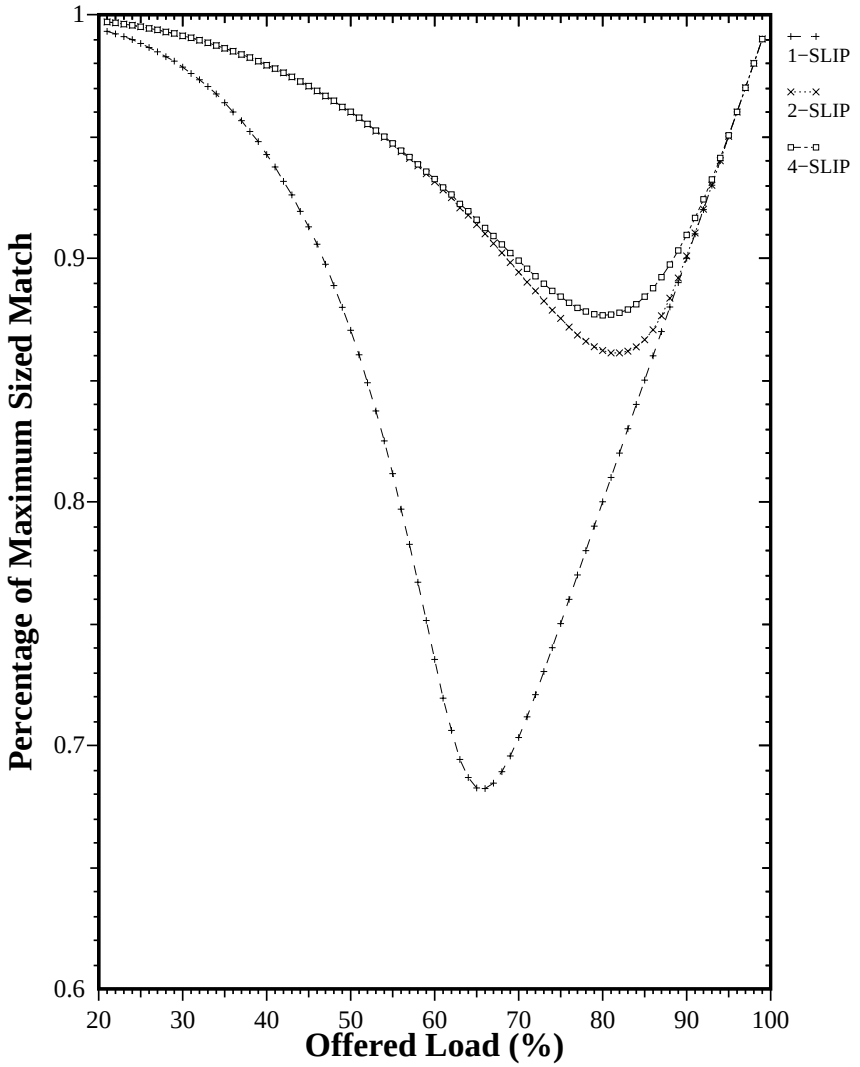


FIGURE 3.6 Comparison of the match size for *i*-SLIP with the size of a maximum sized match for the same set of requests. Results are for a 16x16 switch and uniform i.i.d. Bernoulli arrivals.

Although the average queueing delay is reduced by increasing the number of iterations, the synchronization of the output arbiters *increases*, as shown in Figure 3.7. This unfortunate behavior is the consequence of only allowing the arbiters to be updated after the first iteration. With 1-SLIP, every successful connection moves an arbiter's pointer, leading to a rapid desynchronization under high offered load. For 2-SLIP and 4-SLIP, only those connections made by the first iteration move the arbiter's pointer, leading to a slower rate of desynchronization. Later in this chapter we will consider ways that *i*-SLIP can be modified to reduce the arbiter synchronization.

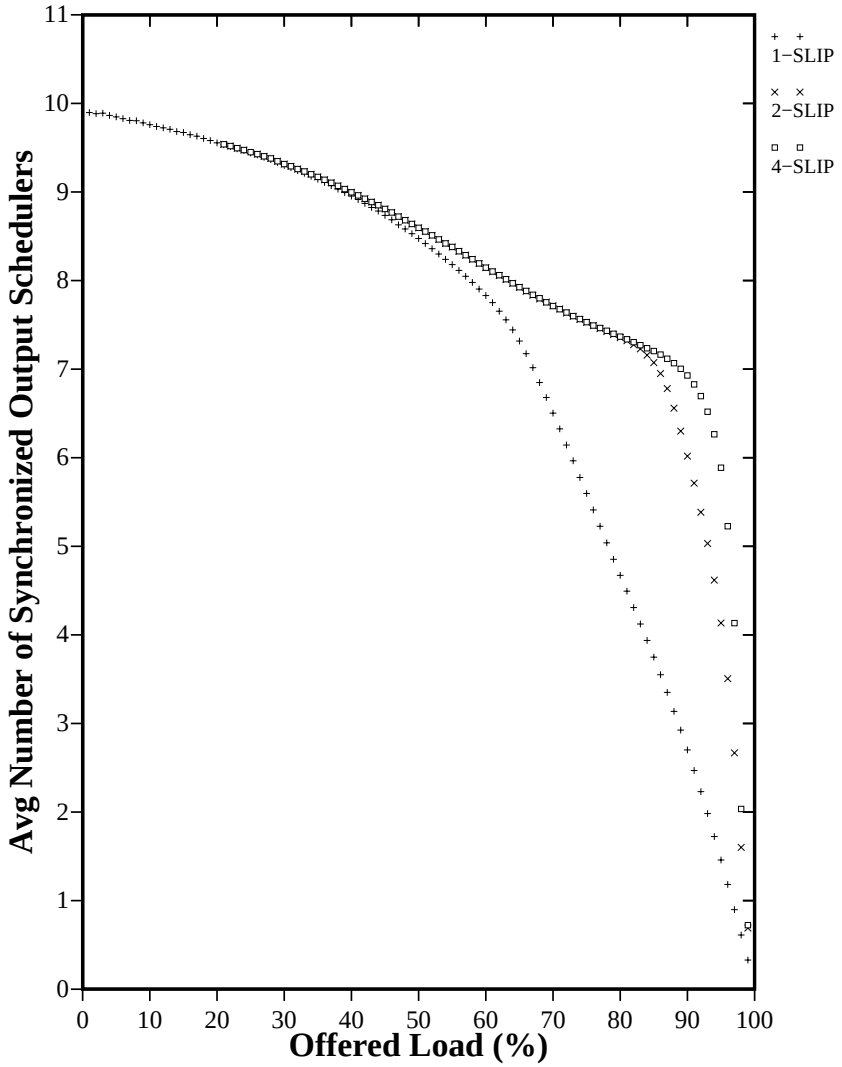


FIGURE 3.7 Average number of synchronized output schedulers as a function of offered load for uniform i.i.d. Bernoulli arrivals. Scheduling is with 1-SLIP, 2-SLIP and 4-SLIP.

3.3 Bursty Traffic

As we did for 1-SLIP, we illustrate the effect of burstiness on *i*-SLIP using an on-off arrival process modulated by a 2-state Markov-chain. The source alternately produces a burst of full cells (all with the same destination) followed by an idle period of empty cells. The bursts and idle periods contain a geometrically distributed number of cells.

Figure 3.8 shows the performance of *i*-SLIP under this arrival process for a 16x16 switch, comparing the performance for 1, 2 and 4 iterations. As we would expect, the increased burst size

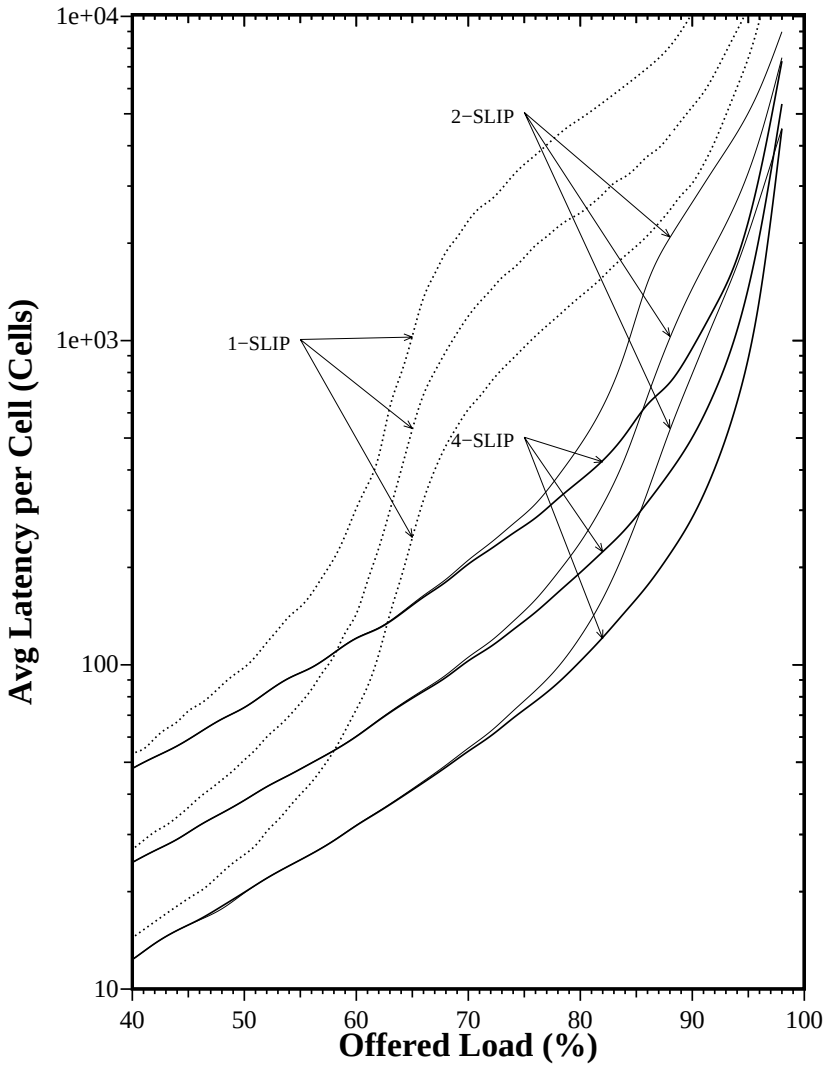


FIGURE 3.8 Performance of *i*-SLIP for 1, 2 and 4 iterations under bursty arrivals. Arrival process is a 2-state Markov-modulated on-off process. Average burst lengths are 16, 32 and 64 cells.

leads to a higher queueing delay whereas an increased number of iterations leads to a lower queueing delay. In all three cases, the average latency is *proportional* to the expected burst length. As pointed out in Chapter 2, the performance for bursty traffic is not heavily influenced by the queueing policy.

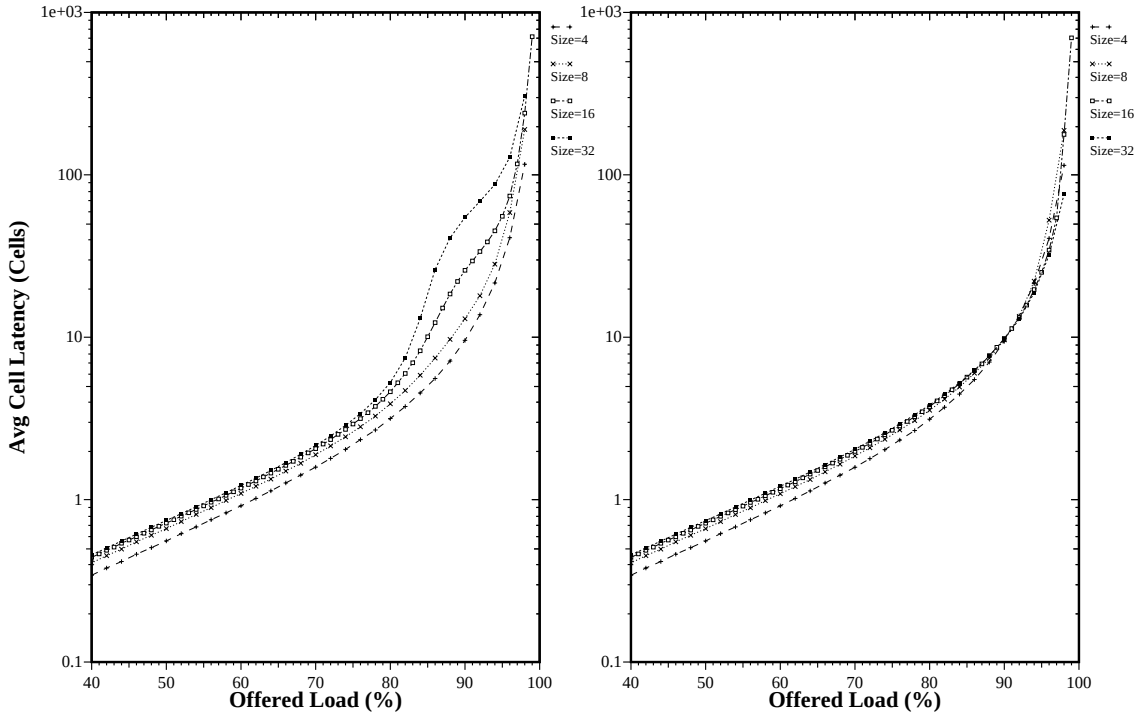


FIGURE 3.9 The performance of *i*-SLIP as function of switch size. Uniform i.i.d. Bernoulli arrivals.

3.4 As a Function of Switch Size

In Chapter 2 we found that 1-SLIP performance degrades as the switch size increases. As shown in Figure 3.9(a), we find similar behavior for 2-SLIP. Under low offered load the average cell latency approaches a constant, whereas under high load the delay is approximately proportional to N.

Although similar under low offered load, 4-SLIP exhibits quite different behavior under high offered load: the average latency can actually *decrease* with N. This is shown in Figure 3.9(b). Under an offered load below approximately 80% the ordering is strict: increasing *N* increases average latency. Between 80% and 100% offered load, the ordering changes — at 99% offered load the ordering has reversed and the average latency decreases strictly with *N*.

It should be noted that the *maxsize* algorithm does not exhibit the same behavior. The average latency for the *maxsize* algorithm always increases with *N*.

Because of the difficulty of analyzing this algorithm with more than a single iteration, we offer only a heuristic explanation for this result. We believe the result to be the combination of two effects. First, for switches up to a size 64x64, the algorithm almost always converges in fewer than 4 iterations. This means that in 4 iterations, the match size is close to the *maximal* match: the largest match possible without rearranging connections. Second, the number of possible matches under heavy load equals approximately *N*! which grows rapidly with *N*. Moreover, as *N* increases it becomes more likely that a *maximal* match equals a maximum match. Hence, with sufficient iterations and as *N* increases, it becomes more likely that the SLIP algorithm will find a match close or equal to a maximum sized match. By comparing the size of the match with the maximum sized match, we have found this to be the case.

It should be noted that the PIM algorithm exhibits similar behavior. This supports our explanation: PIM like *i*-SLIP, converges on a *maximal* match.

4 Variations of Iterative SLIP

In Chapter 2 we found that 1-SLIP performs well because of the desynchronization of the output arbiters under high offered load. As shown in Figure 3.7 *i*-SLIP is less effective at desynchronizing its arbiters, because the pointers can only be updated for connections made by the first iteration.

In this section we consider two variations on *i*-SLIP that allow the pointers to be updated after every iteration. Both variations are significantly more complex to implement than basic *i*-SLIP and the performance improvements are inconclusive. We believe that these algorithms require further study.

4.1 Iterative SLIP with LRU Accept Arbiters

Figure 3.1 showed how starvation could occur if we allowed the pointers to be updated after every iteration. In the example, the problem was not that output 2 never grants to input 1. Rather,

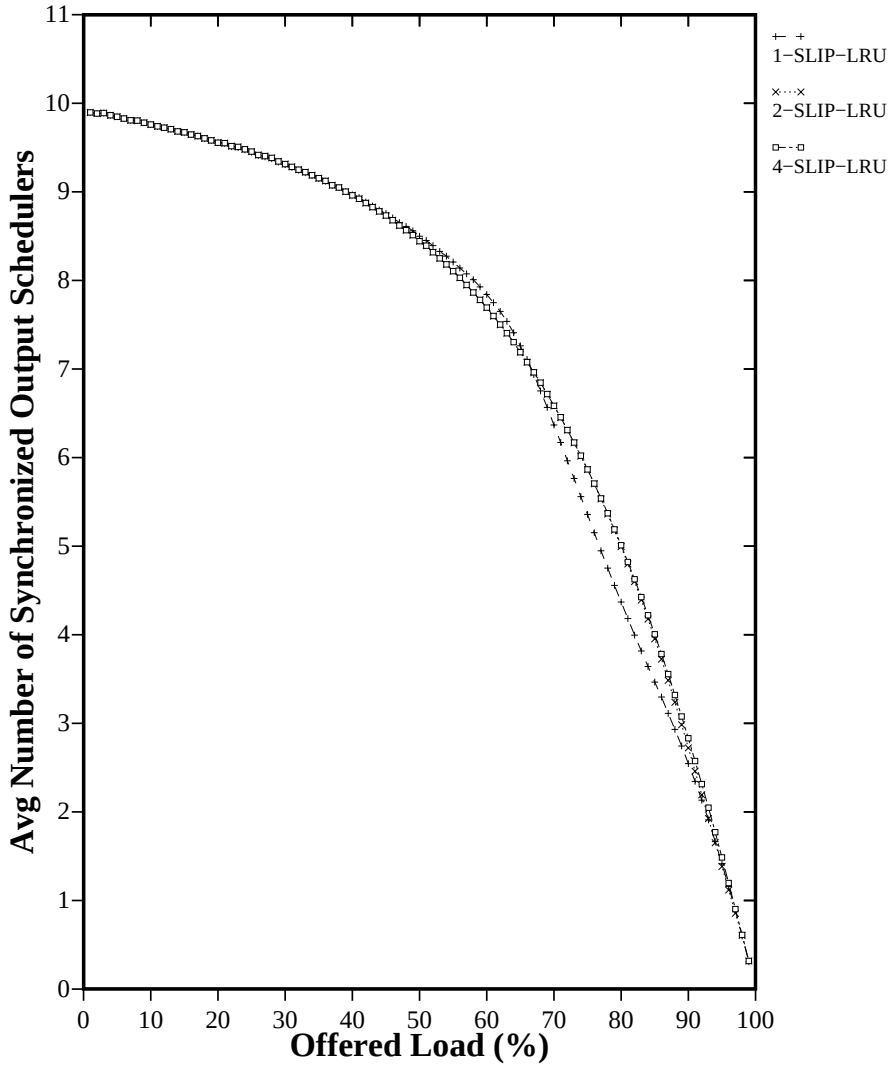


FIGURE 3.10 *i*-SLIP-LRU algorithm under uniform i.i.d. Bernoulli traffic. For 1 iteration, the number of desynchronized schedulers is almost identical to 1-SLIP. However, the number of schedulers increases only slightly with the number of iterations. This is quite different from i-SLIP for 2 and 4 iterations (see Figure 3.7)

when input 1 received a grant, it never accepted it. If we can encourage input 1 to accept output 2 if it has not done so recently, then we can prevent the connection from being starved. One way to achieve this is for the input arbiter to give highest priority to the least recently used (LRU) output[1]. We call this algorithm “iterative SLIP output arbiters with LRU input arbiters” (*i*-SLIP-LRU).

The *i*-SLIP-LRU algorithm allows us to update the pointers after each iteration so that every established connection will help desynchronize the output arbiters. Figure 3.10 demonstrates that

1. Starvation could also be avoided by using random selection at the input arbiter.

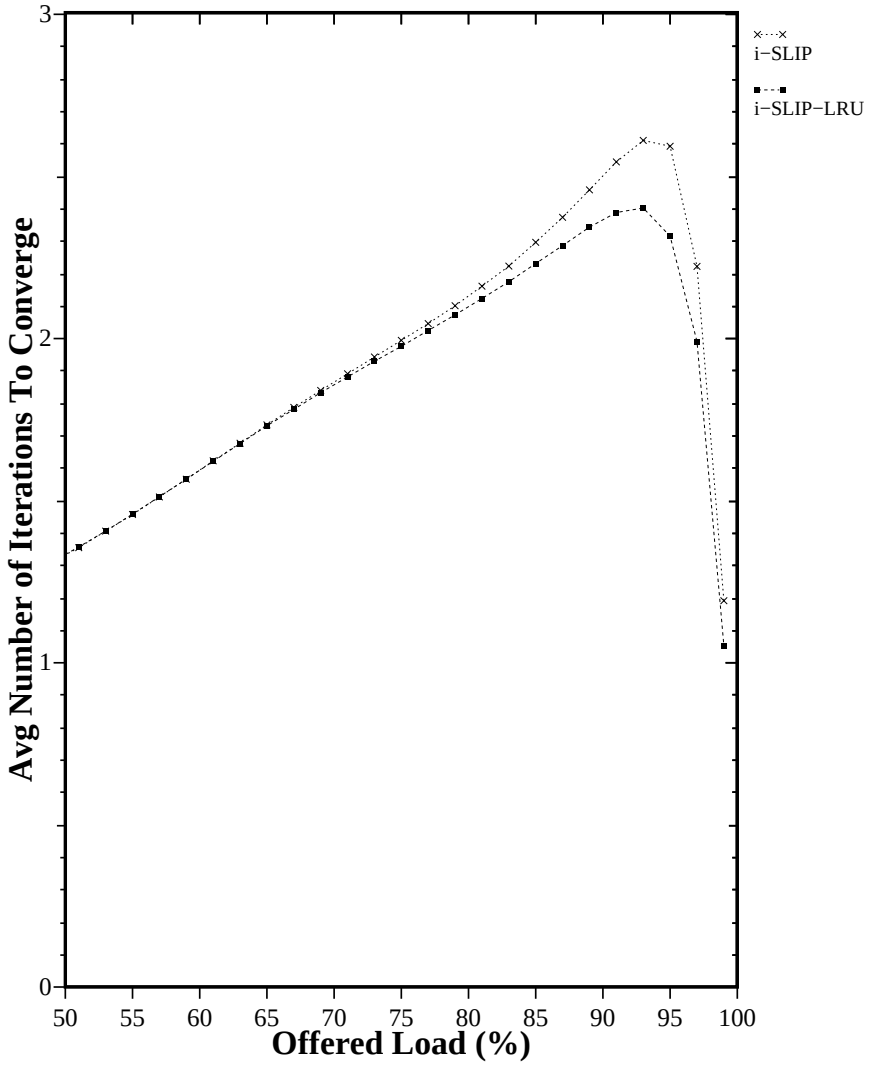


FIGURE 3.11 An example of the average number of iterations for *i*-SLIP-LRU to converge for uniform i.i.d. Bernoulli traffic as a function of the offered load. The algorithm is run to completion during each cell time to determine how many iterations are performed before no more connections can be added.

unlike *i*-SLIP, the number of synchronized schedulers for *i*-SLIP-LRU increases only very slightly with the number of iterations. This is beneficial in two ways: it means that for a fixed number of iterations, *i*-SLIP-LRU will provide higher performance, or alternatively, if run to completion *i*-SLIP-LRU will converge faster (Figure 3.11).

4.2 Separate Pointers for each Iteration

An alternative way to prevent starvation in iterative SLIP is to maintain a separate pointer for each iteration. For *n*-SLIP, output *j* now maintains *n* grant pointers, $g_j(1)$, …, $g_j(n)$. During the first iteration, the arbiter uses $g_j(1)$, updating the pointer if and only if a connection is established in this iteration. In the next iteration, the arbiter uses $g_j(2)$, and so on. In other words, the arbiter pointer is updated at the end of every iteration and for every connection.

We now show that no queue is starved of service by this algorithm. Assume that the queue Q(*i*,*j*) at input *i* is non-empty and thus requests service from output *j*. Now assume that we are in iteration *k*. Output *j* is using the pointer $g_j(k)$, and we shall assume that currently $g_j(k) = i$. During iteration *k* of every cell time, output *j* will grant to input *i* until either (i) input *i* accepts output *j* and Q(*i*,*j*) is served, or (ii) output *j* serves Q(*i*,*j*) in iteration $k' \neq k$, and Q(*i*,*j*) becomes empty. If Q(*i*,*j*) does not become empty in iteration k', then eventually it will be served in iteration *k* and $g_j(k)$ will be updated. So, in either case, Q(*i*,*j*) is served and $g_j(k)$ will eventually move. This means every non-empty queue will eventually be served and none will be starved of service indefinitely.

The advantage of this algorithm is that the pointers at each iteration tend to become desynchronized. More importantly, every connection that is established helps desynchronize the arbiters.

We find that for the arrival processes described in Section 3, this variation on the basic SLIP algorithm does *not* improve performance. It therefore does not seem worth the extra complexity of maintaining multiple pointers at each output. We believe that this requires further study.

5 Implementing Iterative SLIP

To conclude the description of *i*-SLIP, we consider the complexity of implementing the algorithm in hardware. Implementation of *i*-SLIP is very similar to non-iterative SLIP, described in Chapter 2 Section 7. Figure 3.12 shows how for an NxN switch, 2N arbiters and an N^2-bit memory

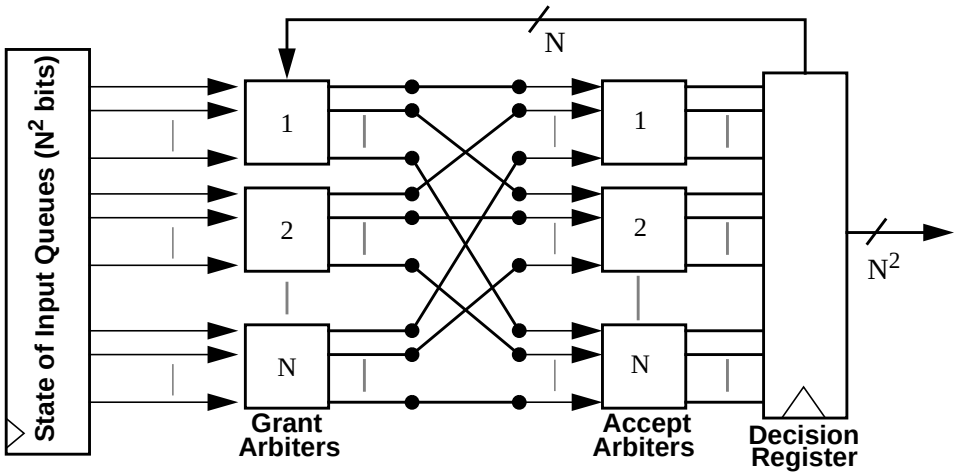


FIGURE 3.12 Interconnection of 2N arbiters to implement *i*-SLIP for an NxN switch.

may be interconnected to implement *i*-SLIP. The arbiters are almost identical to those used for non-iterative SLIP. They differ in two ways:

1. When an input or output is matched, its arbiter is disabled in subsequent iterations, preventing it from making additional matches. This is simple for the accept arbiter: whenever it makes a decision, it is disabled in all further iterations. However, it is not known whether a grant arbiter's decision is accepted until the end of the iteration. The decision register must feedback an indication to the grant arbiters. This is shown in Figure 3.12.
2. The arbiters only update pointers a_i and g_i after the first iteration.

As before, the complexity of *i*-SLIP is dominated by the arbiters. In fact, the number of gates required to implement *i*-SLIP is almost identical to non-iterative SLIP.

In some implementations it may be desirable to reduce the number of arbiters, sharing them among the grant and accept steps of the algorithm. An implementation using only N arbiters is shown in Figure 3.13. The results from the arbiters in the grant phase are registered and fed back for the accept phase. The number of gates for this implementation is almost halved, but with the performance penalty of an extra clock delay through the holding register.

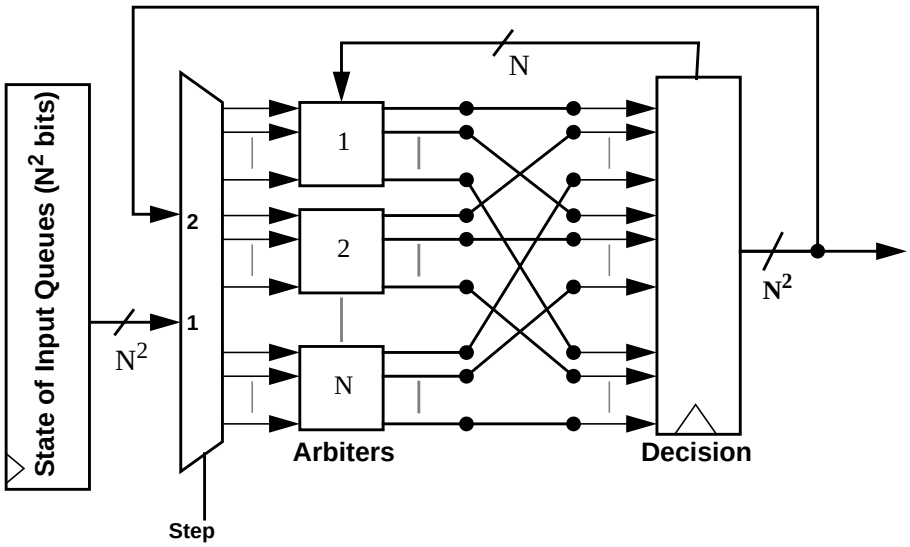


FIGURE 3.13 Interconnection of N arbiters to implement i-SLIP for an NxN switch. Each arbiter is used for both input and output arbitration. In this case, each arbiter contains *two* registers to hold pointers g_i and a_i.

CHAPTER 4

Weighted Matching Algorithms

1 Introduction

In this chapter we describe algorithms that consider more than one bit of information per queue, for example the occupancy of the queue, or the waiting time of queued cells. These algorithms find the maximum or maximal *weight* matching, giving preference to queues with a larger occupancy, or to cells that have been waiting longest.

We saw in Chapter 1 that maximizing the size of the match is not necessarily desirable as this can lead to instability for an offered load below capacity, and can lead to starvation for an offered load above capacity. This was demonstrated in Chapter 2 where we considered simple arrival patterns that are unstable for both the *maxsize* and SLIP algorithms, even for a 2x2 switch. The reason that these algorithms become unstable is that they only consider one bit of information per input queue: whether the queue is empty or non-empty. As we shall see, the maximum weight algorithms are stable over a wider range of workloads.

We start by describing two maximum weight matching algorithms, *longest queue first* (LQF) and *oldest cell first* (OCF) and consider their performance. We prove that the LQF algorithm is stable under i.i.d. arrivals and conjecture that both algorithms, although too complex to implement in hardware, are stable under all admissible offered loads.

We describe the more practical, parallel and iterative algorithms, *i*-LQF and *i*-OCF which attempt to find maximal weight matchings in a similar manner to *i*-SLIP and present schematic implementations of both algorithms.

Finally, we describe an interesting class of algorithms that solve the *stable marriage problem*. Solutions to this well-studied problem find a special kind of weighted bipartite matching called a *stable* matching. Although stable matchings are generally different from either a maximum sized or maximum weight match, they provide good performance and are readily implemented in hardware.

2 Maximum Weight Matching

Figure 1.2 shows an example of a matching on a weighted bipartite graph. The maximum *weight* matching, M is one that maximizes $\sum_{(i,j) \in M} w_{i,j}$, where $w_{i,j}$ is the weight assigned to the edge between vertices i and j. As with the maximum size matching, the maximum weight matching for a bipartite graph can be found by solving an equivalent network flow problem. The most efficient known algorithm for solving this problem converges in $O(N^2 \log_3 N)$ running time [41].

We now consider two types of maximum weight matching that may be used to schedule cells in an input-queued switch: LQF and OCF.

In the LQF algorithm, preferential service is given to input queues that are more heavily occupied. As illustrated in Figure 4.1, this is achieved by defining $w_{i,j}(t)$ to be equal to the queue occupancy $L_{i,j}(t)$.

The OCF algorithm gives preferential service to cells that have been queued for a long time. This is achieved by defining $w_{i,j}(t)$ to be equal to the waiting time $W_{i,j}(t)$ of the cell at the head of queue $Q(i,j)$.

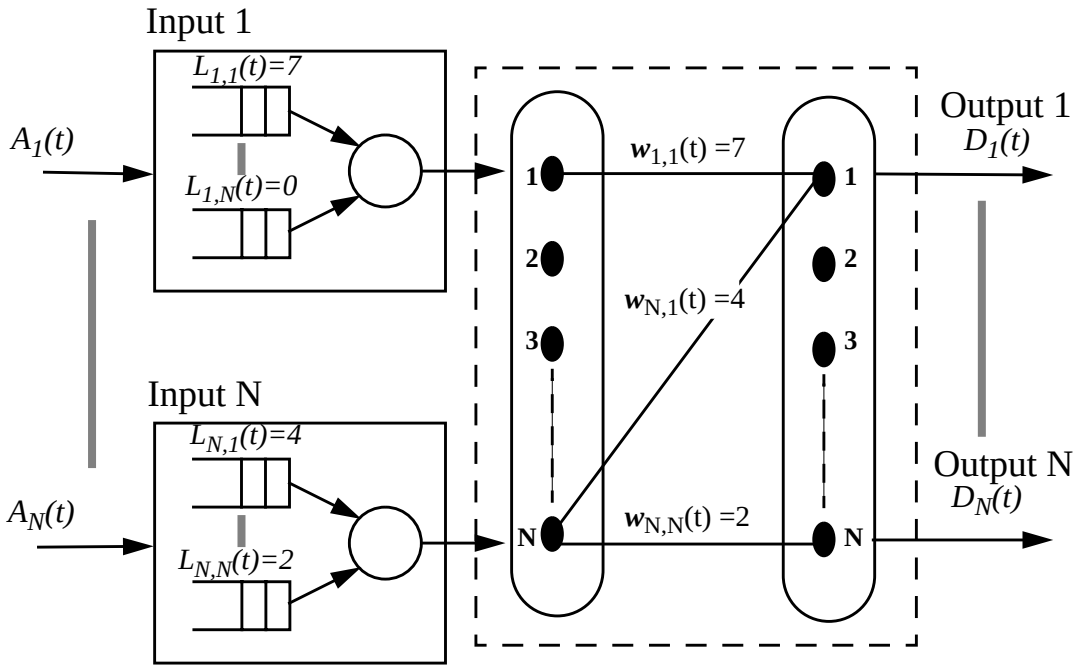


FIGURE 4.1 Example of weights for the LQF maximum weight matching algorithm.

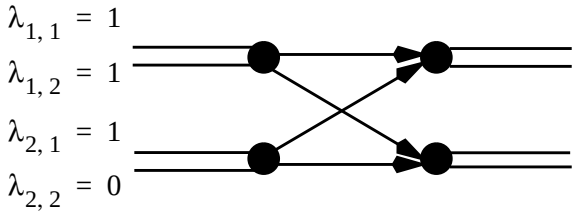


FIGURE 4.2 Example of 2x2 switch for which, using the LQF algorithm, inadmissible traffic may lead to the starvation of an input queue.

2.1 Starvation with LQF

Under inadmissible traffic it is possible for the LQF algorithm to permanently starve an input queue. As a simple example, consider the 2x2 switch shown in Figure 4.2. Three queues have an arrival rate equal to their capacity and will be unstable, growing without bound. Now assume that

all three queues have grown to a length of two cells. Further assume that a single cell arrives at the fourth queue, *Q(2,2)*, but that no further cells arrive at this queue. Because of the large arrival rate, the other queues will never contain fewer than two cells and so *Q(2,2)* will never be served.

OCF, however, cannot starve a queue under any offered load. Cells at the head of queues that have not been served recently increase in weight until, eventually, they are served.

2.2 Performance of LQF and OCF Algorithms

2.2.1 Uniform Workload

If all switch inputs and outputs are identically loaded, the LQF algorithm has an *average* performance identical to the *maxsize* algorithm, Figure 4.3(a). This is because there is no benefit, on average, in distinguishing between different input queues when they all have identical average arrival rates. We have found this to be the case for a range of uniform workloads with and without correlated arrivals. The OCF algorithm has a slightly worse average behavior than LQF.

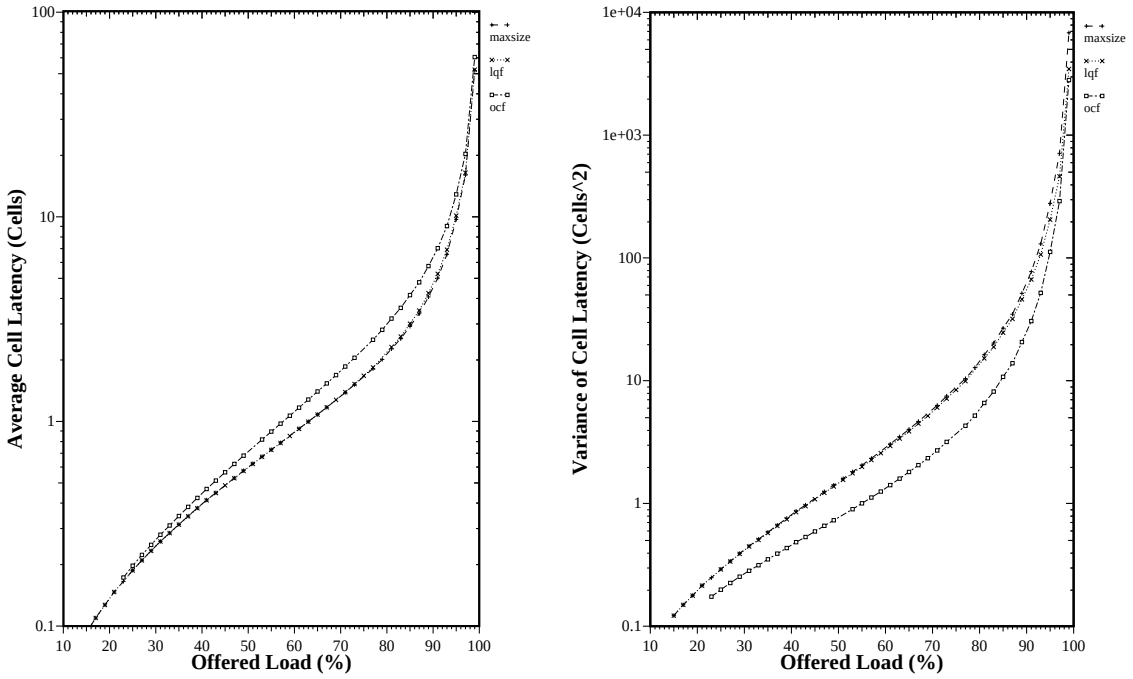
However, because during fluctuations in waiting time it favors cells that have been waiting longest, the *variance* in cell latency is lower for the OCF algorithm. This is illustrated in Figure 4.3(b), where the variance of cell latency is plotted against the offered load.

2.2.2 Non-Uniform Workload

The difference between the maximum weight and maximum size matching algorithms is more marked under a non-uniform workload, particularly when not all flows are active.

We illustrate this by way of a simple example: an arbitrary arrival pattern for a 4x4 switch, shown in Figure 4.4. We find that for the *maxsize* algorithm the switch is unstable for $\lambda > 0.31$, whereas for the LQF and OCF algorithms, the switch is stable for all admissible values of λ.

The LQF and OCF algorithms maintain much closer average queue lengths than the *maxsize* algorithm. Figure 4.5 shows the average cell latency through each of the 16 input queues in the 4x4 switch of Figure 4.4, with all average rates $\lambda = 0.30$. Even though the *maxsize* algorithm is stable for this workload, the average queue lengths differ widely. This difference increases as the



(a) Average cell latency vs. offered load. (b) Variance in cell latency vs. offered load.

FIGURE 4.3 16x16 switch scheduled using the LQF, OCF and *maxsize* scheduling algorithms.

All flows have identical average arrival rate $\lambda = \lambda_{i,j}$ such that $\sum_{i \in I} \lambda_{i,j} < 1, \sum_{j \in J} \lambda_{i,j} < 1$

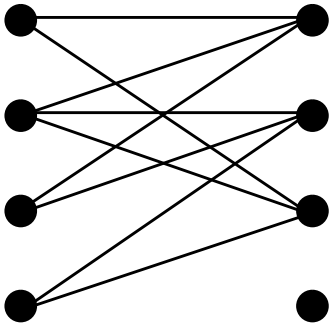


FIGURE 4.4 Example of 4x4 switch with non-uniform traffic pattern. Although admissible, this traffic pattern can be unstable for the maximum scheduling algorithm. It is stable for the LQF and OCF algorithms.

offered load increases, and it is the three queues $Q(2, 1)$, $Q(2, 2)$, $Q(2, 3)$ that become unstable

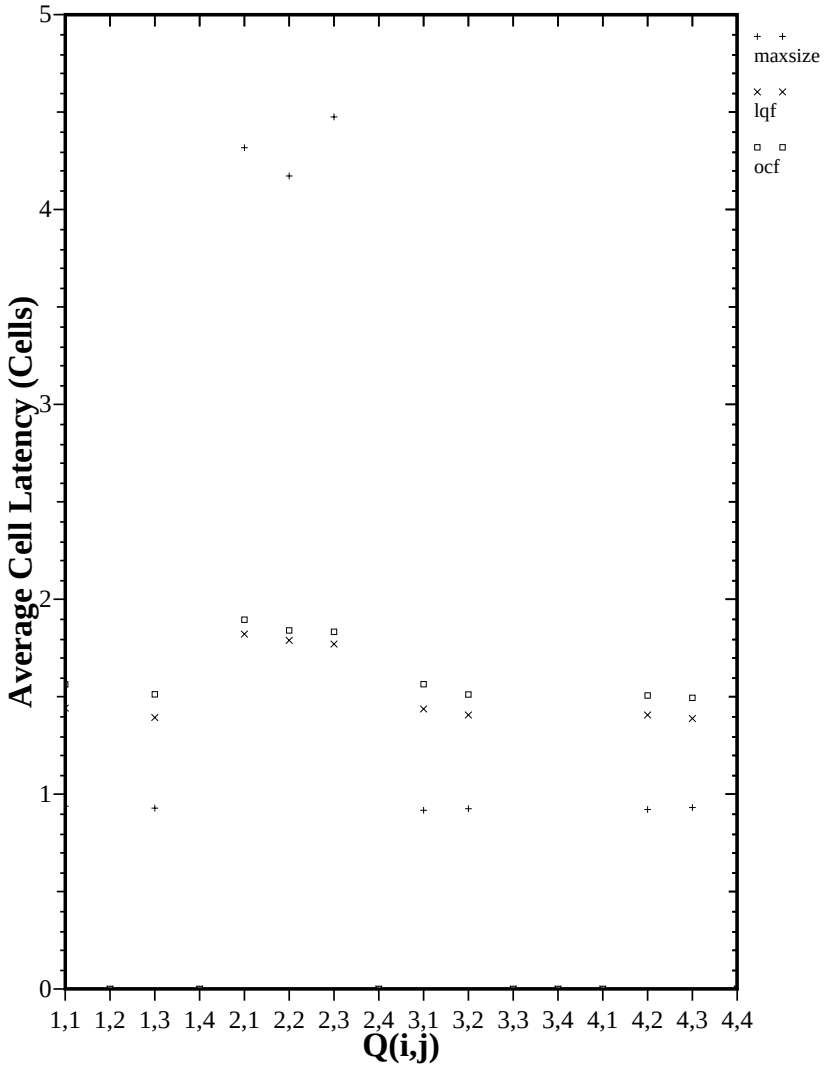


FIGURE 4.5 Variation in queue lengths under the non-uniform workload shown in Figure 4.4, with i.i.d. Bernoulli arrivals and $\lambda = 0.30$.

when the average arrival rate exceeds $\lambda = 0.31$. Both LQF and OCF maintain average queue lengths that are almost identical independent of the arrival rate.

2.2.3 Stability of 2x2 Switch

In this section we consider the stability of a 2x2 switch, scheduled with a maximum weight matching algorithm.

We start with a simple observation: if an arrival pattern is such that all input queues are permanently occupied, the switch will be stable. This is because the switch can serve two queues in every cell time.

Theorem 4.1: *If* $L_{1,1}(n), L_{1,2}(n), L_{2,1}(n), L_{2,2}(n) > 0$ *for all n, LQF and OCF are stable for a 2x2 switch.*

Proof: Assume that at time *n* the switch is stable. i.e.

$$L(n) \equiv L_{1,1}(n) + L_{1,2}(n) + L_{2,1}(n) + L_{2,2}(n) < \infty \quad (1)$$

Because $L_{1,1}(n), L_{1,2}(n), L_{2,1}(n), L_{2,2}(n) > 0$, the LQF and OCF algorithms will always serve exactly 2 queues. There can be no more than two arrivals to the switch in a cell time, so

$$L(n+1) \leq L(n) \quad (2)$$

and so the switch must be stable at time *n*+1. Hence $L(n)$ cannot become unbounded, and the switch is stable. ❒

In general, not all input queues will be permanently occupied. At any time, one or more queues may be empty and, as a result, the scheduling algorithm may select to serve fewer than two queues. Recall that it was when one queue was empty that the maximum size matching algorithm (and SLIP and PIM) could become unstable. However, below we show that for a 2x2 switch LQF and OCF are stable when one or more queues are permanently empty.

Theorem 4.2: *LQF and OCF are stable for the 2x2 switch with three active flows illustrated in Figure 4.6 and independent arrivals.*

Proof: Appendix 3 Section 1 finds sufficient conditions on a scheduling algorithm so that this switch is stable for independent arrivals:

1. If $L_{1,1}(n) = 0$, then set crossbar to configuration *B*.
2. Else, if $L_{2,1}(n) = 0$ and/or $L_{1,2}(n) = 0$, then set crossbar to configuration *A*.
3. Else, set crossbar configuration to either *A* or *B*.

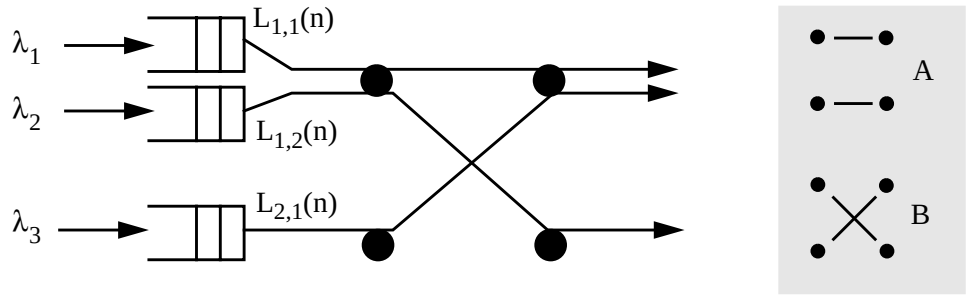


FIGURE 4.6 2x2 switch with three active flows. The two possible switch configurations, *A* and *B*, are shown.

LQF clearly satisfies conditions (1) and (3) above, but not (2). Define an algorithm, *G*, that is exactly the same as LQF under conditions (1) and (3), but also satisfies condition (2). Clearly, *G* is stable. LQF serves $Q(1, 2)$ and $Q(2, 1)$ at least as often as *G*, so under LQF these two queues must be stable. It remains to be shown that $Q(1, 1)$ is stable under LQF. Assume that under LQF, $Q(1, 1)$ is *unstable*. For this to be the case $Q(1, 1)$ must grow so that at some time

$$L_{1,1}(n) = 1 + L_{1,2}(n) + L_{2,1}(n), \tag{3}$$

and $Q(1, 1)$ will then be served continuously until

$$L_{1,1}(n) \le L_{1,2}(n) + L_{2,1}(n). \tag{4}$$

During the time that $Q(1, 1)$ is served continuously, its queue length cannot increase. Therefore, $L_{1,1}(n)$ is bounded by $1 + L_{1,2}(n) + L_{2,1}(n)$. $Q(1, 2)$ and $Q(2, 1)$ are stable, so $1 + L_{1,2}(n) + L_{2,1}(n)$ is bounded, which means that $Q(1, 1)$ is stable. An analogous argument based on waiting times holds for OCF. ❒

It is interesting to consider why *maxsize* can be unstable for this switch (Chapter 1, Section 3.3). Although the conditions above are not strictly necessary for the switch to be stable, they give an intuitive explanation. If condition (2) above is true, the *maxsize* algorithm may select either configuration *A* or *B*, even if one of the queues grows without bound.

Theorem 4.3: *For any arrival process to a 2x2 switch, if for some n*

$$\left| \{L_{1,1}(n) + L_{2,2}(n)\} - \{L_{1,2}(n) + L_{2,1}(n)\} \right| \leq 3 \tag{5}$$

then for all $n' \geq n$

$$\left| \{L_{1,1}(n') + L_{2,2}(n')\} - \{L_{1,2}(n') + L_{2,1}(n')\} \right| \leq 4\,. \tag{6}$$

Proof: See Appendix 3 Section 2. ❒

To summarize these results for a 2x2 switch:

1. For any arrival process, it is not possible for all 4 input queues to become unstable simultaneously.
2. When arrivals are i.i.d. and only three flows are active, it is not possible for any queue to become unstable.
3. For any arrival process, if the queues are initially empty, the occupancies of the two sets of input queues: $L_{1,1}(n) + L_{2,2}(n)$ and $L_{1,2}(n) + L_{2,1}(n)$ can differ by at most four cells. This means that if instability occurs, at least one queue from each set must be unstable.

This leads us to make the following conjecture, as yet unproved.

Conjecture: *For a 2x2 switch, LQF and OCF are stable for all ergodic, admissible arrival processes.*

2.2.4 Stability of NxN Switch

Theorem 4.4: *LQF is stable for all admissible i.id. arrival processes.*

Proof: The proof is given in appendix 4. In summary, we show that for an NxN switch scheduled using the LQF algorithm, there is a negative single-step drift in the sum of the squares of the state. In particular,

$$\mathrm{E}\left[\underline{L}^T(n+1)\underline{L}(n+1) - \underline{L}^T(n)\underline{L}(n) \middle|\ \underline{L}(n)\right] \leq -\varepsilon\|\underline{L}(n)\| + k\,, \tag{7}$$

$k > 0, \varepsilon > 0$. ❒

The term $-\varepsilon\|\underline{L}(n)\|$ indicates that whenever the occupancy of the input queues is large enough, the expected drift is negative. Should $\|\underline{L}(n)\|$ become very large, the downward drift also becomes large, and so the stability is quite "strong". This leads us to the following conjecture.

Conjecture: *LQF and OCF are stable for all ergodic, admissible arrival processes.*

One possible definition of stability for ergodic arrivals is

$$\lim_{T \to \infty} \frac{1}{T} \mathrm{E}\left[\sum_{n=0}^{T} \|\underline{L}(n)\|\right] < \infty . \quad (8)$$

Although we have not been able to find admissible arrival processes for which an NxN switch is unstable using the LQF and OCF, we have not been able to prove that this conjecture is true in general. This remains an open problem.

3 Iterative Maximal Weight Matching Algorithms

A goal of this work is to determine fast scheduling algorithms that can be readily implemented in hardware. Unfortunately, the maximum weight matching algorithms, LQF and OCF, are very complex to implement, and require an $O(N^2 \log_a N)$ running time.

As an alternative, we now consider iterative approximations to LQF and OCF, based on the SLIP and PIM algorithms, that are designed to be readily implemented in hardware and to quickly find a maximal weight match. These algorithms are called respectively, *i*-LQF and *i*-OCF.

3.1 *i*-LQF

Like PIM and SLIP, *i*-LQF is an iterative algorithm consisting of N output and N input arbiters operating in parallel. The scheduler maintains an N^2 word memory; each entry indicates the occupancy of an input queue, $L_{i,j}(t)$. The word width, *b*, is determined by the maximum queue length, L_{max}

$$2^b \geq L_{max} . \quad (9)$$

As before, at the beginning of each cell time the match process begins over. All inputs and outputs are initially unmatched and only those inputs and outputs not matched at the end of one iteration are eligible for matching in the next. Connections made in one iteration are never removed by a later iteration, even if a larger sized match would result. The three steps of each iteration are as follows:

Step 1. *Request*. Each unmatched input sends a request word of width 2^b bits to each output for which it has a queued cell, indicating the number of cells that it has queued to that output.

Step 2. *Grant*. If an unmatched output receives any requests, it chooses the largest valued request. Ties are broken randomly.

Step 3. *Accept*. If an unmatched input receives one or more grants, it accepts the one to which it made the largest valued request. Ties are broken randomly.

3.2 Properties

The *i*-LQF algorithm has the following properties:

Property 1. Independent of the number of iterations, the longest input queue is always served. The longest input queue will lead to the largest request in the first iteration, which must be granted, resulting in the largest grant. This must be accepted.

Property 2. As with *i*-SLIP, the algorithm converges in at most N iterations. If during some iteration no connection is made, the algorithm has converged and no further connections are possible. So, prior to convergence at least one connection is added per iteration. There are N inputs and N outputs, requiring at most N iterations to converge.

Property 3. For an inadmissible offered load, an input queue may be starved. This is the same as for LQF, described in Section 2.1.

3.3 *i*-OCF

The *i*-OCF algorithm eliminates the starvation problem of *i*-LQF by favoring cells with a longer waiting time. *i*-OCF differs from *i*-LQF only in Step 1: the value of the request from input *i* to output *j* equals the waiting time, $W_{i,j}(t)$, of the cell at the head of queue $Q(i, j)$.

3.4 Properties

The *i*-OCF algorithm has the following properties:

Property 1. Independent of the number of iterations, the cell (*C)* that has been waiting the longest time in the input queues is served. First note that *C* must be at the head of its FIFO queue. The input will make the largest request to the scheduler in the first iteration on behalf of *C,* which must be granted, resulting in the largest grant. This must be accepted.

Property 2. As with *i*-LQF, the algorithm converges in at most N iterations.

Property 3. No input queue can be starved indefinitely.

3.5 Performance of *i*-LQF and *i*-OCF

3.5.1 Uniform Workload

Both *i*-LQF and *i*-OCF have worse throughput-delay performance than LQF and OCF. This is to be expected — neither of the iterative algorithms will remove connections in an attempt to maximize the match. The performance of both algorithms under uniform i.i.d. Bernoulli arrivals is illustrated in Figure 4.7. Once again, both iterative algorithms are stable up to an offered load of 100%, albeit with a slightly larger latency than for the maximum weight algorithms.

3.5.2 Nonuniform Workload

We have not found a workload for which the *i*-LQF and *i*-OCF are unstable; under non-uniform arrival patterns, both algorithms exhibit similar throughput-delay performance to the LQF and OCF algorithms. We illustrate this by way of the same simple (but arbitrary) example as before, shown in Figure 4.4. Our results in Figure 4.8 show not only that the *i*-LQF and *i*-OCF algorithms are stable, but that their performance is very close to the maximum weight algorithms.

3.5.3 Stability for 2x2 Switch

Conjecture: *With sufficient iterations, i-LQF and i-OCF are stable for a 2x2 switch with all admissible workloads.*

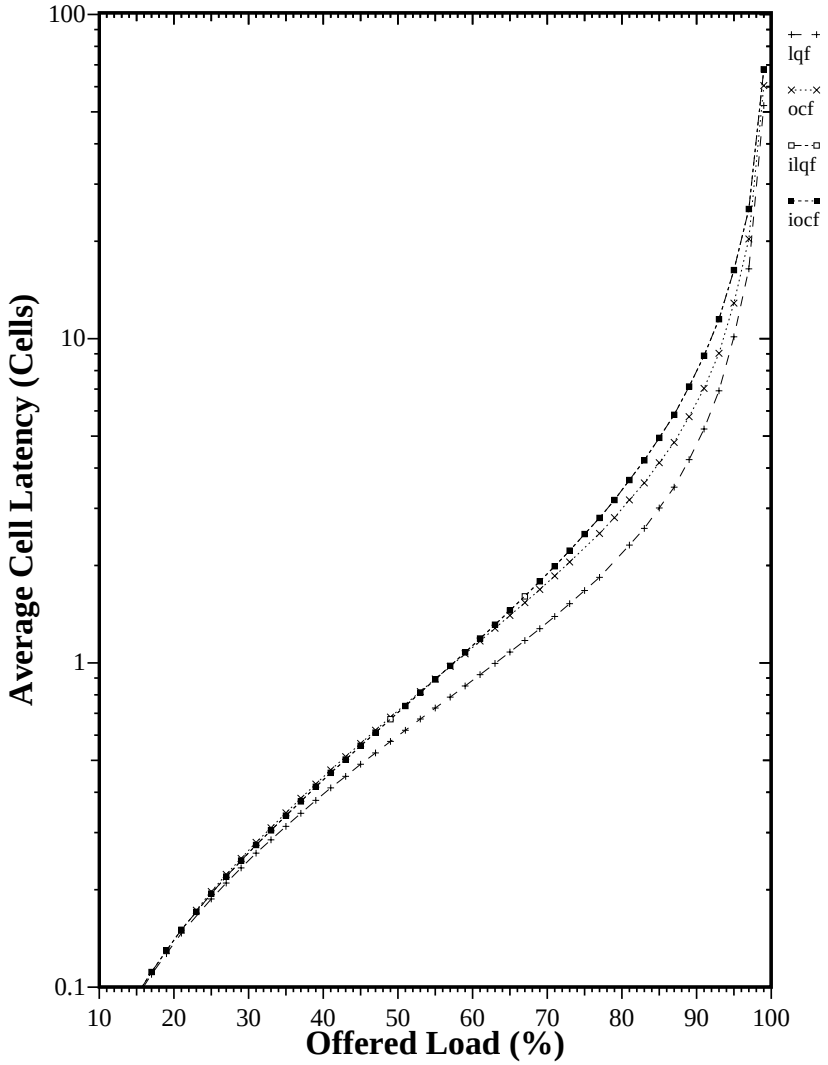


FIGURE 4.7 Performance of *i*-LQF and *i*-OCF algorithms for uniform i.i.d. Bernoulli arrivals, compared with LQF and OCF algorithms.

3.6 Implementation of *i*-LQF and *i*-OCF

Both *i*-LQF and *i*-OCF are relatively simple to implement in hardware, although more complex than the *i*-SLIP algorithm described in Chapter 3. The main difference is that the simple priority encoders that perform arbitration in SLIP are replaced by more complex comparators.

Figure 4.10 shows a schematic design for *i*-LQF. The design consists of 2N arbiters.[1] Each

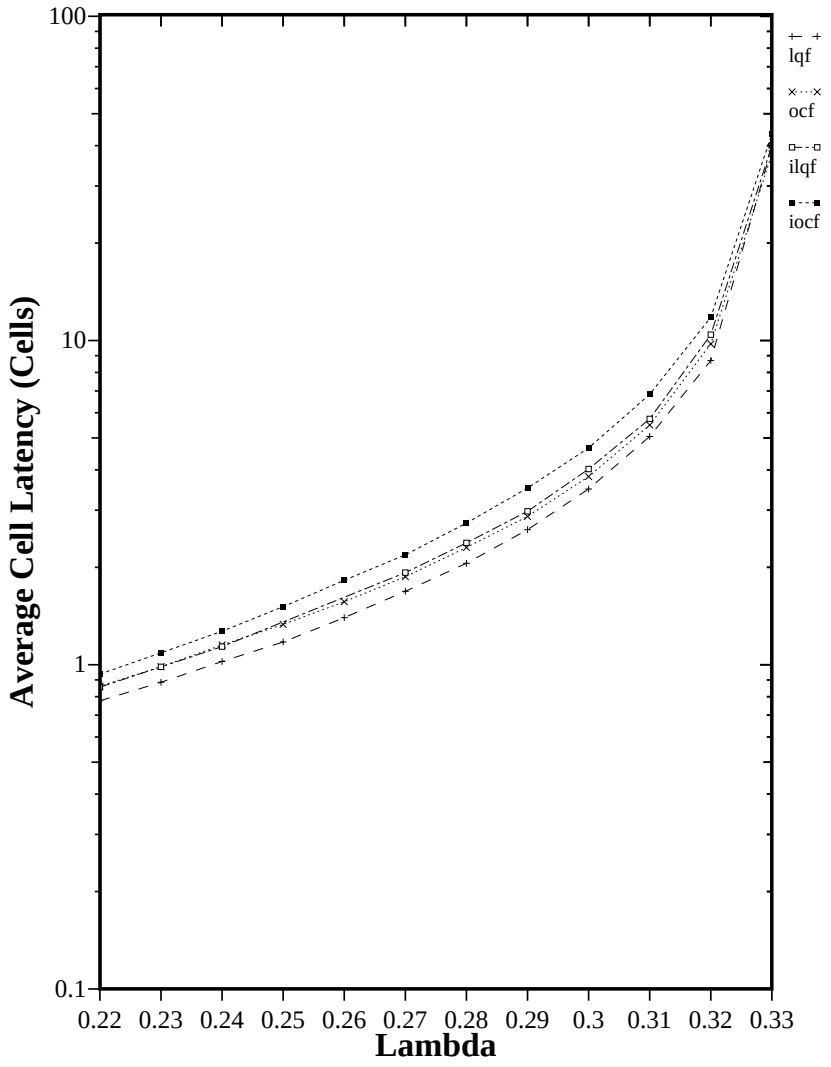


FIGURE 4.8 Performance of *i*-LQF and *i*-OCF algorithms under the non-uniform workload, shown in Figure 4.4.

For *i*-LQF, the registers for output *j* maintain the occupancy values for each input queue $Q(i \in I, j)$ and similarly, the registers at input *i* maintain the occupancy values for each input queue $Q(i, j \in J)$. If output *j* grants to input *i*, then grant arbiter *j* enables the register containing $L_{i,j}(t)$ at accept arbiter *i*. The finite state machine prevents matched inputs and outputs from par-

1. As with SLIP, the arbiters may be reduced to N by sharing them between the grant and accept stages.

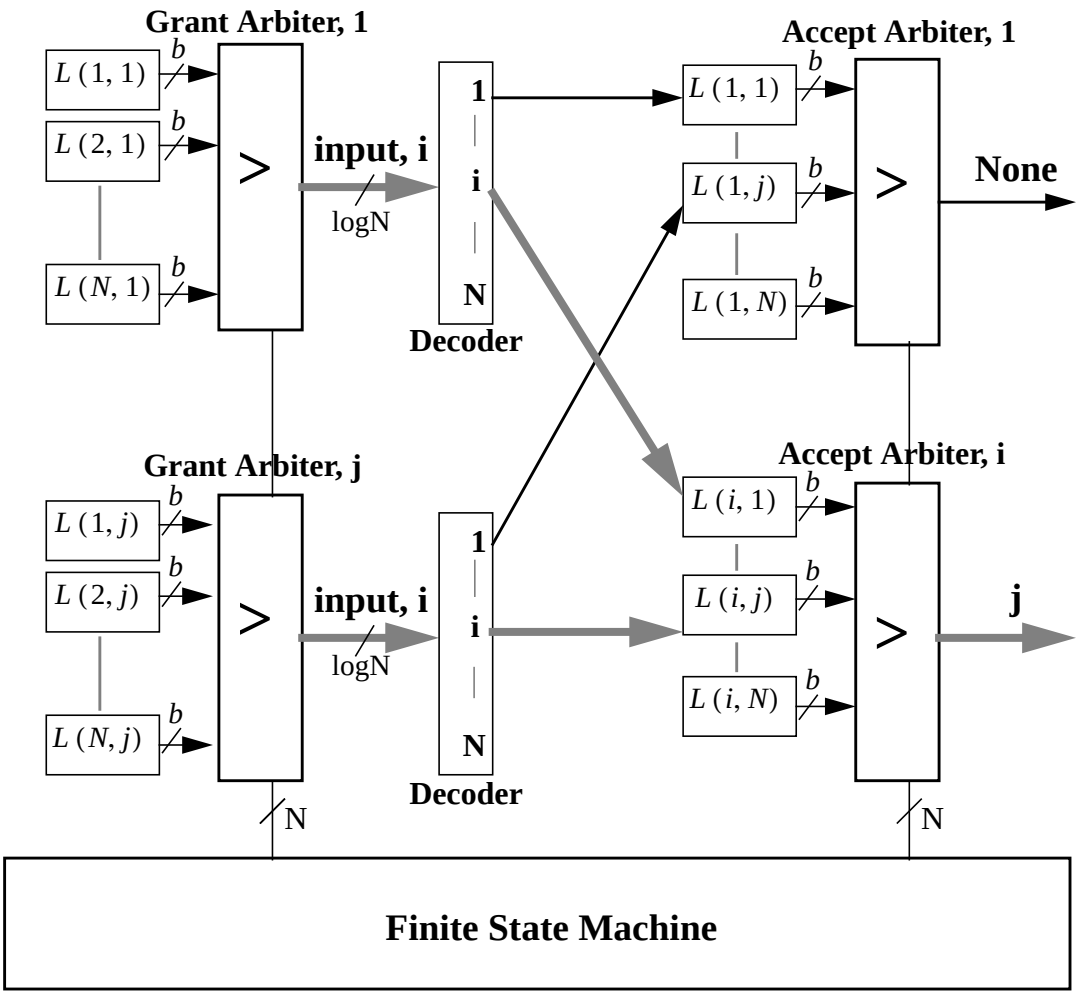


FIGURE 4.9 Implementation of NxN *i*-LQF algorithm using 2N arbiters. For brevity, only grant arbiters 1 and *j* and accept arbiters 1 and *i* are shown here.

ticipating in future iterations by disabling the corresponding arbiters. The implementation for *i*-OCF is exactly the same as for *i*-LQF, except that the registers hold cell waiting times, rather than queue occupancies.

The implementation for *i*-LQF is clearly more complex than for *i*-SLIP, both in the number of gates required to implement the arbiters and in the size of the registers required to hold the queue occupancy values. *i*-LQF also requires more state to be updated at the beginning and end of each cell time. In a cell time, at most one queue at each input can increase its occupancy and by only one cell. The input must indicate to the central scheduler which queue has increased; the scheduler

must increment the corresponding value. Likewise, in a cell time, at most one queue at each input may decrease its occupancy, and by only one cell. The input does not need to indicate which queue has changed: it is the one that the scheduler selected during its arbitration. The scheduler must decrement the corresponding value.

Choosing the value for *b* is an important design decision, affecting the number of different queue lengths that can be distinguished. If the size of the input queues is large, or if the gate count is to be minimized, it may be required that $2^b < L_{max}$. Two possible modifications to the algorithm in this case are: (1) if the occupancy $L_{i,j}(t) \geq 2^b$, issue a request of size 2^b; or (2) if the maximum queue size $L_{max} = 2^b \times 2^n$, issue a request of size $L_{i,j}(t)/2^n$, which is readily achieved in hardware by truncating the lowest *n*-bits.

It is interesting to note that although for small values of N and *b* the complexity of *i*-LQF is greater than for *i*-SLIP, its complexity increases more slowly with N. In Chapter 2 we found that the number of gates required to implement *i*-SLIP increases with N^4. *i*-LQF consists of 2N comparators, each comprising $O(\log b \log N)$ gates and N^2 registers each with $O(b)$. For small *b* and N, the comparators will dominate the total number of gates, increasing with $N \log N$. With sufficiently large N, the total gate count is dominated by the registers which increase with N^2. In both cases, this increase is at a slower rate than for *i*-SLIP.

4 Stable Marriages

The *stable marriage problem* was first introduced in 1962 by Gale and Shapley [13] and since then has been studied extensively [15], [26]. Solutions to the stable marriage problem find a *stable* and complete matching on a bipartite graph and can therefore be used to schedule cells in an input-queued switch. More importantly, there exists a well known algorithm (the Gale-Shapley Algorithm — GSA) that is feasible to implement in hardware and will always find a stable matching in *N* iterations.

We begin this section with a description of the stable marriage problem and describe the GSA. We then consider two different ways that the GSA can be used to schedule cells in an input-queued switch; one is a variation of LQF and the other a variation of OCF. We finish the section with a description of the implementation of these algorithms.

4.1 The Stable Marriage Problem

Although rather dated, the classical problem is stated in terms of two equal-sized sets: a set of N men and a set of N women all of whom wish to get married to a member of the other set. Each man independently creates an ordered preference list, ranking each of the N women. Each woman does the same, ranking each of the N men. The aim is to find a *stable* matching between the set of men and women so that each man is matched to a woman and each woman is matched to a man. A match is *unstable* if there is a couple who are not matched to each other, yet both prefer the other to their partner in the matching. A *stable* matching is any matching that is not *unstable*.

4.2 The Gale-Shapley Algorithm

In their original paper, Gale and Shapley prove that:

1. Every instance of the stable marriage problem admits at least one stable matching. They prove this with an algorithm, GSA, that will always find a matching in $O(N^2)$ running time.
2. The GSA has two distinct versions: the male-optimal GSA and the female-optimal GSA. The male-optimal GSA will simultaneously give all the men the best partner and all the women the worst partner that they could have in any stable matching; and vice-versa for the female-optimal GSA.

The Gale-Shapley Algorithm is usually expressed in terms of a series of marriage proposals. In the male-optimal version, the proposals are always from men to women. We will describe here the male-optimal version of the algorithm.[1] Initially, each person is *free* and may become *engaged* as the algorithm progresses. Women who become engaged never become free again, whereas engaged men may be rejected by their partner and become free again.

1. The female-optimal algorithm is obtained by simply reversing the roles of the sexes.

Each man proposes to the women, one at a time, in the order that they appear in his preference list. If the woman that he proposes to is free, she will accept his proposal. If she is already engaged, but prefers the new proposal, she will reject her previous partner in favor of the new man. The rejected man is now free and will resume making proposals to the remaining women in his preference list. The algorithm terminates when everyone is engaged.

The somewhat surprising findings of Gale and Shapley were that the algorithm will always converge before any man reaches the end of his preference list, and that on completion of the algorithm the engaged couples always form a stable matching.

4.3 Analogy to Switch Scheduling

A stable matching is an example of a bipartite graph matching. As described in Chapter 1, scheduling cells in an input-queued switch is analogous to finding a bipartite graph matching between the set of switch inputs and outputs. So if, for example, we assign the set of men to represent the switch outputs and the set of women to represent the switch inputs, a stable matching will represent a legal switch configuration.

Next we shall consider the principle ways that the GSA differs from the iterative algorithms discussed earlier in this thesis. We then describe two algorithms, GS-LQF and GS-OCF, that are based on the GSA and may be used for scheduling cells in an input-queued switch.

There are three main ways in which the GSA differs from the iterative maximum weight algorithms described earlier in this chapter:

1. A *stable* matching is in general different from a maximum sized or maximum weight matching. It is not clear that a stable matching will lead to an efficient use of switch bandwidth, or that it will prevent connections from being starved of service. In particular, we know that the GSA will favor either outputs or inputs. Later, we will consider an algorithm that provide a more egalitarian matching.
2. The stable marriage problem and the GSA are usually defined for a complete matching in which every input and every output is matched. It is not always the case in an input-queued switch that a complete match is possible: most often, only a subset of the input-queues are occupied. This means that some inputs and outputs will have missing entries in their preference lists. Fortunately, with a small modification, the GSA algorithm will

still work.[1] However, it is known that just a few missing elements in preference lists can lead to a large reduction in the size of a stable matching [15].

3. In the iterative algorithms *i*-SLIP, *i*-LQF and *i*-OCF, once a connection has been accepted it is not rejected by a later iteration. In the GSA, connections made in one iteration may be rejected by a later iteration. In principle, by rearranging connections the GSA can find a larger sized or weight match than iterative algorithms that do not remove connections established in an earlier iteration. The benefit of this is not clear: in (1) above we saw that the stable matching does no attempt to maximize either the size or the weight of a match.

4.3.1 The GS-LQF Algorithm

GS-LQF (GSA with LQF preference lists) is the Gale-Shapley algorithm with preference lists based on the occupancy of the input queues.

Output j determines its preference list $\mathbf{R}_{\mathrm{O}}(j)$ based on the occupancy of each of the N input queues, $L(i,j), \forall i \in \mathbf{I}$

$$\mathbf{R}_{\mathrm{O}}(j) = [i_1, i_2, \ldots, i_N], \quad \text{where: } L(i_1,j) \geq L(i_2,j) \geq \ldots \geq L(i_N,j). \tag{10}$$

Similarly, input i determines its preference list

$$\mathbf{R}_{\mathrm{I}}(i) = [j_1, j_2, \ldots, j_N], \quad \text{where: } L(i,j_1) \geq L(i,j_2) \geq \ldots \geq L(i,j_N). \tag{11}$$

4.3.2 The GS-OCF Algorithm

GS-OCF (GSA with OCF preference lists) is the Gale-Shapley algorithm with preference lists based on the waiting time of the cells at the head of each input queue.

Output j determines its preference list $\mathbf{R}_{\mathrm{O}}(j)$ based on the waiting times of the cells at the head of each of the N input queues, $W(i,j), \forall i \in \mathbf{I}$

$$\mathbf{R}_{\mathrm{O}}(j) = [i_1, i_2, \ldots, i_N], \quad \text{where: } W(i_1,j) \geq W(i_2,j) \geq \ldots \geq W(i_N,j). \tag{12}$$

Similarly, input i determines its preference list

$$\mathbf{R}_{\mathrm{I}}(i) = [j_1, j_2, \ldots, j_N], \quad \text{where: } W(i,j_1) \geq W(i,j_2) \geq \ldots \geq W(i,j_N). \tag{13}$$

1. It may be shown that the algorithm will partition the inputs and outputs into those that are matched in all stable matchings and those that are never matched [15].

4.4 Performance of GS-LQF and GS-OCF Algorithms

The performance of the GSA depends on the weight of the stable matching. In general, the relationship between a stable matching and a maximal weight matching is unknown. In fact, it is not intuitively obvious that a stable matching algorithm will perform well in this application, particularly when several of the input queues are empty.

Surprisingly, we have found through simulation that the performance of the GS-LQF and GS-OCF[1] algorithms are, respectively, *indistinguishable* from the performance of *i*-LQF and *i*-OCF. It remains an open problem whether the performance is identical for all traffic patterns.

4.5 Implementation of GS-LQF and GS-OCF

We now describe a parallel, iterative version of the GSA to match inputs to outputs in an input-queued switch. The algorithm is relatively simple to implement in hardware, although more complex than the *i*-SLIP algorithm described in Chapter 3.

Figure 4.10 shows the schematic design of GS-LQF which is almost identical to the implementation of *i*-OCF in Figure 4.10. As with *i*-LQF, the design consists of 2N arbiters. Each grant arbiter maintains a register value for each of the requesting input queues. For GS-LQF, the registers at output *j* maintain the preference list: the occupancy values for each input queue, $\mathbf{R}_O(j)$ in Equation 11. Similarly, the registers at input *i* maintain the elements of $\mathbf{R}_I(i)$. The Finite State Machine controls the preference lists at each arbiter by enabling only those that are active in a particular iteration.

In the first iteration, all of the entries in the arbiters' preference lists are enabled. Each output will grant ("propose") to the input which makes the largest request. For example, in Figure 4.10, grant arbiters 1 and *j* both grant to input *i*. Grants are single bit values, used to enable entries in an accept arbiter's preference list. The input may or may not have already accepted a grant. If it has not, then it accepts the largest enabled entry in its preference list. If it has, then it only accepts the new value if it exceeds the value of the previously accepted grant.

1. Our simulations only considered the output-optimal algorithms.

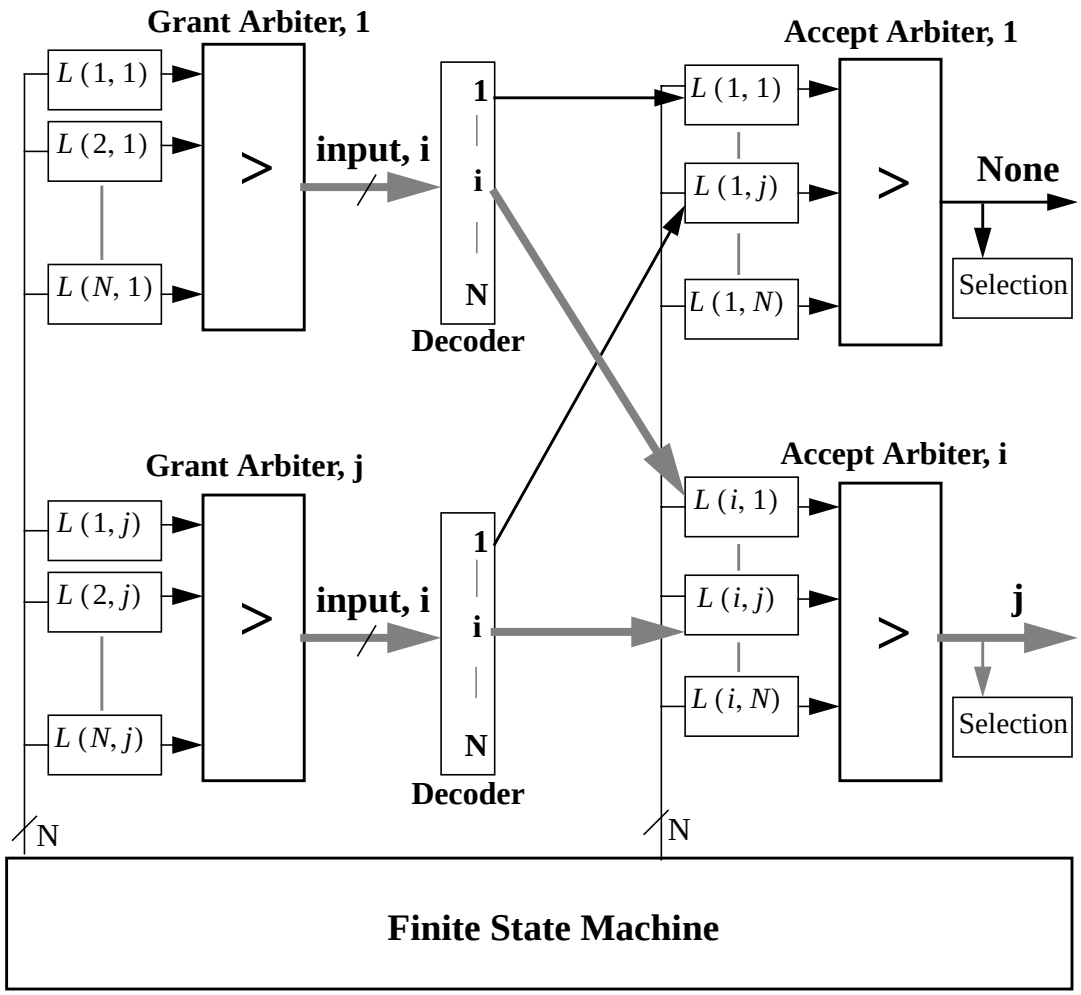


FIGURE 4.10 Implementation of NxN GS-LQF algorithm using 2N arbiters. For brevity, only grant arbiters 1 and j and accept arbiters 1 and i are shown here.

As with the implementation of *i*-LQF the complexity of the implementation of GS-LQF depends on the number of bits, *b*, used to represent $L_{i,j}(t)$. This will determine the number of gates required to register the preference lists, $O(bN^2)$ and the number of gates required to implement each comparator, $O(\log N \log b)$.

4.6 Egalitarian Stable Marriage

In [15], the authors describe an algorithm to find an egalitarian stable marriage. Whereas the male-optimal GSA simultaneously:

$$\begin{aligned} &\text{minimizes} \quad \sum_{(m,w) \in M} R_m(m,w), \quad \text{and} \\ &\text{maximizes} \quad \sum_{(m,w) \in M} R_w(w,m) \end{aligned} \tag{14}$$

the egalitarian stable matching

$$\text{minimizes} \quad \sum_{(m,w) \in M} [R_m(m,w) + R_w(w,m)], \tag{15}$$

where:

$R_m(m,w)$ = the position of woman w in man m's preference list,

$R_w(w,m)$ = the position of man m in woman w's preference list.

If we could use this algorithm to schedule cells, it would remove the decision as to whether to give preference to either inputs or outputs. Unfortunately, the best known algorithm for finding an egalitarian matching requires an $O(N^4)$ running time and is impractical to implement in hardware.

References

[1] Akata, M.; Karube, S.-I.; Sakamoto, T.; Saito, T.; Yoshida, S.; Maeda, T. "A 250 Mb/s 32x32 CMOS crosspoint LSI for ATM switching systems," *IEEE J. Solid-State Circuits*, Vol.25, No.6, pp.1433-1439, Dec. 1990.

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

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

[4] Anick, D.; Mitra, D.; Sondhi, M.M. "Stochastic theory of a data-handling system with multiple sources," *Bell System Technical Journal*, Vol.61, pp.1871-1894, 1982.

[5] Brown, T.X; Liu, K.H. "Neural network design of a Banyan network controller," *IEEE J. Selected Areas Communications*, Vol.8, pp.1289-1298, Oct. 1990.

[6] Chen, M.; Georganas, N.D., "A fast algorithm for multi-channel/port traffic scheduling" *Proc. IEEE Supercom/ICC '94*, pp.96-100.

[7] Cruz, R., "A calculus for network delay, part I: network elements in isolation," *IEEE Trans. Information Theory*, Vol. 37, No.1, pp.114-121, 1991.

[8] Demers, A.; Keshav, S.; Shenker, S. "Analysis and simulation of a fair queueing algorithm." *Internetworking: Research and Experience*, Sept. 1990, vol.1, (no.1):3-26.

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

[10] Elwalid, A.I.; Mitra, D. "Effective bandwidth of general Markovian traffic sources and admission control of high speed networks," *IEEE/ACM Trans. Networking*, June 1993, vol.1, (no.3):329-43.

[11] Eng, K.; Hluchyj, M.; and Yeh, Y. "Multicast and broadcast services in a knockout packet switch," *INFOCOM '88*, 35(12) pp.29-34.

[12] Even, S.; Tarjan, R.E. "Network flow and testing graph connectivity", *SIAM J. Comput.*, 4 (1975), pp.507-518.

[13] Gale, D.; Shapley, L.S.; "College Admissions and the stability of marriage", *American Mathematical Monthly*, Vol.69, pp9-15, 1962.

[14] Giacopelli, J.; Hickey, J.; Marcus, W.; Sincoskie, D.; and Littlewood, M. “Sunshine: A high-performance self-routing broadband packet switch architecture,” *IEEE J. Selected Areas Communications*, Vol.9, No.8, pp.1289-1298, Oct 1991.

[15] Gusfield, D; Irving, R; “The Stable Marriage Problem: Structure and Algorithms”, *The MIT Press*, Cambridge, MA, USA. 1989.

[16] Heffes, H.; Lucantoni, D. M., “A Markov modulated characterization of packetized voice and data traffic and related statistical multiplexer performance,” *IEEE J. Selected Areas in Communications*, 4, 1986, pp.856-868.

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

[18] Hopfield, J.J. “Neural networks and physical systems with emergent collective computational abilities,” *Proc. National Academy of Science*, Vol. 79 pp.2554-2558, 1982.

[19] Huang, A.; Knauer, S. “Starlite: A wideband digital switch,” *Proc. GLOBECOM ‘84* (1984), pp.121-125.

[20] Hui, J.; Arthurs, E. “A broadband packet switch for integrated transport,” *IEEE J. Selected Areas Communications, 5*, 8, Oct 1987, pp 1264-1273.

[21] Jain, R.; Routhier, S.A. “Packet Trains: measurements and a new model for computer network traffic,” *IEEE J. Selected Area Communications*, Vol.4, pp.986-995, 1986.

[22] Karol, M.; Hluchyj, M.; and Morgan, S. “Input versus output queueing on a space division switch,” *IEEE Trans. Communications*, 35(12) (1987) pp.1347-1356.

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

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

[25] Karp, R.; Vazirani, U.; and Vazirani, V. “An optimal algorithm for on-line bipartite matching,” *Proc. 22nd ACM Symp. on Theory of Computing*, pp.352-358 Maryland, 1990.

[26] Knuth, D.E; “Marriages Stables”, *Les Presses de l’Université de Montréal*, Montréal, 1976.

[27] Kumar, P.R.; Meyn, S.P.; “Stability of Queueing Networks and Scheduling Policies”, *IEEE Transactions on Automatic Control*, Vol.40, No.2, Feb. 1995.

[28] Low, S.; Varaiya, P. “Burstiness bounds for some burst reducing servers,” *Proc. INFOCOM ‘93*, pp.2-9, March 1993.

[29] Kelly, F.P.. “Effective bandwidths at multiclass queues,” *Queueing Systems Theory and Applications*, Oct. 1991, vol.9, (no.1-2):5-15.

[30] Keshav, S. “On the efficient implementation of fair queueing,” *Internetworking: Research and Experience*, Sept. 1991, vol.2, (no.3):157-73.

[31] Kesidis, G.; Walrand, J.; Chang, C.-S. “Effective bandwidths for multiclass Markov fluids and other ATM sources.” *IEEE/ACM Transactions on Networking*, Aug. 1993, vol.1, (no.4):424-8.

[32] Leland, W.E.; Willinger, W.; Taqqu, M.; Wilson, D. “On the self-similar nature of Ethernet traffic”, *Proc. of Sigcomm*, San Francisco, 1993, pp.183-193.

[33] Li, S.-Y. “Theory of periodic contention and its application to packet switching”, *Proc. of INFOCOM 1988*, pp.320-325.

[34] Marrakchi, A.; Troudet, T.P. “A neural net arbitrator for large crossbar packet switches,” *IEEE Trans. Circuits and Systems*, Vol.CAS-36, pp.1039-1041, July 1989.

[35] Matsunaga, H.; Uematsu, H. “A 1.5Gb/s 8x8 cross-connect switch using a time reservation algorithm,” *IEEE J. Selected Area Communications*, Vol.9, No.8, pp.1308-1317, Oct. 1991.

[36] Neuts, M. “Matrix Geometric Solutions in Stochastic Models: An Algorithmic Approach,” *Johns Hopkins University Press*, Baltimore, 1981.

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

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

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

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

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

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

[43] Wolff, R.W. “Stochastic modeling and the theory of queues,” *Prentice Hall Intl.*, New Jersey, 1989.

[44] Zhang, H.; Keshav, S. “Comparison of rate-based service disciplines,” *Computer Communication Review*, Sept. 1991, vol.21, (no.4):113-21.

[45] Zhang, L. “Virtual Clock: A New Traffic Control Algorithm for Packet Switching Networks,” *ACM Transactions on Computer Systems*, Vol 9. No.2, pp.101-124, May 1991.

APPENDIX 1

Arbiter Synchronization for Single-Iteration SLIP

In this appendix, we find an approximate expression for the expected number of synchronized output schedulers, $E[S(t)]$.

We partition the set of switch inputs $\mathbf{I} = \{1, \ldots, N\}$ into two subsets at time t: $\mathbf{A}(t)$, the set of inputs that are matched and $\mathbf{B}(t)$, the set of inputs that are not matched. If the arrival rate averaged across all inputs is λ, then for a sustainable and stationary ergodic arrival process the expected match size is λN and on average, λN inputs will send a cell. Clearly then, the expected size of $\mathbf{A}(t)$ and $\mathbf{B}(t)$ are

$$E[|\mathbf{A}(t)|] = \lambda N \quad , \qquad E[|\mathbf{B}(t)|] = (1 - \lambda) N = \bar{\lambda} N. \tag{1}$$

Similarly, we partition the set of switch inputs $\mathbf{O} = \{1, \ldots, N\}$ into two subsets at time t: $\mathbf{A}^O$(t)and $\mathbf{B}^O$(t). $\mathbf{A}^O$(t) is the set of outputs that are matched to inputs in $\mathbf{A}$(t), and $\mathbf{B}^O$(t) are the outputs not matched at time t.

As a result of the matching at time t, the set $\mathbf{A}(t)$ is transformed into the set $\tilde{\mathbf{A}}(t + 1)$, the set of inputs that the outputs in $\mathbf{A}^O$(t) point to at time t+1. Each element in $\mathbf{A}(t)$ is unique, and because

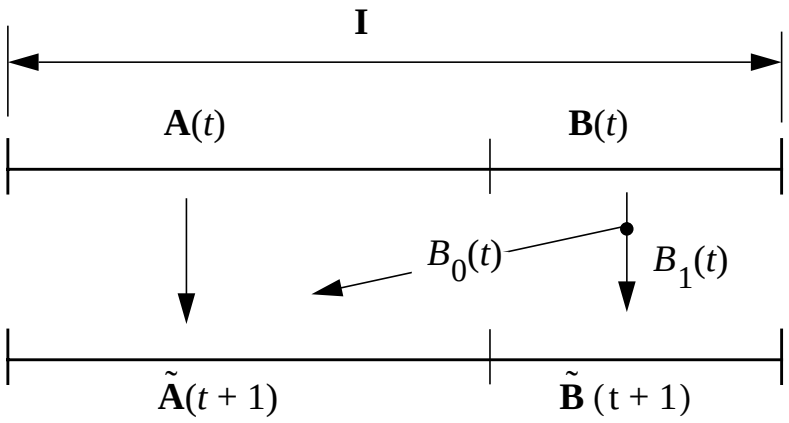


FIGURE A1.1 Mapping of matched and unmatched inputs at time *t*, to modified sets at time *t*+1.

they were matched at time t, each element mapped from $\mathbf{A}(t)$ into $\tilde{\mathbf{A}}(t+1)$ is also unique. The expected size of $\tilde{\mathbf{A}}(t+1)$ is

$$\mathrm{E}\left[\left|\tilde{\mathbf{A}}(t+1)\right|\right] = \mathrm{E}\left[\left|\tilde{\mathbf{A}}(t)\right|\right] = \mathrm{E}\left[\left|\mathbf{A}(t)\right|\right] = \lambda N. \quad (2)$$

Because none of its elements are matched, the set $\mathbf{B}(t)$ is unchanged, i.e. $\tilde{\mathbf{B}}(t+1) = \mathbf{B}(t)$.

To determine $\mathrm{E}[S(t+1)]$, the expected number of synchronized output arbiters at time t+1, we must find the number of elements in $\tilde{\mathbf{A}}(t+1)$ that are still unique and the number that clash with elements mapped from $\mathbf{B}(t)$. Without loss of generality, and to simplify our calculations, we assume that a one-to-one mapping is applied to $\tilde{\mathbf{A}}(t+1)$ such that $\tilde{\mathbf{A}}(t+1) = \mathbf{A}(t)$ and hence $\tilde{\mathbf{B}}(t+1) \neq \mathbf{B}(t)$. As before, the elements of $\tilde{\mathbf{A}}(t+1)$ are unique, and we can think of the elements of $\tilde{\mathbf{B}}(t+1)$ as randomly distributed over **I**. This is shown in Figure A1.1.

To find $\mathrm{E}[S(t+1)]$, we partition $\mathbf{B}(t)$ into $B_0(t)$ elements that are mapped into $\tilde{\mathbf{A}}(t+1)$, and $B_1(t)$ elements that are mapped into $\tilde{\mathbf{B}}(t+1)$.

Finally, we define $U_A(t+1)$ and $U_B(t+1)$ as the number of unique elements in $\tilde{\mathbf{A}}(t+1)$ and $\tilde{\mathbf{B}}(t+1)$ respectively, and

$$\mathrm{E}\,[S(t+1)] \;=\; N - \mathrm{E}\,[U_A(t+1)] - \mathrm{E}\,[U_B(t+1)] \quad . \tag{3}$$

If we assume that under the mapping the elements of $\mathbf{B}(t)$ are *uniformly* distributed in $\mathbf{I}$, then

$$\begin{aligned} \mathrm{E}\left[U_A(t+1) \,\middle|\, |\tilde{\mathbf{A}}(t+1)|, B_0(t)\right] &= |\tilde{\mathbf{A}}(t+1)| \cdot \left(\frac{|\tilde{\mathbf{A}}(t+1)| - 1}{|\tilde{\mathbf{A}}(t+1)|}\right)^{B_0(t)} \\ \therefore\ \mathrm{E}\,[U_A(t+1) \,|\, |\mathbf{B}(t)|, B_1(t)] &= (N - |\mathbf{B}(t)|) \cdot \left(\frac{(N - |\mathbf{B}(t)|) - 1}{(N - |\mathbf{B}(t)|)}\right)^{|\mathbf{B}(t)| - B_1(t)} \end{aligned} \tag{4}$$

and,

$$\begin{aligned} \mathrm{E}\left[U_B(t+1) \,\middle|\, |\tilde{\mathbf{B}}(t)|, B_1(t)\right] &= B_1(t) \cdot \left(\frac{|\tilde{\mathbf{B}}(t+1)| - 1}{|\tilde{\mathbf{B}}(t+1)|}\right)^{B_1(t)-1} \\ \therefore\ \mathrm{E}\left[U_B(t+1) \,\middle|\, |\mathbf{B}(t)|, B_1(t)\right] &= B_1(t) \cdot \left(\frac{|\mathbf{B}(t)| - 1}{|\mathbf{B}(t)|}\right)^{B_1(t)-1} \end{aligned} \tag{5}$$

Hence,

$$\begin{aligned} &\mathrm{E}\left[S(t+1) \,\middle|\, |\mathbf{B}(t)|, B_1(t)\right] \;= \\ &N - (N - |\mathbf{B}(t)|) \cdot \left(\frac{(N - |\mathbf{B}(t)|) - 1}{(N - |\mathbf{B}(t)|)}\right)^{|\mathbf{B}(t)| - B_1(t)} - B_1(t) \cdot \left(\frac{|\mathbf{B}(t)| - 1}{|\mathbf{B}(t)|}\right)^{B_1(t)-1}. \end{aligned} \tag{6}$$

To find $\mathrm{E}\,[S(t+1)]$ we need to know the distributions of $|\mathbf{B}(t)|$ and $B_1(t)$. Unfortunately, both random variables depend on the traffic arrival pattern. Furthermore, we cannot use Jensen's inequality to bound Eq. 6 from below or above. This is because Eq. 4 is a concave function of $|\mathbf{B}(t)|$ and $B_1(t)$ whereas Eq. 5 is convex.

However, simulations with a variety of arrival patterns indicate that $\mathrm{E}\,[S(t+1)]$ is relatively insensitive to traffic statistics. We therefore approximate the random variables with $|\mathbf{B}(t)| \approx \mathrm{E}\,[|\mathbf{B}(t)|] \;=\; \bar{\lambda} N$ and $B_1(t) \approx \mathrm{E}\,[B_1(t)] \;=\; \bar{\lambda}^2 N$.

This leads us to the approximation,

$$\mathrm{E}\,[S(t)] \approx N - \lambda N\left(\frac{\lambda N - 1}{\lambda N}\right)^{\lambda\bar{\lambda}N} - \bar{\lambda}^2 N\left(\frac{\bar{\lambda}N - 1}{\bar{\lambda}N}\right)^{\bar{\lambda}^2 N - 1} . \tag{7}$$

APPENDIX 2

Stability of Single-Iteration SLIP Algorithm

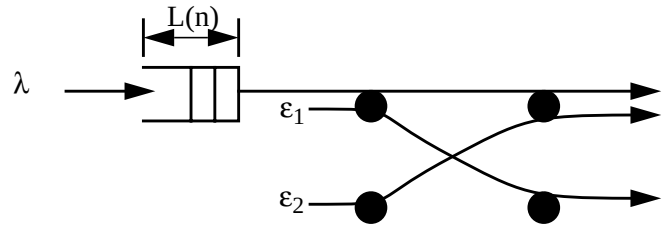


FIGURE A2.1 2x2 switch with a single queue.

1 Single-Step Drift Analysis of 2x2 Switch with 1 Queue

1.1 First Approximation

Consider the switch in Figure A2.1. All three arrival processes are i.i.d. Bernoulli. We wish to find the values of λ, ε_1 and ε_2 for which the switch is *stable*.

Define $\hat{L}$ to be the expected value of $L(n+1)$ (the occupancy of Q(1,1) at time n+1) conditioned on $L(n)$ and $L(n) > 0$

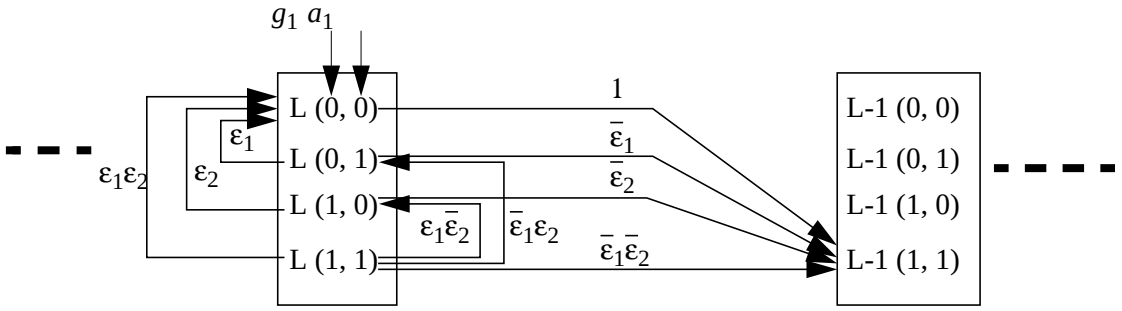
$$\hat{L} = \mathrm{E}\,[L(n+1) \mid L(n), L(n) > 0] \quad . \tag{1}$$

If $\hat{L} - L(n) > 0$ then $L(n)$ has a single-step positive drift which means that $E[L(n)] \to \infty$ and the switch is unstable.

This system may be described as a discrete-time Markov chain (DTMC) with state

$$\underline{X}_L = (g_1, a_1) \tag{2}$$

where L is the occupancy of $Q(1, 1)$, the value of the pointer g_1 at output 1, and the value of the pointer a_1 at input 1. The evolution of state for the switch using the SLIP algorithm, conditioned on $L(n) > 0$ and $\lambda = 0$ is shown below



where $\bar{\varepsilon}_1 = 1 - \varepsilon_1, \bar{\varepsilon}_2 = 1 - \varepsilon_2$. The state transition matrix, P conditioned on $L(n) > 0$ and $\lambda = 0$ is

$$P = \begin{bmatrix} 0 & 0 & 0 & 1 \\ \varepsilon_1 & 0 & 0 & \bar{\varepsilon}_1 \\ \varepsilon_2 & 0 & 0 & \bar{\varepsilon}_2 \\ \varepsilon_1\varepsilon_2 & \bar{\varepsilon}_1\varepsilon_2 & \varepsilon_1\bar{\varepsilon}_2 & \bar{\varepsilon}_1\bar{\varepsilon}_2 \end{bmatrix} \tag{3}$$

from which we obtain the steady-state distribution

$$\begin{aligned} \pi(1, 1) &= \frac{1}{1 + \bar{\varepsilon}_1\varepsilon_2 + \varepsilon_1\bar{\varepsilon}_2 + \varepsilon_1\varepsilon_2(1 + \bar{\varepsilon}_1 + \bar{\varepsilon}_2)} \\ \pi(1, 0) &= \varepsilon_1\bar{\varepsilon}_2 \cdot \pi(L, 1, 1) \\ \pi(0, 1) &= \bar{\varepsilon}_1\varepsilon_2 \cdot \pi(L, 1, 1) \\ \pi(0, 0) &= [\varepsilon_1\varepsilon_2(1 + \bar{\varepsilon}_1 + \bar{\varepsilon}_2)] \cdot \pi(L, 1, 1) \end{aligned} \tag{4}$$

From $\underline{\pi}$ we find

$$\hat{L} = L(n) - \left[\frac{1}{1 + \bar{\varepsilon}_1\varepsilon_2 + \varepsilon_1\bar{\varepsilon}_2 + \varepsilon_1\varepsilon_2\,(1 + \bar{\varepsilon}_1 + \bar{\varepsilon}_2)}\right] \tag{5}$$

which if we consider the arrivals at rate λ gives J, the expected single-step increase function

$$J = \lambda - \left[\frac{1}{1 + 2\varepsilon + \varepsilon^2 - 2\varepsilon^3}\right] \tag{6}$$

where we define $\varepsilon = \varepsilon_1 = \varepsilon_2$ because of the symmetric and identical dependence on ε_1 and ε_2. The unstable region of operation is given by

$$\lambda > \left[\frac{1}{1 + 2\varepsilon + \varepsilon^2 - 2\varepsilon^3}\right]. \tag{7}$$

We can find the maximum positive drift J_{max} (the "most unstable operating point") by defining

$$\lambda = 1 - \varepsilon - \delta, \qquad \delta < 1 \tag{8}$$

From Eq. 6 we find that $J_{max}\big|_{\delta = 0} \approx 0.098$ which tells us that the drift can be positive for any value of $\delta < 0.098$, or alternatively

$$J > 0 \Rightarrow \lambda + \varepsilon > 1 - 0.098\,. \tag{9}$$

1.2 Second Approximation

The first approximation assumes that cells arriving at rates ε_1 and ε_2 are discarded if they are not successfully scheduled. However, if unsuccessful cells are queued rather than discarded, they will affect the service rate of $Q(1, 1)$ over multiple cell times. We model this effect by approximating the *busy* and *idle* cycles of input queues $Q(1, 2)$ and $Q(2, 1)$ with a 2-state Markov process, shown in Figure A2.2.

The behavior of $Q(1, 2)$ and $Q(2, 1)$ may be modelled by an M/G/1 queue with an arrival rate λ_2 and service rate $1 - \frac{1}{2}\lambda_1$. From [43] the expected duration of the busy and idle cycles

$$\mathrm{E}[B] = \frac{1}{\left(1 - \frac{1}{2}\lambda_1\right) - \lambda_2} = \frac{p}{1 - p} \tag{10}$$

$$\mathrm{E}[I] = \frac{1}{\lambda_2} = \frac{q}{1 - q} \tag{11}$$

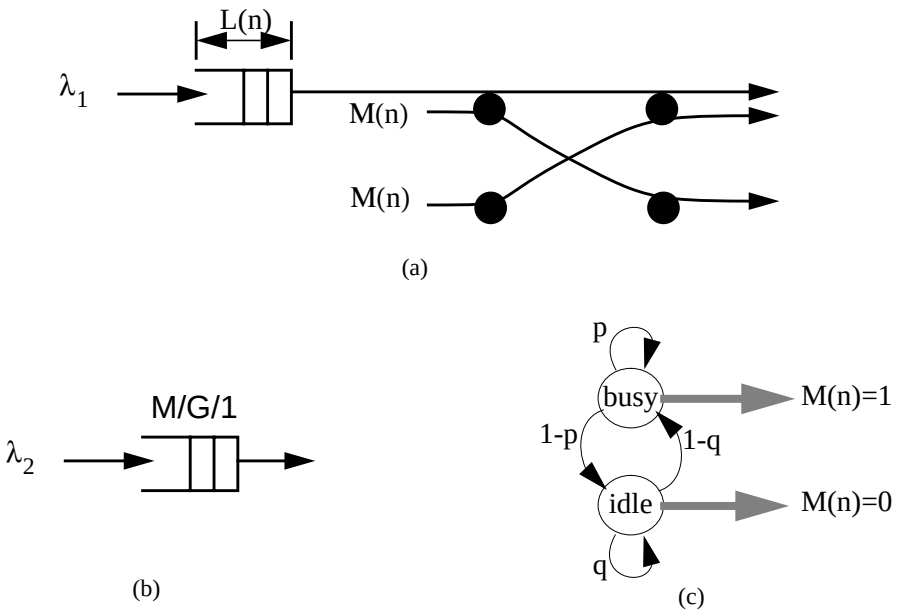


FIGURE A2.2 (a) Approximation of arrivals as an on-off process modulated by a 2-state discrete-time Markov chain, M(n). (b) The arrival process models the busy/idle cycles of input queues Q(1,2) and Q(2,1). (c) The Markov chain alternates between the busy and idle states. In the busy state, the arrival rate is 1. In the idle state the arrival rate is 0.

from which we obtain p and q as functions of λ_1 and λ_2.

To find

$$\hat{L} = \mathrm{E}\,[L(n+1) |\ L(n), L(n) > 0] \tag{12}$$

we model the evolution of the system using a DTMC with state

$$\underline{X} = (g_1, a_1, s_1, s_2) \tag{13}$$

where s_i is the state (*busy* or *idle*) of the 2-state DTMC modulating the arrival process at input *i*. The state 16x16 transition matrix is

$$P = \begin{array}{c|cccc|cccc|cccc|cccc} s_1, s_2 = & \multicolumn{4}{c|}{(B,B)} & \multicolumn{4}{c|}{(B,I)} & \multicolumn{4}{c|}{(I,B)} & \multicolumn{4}{c}{(I,I)} \\ & 00 & 01 & 10 & 11 & 00 & 01 & 10 & 11 & 00 & 01 & 10 & 11 & 00 & 01 & 10 & 11 \\ \hline (B,B)\ 00 & & & & p^2 & & & & \bar{p}p & & & & \bar{p}p & & & & \bar{p}\bar{p} \\ (B,B)\ 01 & p^2 & & & & \bar{p}p & & & & \bar{p}p & & & & \bar{p}\bar{p} & & & \\ (B,B)\ 10 & p^2 & & & & p\bar{p} & & & & p\bar{p} & & & & \bar{p}\bar{p} & & & \\ (B,B)\ 11 & p^2 & & & & pp & & & & pp & & & & pp & & & \\ \hline (B,I)\ 00 & & & & p\bar{q} & & & & pq & & & & p\bar{q} & & & & \bar{p}q \\ (B,I)\ 01 & p\bar{q} & & & & pq & & & & p\bar{q} & & & & \bar{p}q & & & \\ (B,I)\ 10 & & & p\bar{q} & \bar{p}q & & & pq & & & & p\bar{q} & & & & \bar{p}q & \\ (B,I)\ 11 & & p\bar{q} & & & & pq & & & & p\bar{q} & & & & \bar{p}q & & \\ \hline (I,B)\ 00 & & & & p\bar{q} & & & & pq & & & & p\bar{q} & & & & \bar{p}q \\ (I,B)\ 01 & & & & p\bar{q} & & & & pq & & & & p\bar{q} & & & & \bar{p}q \\ (I,B)\ 10 & p\bar{q} & & & & pq & & & & p\bar{q} & & & & \bar{p}q & & & \\ (I,B)\ 11 & & p\bar{q} & & & & pq & & & & p\bar{q} & & & & \bar{p}q & & \\ \hline (I,I)\ 00 & & & & \bar{q}^2 & & & & \bar{q}q & & & & \bar{q}q & & & & q^2 \\ (I,I)\ 01 & & & & \bar{q}^2 & & & & \bar{q}q & & & & \bar{q}q & & & & q^2 \\ (I,I)\ 10 & & & & \bar{q}^2 & & & & \bar{q}q & & & & \bar{q}q & & & & q^2 \\ (I,I)\ 11 & & & & \bar{q}^2 & & & & qq & & & & qq & & & & q^2 \end{array} \tag{14}$$

from which we find the steady-state distribution

$$\underline{\pi} = (\pi(B,B,0,0)\ldots, \pi(I,I,1,1)) \quad . \tag{15}$$

For brevity, we show an example of just one element of the distribution

$$\pi(B,B,0,0) = \frac{p(q-1)^2(p^2q^2 - pq^2 + p^3q - 3qp^2 + 5pq - 2q - p^3 + p^2 - 3p + 2)}{(q^2p^3 + p^2q^2 - 3q^2p + q^2 + p^4q - p^3q - p^3q - 2qp^2 + 7pq - 3q - p^4 + p^3 + 2p^2 - 3p + 3)\ (q - 2 + p)^2} \tag{16}$$

We find $\hat{L}$ from $\underline{\pi}$ and the expression

$$\begin{aligned} \hat{L} = L(n) - [&\pi(B,B,0,0) + \pi(B,I,0,0) + \pi(I,B,0,0) + \pi(I,I,0,0) + \pi(B,I,1,0) \\ &+ \pi(I,B,0,1) + \pi(I,I,0,1) + \pi(I,I,1,0) + \pi(I,I,1,1)] \end{aligned} \tag{17}$$

This leads to an expression for the single-step drift function *J* as the ratio of two 10th degree polynomials. As a result, we have only been able to find the conditions on λ_1 and λ_2 numerically such that *J* is negative and the switch is stable.

2 Matrix Geometric Solution for 2x2 Switch with 1 Queue

Consider again the switch in Figure A2.1. In this section, we find the steady-state distribution function for the queue occupancy, L and find the values of λ and ε necessary for stability. To do this, we use the matrix-geometric technique of Neuts [36]. We define the state

$$\underline{\mathbf{X}} = [\underline{X}_0, \underline{X}_1, \underline{X}_2, \ldots] \tag{18}$$

where $\underline{X}_L = \{ (L, g_1, a_1) ; g_1 \in \{0, 1\}, a_1 \in \{0, 1\} \}$ and wish to solve the infinite system of linear equations

$$\Pi = \Pi\tilde{\mathbf{P}}, \qquad \Pi\underline{e} = 1 \tag{19}$$

where

$$\Pi = [\underline{\Pi}_0, \underline{\Pi}_1, \underline{\Pi}_2, \ldots], \qquad \underline{\Pi}_i = \text{steady-state distribution of } \underline{X}_i, \tag{20}$$

$\underline{e}$ is the column vector with all its elements equal to 1, and the transition probability matrix is of the form

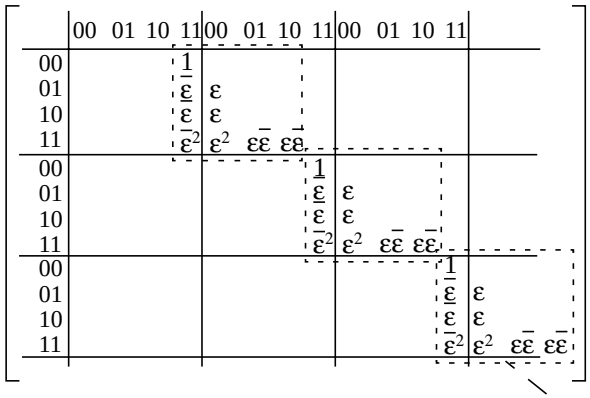
$$\tilde{\mathbf{P}} = \begin{bmatrix} B_0 & A_0 & 0 & \ldots \\ B_1 & A_1 & A_0 & \ldots \\ B_2 & A_2 & A_1 & \ldots \\ \ldots & \ldots & \ldots & \ldots \end{bmatrix}. \tag{21}$$

In this example, $\tilde{\mathbf{P}}$ is most easily understood when separated into two parts, conditioned on whether or not an arrival occurs

$$\tilde{\mathbf{P}} = \lambda\tilde{\mathbf{P}}_{\lambda} + \bar{\lambda}\tilde{\mathbf{P}}_{\bar{\lambda}} \tag{22}$$

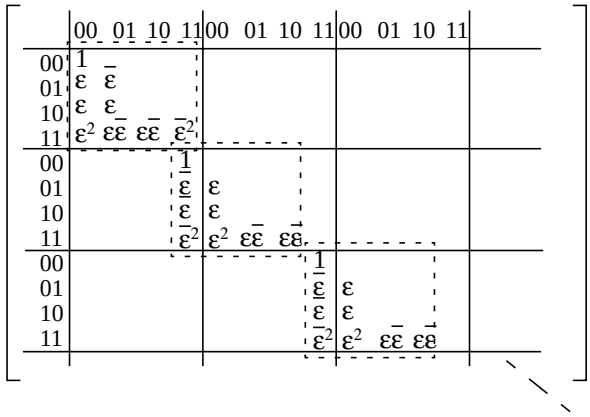
where $\tilde{\mathbf{P}}_{\lambda}$ and $\tilde{\mathbf{P}}_{\bar{\lambda}}$ are

$\tilde{\mathbf{P}}_{\lambda}$ =



L=0 L=1 L=2

$\tilde{\mathbf{P}}_{\bar{\lambda}}$ =



Clearly, $\tilde{\mathbf{P}}$ is of the form of Eq. 21.

To find the steady state distribution Π, we use Lemma 1.2.3 and Theorem 1.2.1 of [36]. If $\tilde{\mathbf{P}}$ is positive recurrent, then we can find (using a method outlined on page 9 of [36]) a unique matrix R which satisifies the equation

$$R = \sum_{k=0}^{\infty} R^k A_k \qquad \text{, where } A_k \text{ is as shown in Eq. 21.} \tag{23}$$

such that $\underline{\Pi}_{i+1} = \underline{\Pi}_i R$ for $i \geq 0$, the spectral radius of R, sp(R)<1, the matrix

$$B[R] = \sum_{k=0}^{\infty} R^k B_k \tag{24}$$

is stochastic,

$$\underline{\Pi}_0 (I - R)^{-1} \underline{e} = 1 \text{, and} \tag{25}$$

$$\underline{\Pi}_0 = \underline{\Pi}_0 B[R] \quad . \tag{26}$$

Alternatively, for the Markov chain to be positive recurrent (i.e. for the system to be stable) it is necessary that the spectral radius of R be less than 1.

Solving for R and $B[R]$ we obtain

$$R = g(\varepsilon, \lambda) \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 - 2\varepsilon\lambda + 2\varepsilon^2\lambda & (1-\varepsilon)\varepsilon\lambda & (1-\varepsilon)\varepsilon\lambda & \frac{\lambda}{1-\lambda} \\ 1 - 2\varepsilon\lambda + 2\varepsilon^2\lambda & (1-\varepsilon)\varepsilon\lambda & (1-\varepsilon)\varepsilon\lambda & \frac{\lambda}{1-\lambda} \\ \varepsilon(3 - 2\varepsilon - 2\lambda + 2\varepsilon\lambda) & 1-\varepsilon & 1-\varepsilon & \frac{\lambda(2 + \varepsilon - 2\varepsilon^2 - 2\varepsilon\lambda + 2\varepsilon^2\lambda)}{1-\lambda} \end{bmatrix} \tag{27}$$

$$B[R] = \begin{bmatrix} 1-\lambda & 0 & 0 & \lambda \\ \varepsilon(1-\lambda) & 1-\varepsilon-\lambda+\varepsilon\lambda & 0 & \lambda \\ \varepsilon(1-\lambda) & 0 & 1-\varepsilon-\lambda+\varepsilon\lambda & \lambda \\ \varepsilon^2(1-\lambda) & (1-\varepsilon)\varepsilon(1-\lambda) & (1-\varepsilon)\varepsilon(1-\lambda) & 1-2\varepsilon+\varepsilon^2+2\varepsilon\lambda-\varepsilon^2\lambda \end{bmatrix}, \tag{28}$$

where

$$g(\varepsilon, \lambda) = \frac{\varepsilon\lambda}{(1 - 2\varepsilon\lambda - \varepsilon^2\lambda + 2\varepsilon^3\lambda + 2\varepsilon^2\lambda^2 - 2\varepsilon^3\lambda^2)} . \tag{29}$$

The spectral radius

$$\text{sp}(R) < 1 \Leftrightarrow \lambda < \left[\frac{1}{1 + 2\varepsilon + \varepsilon^2 - 2\varepsilon^3} \right] \tag{30}$$

which is identical to the stability requirement of Eq. 7.

We can also solve Eq. 25 and Eq. 26 above to find $\underline{\Pi}_0$. The resulting expression is a vector of elements, each of which is the ratio of two 6th order polynomials in ε and λ from which we can successively generate $\underline{\Pi}_1, \underline{\Pi}_2, \underline{\Pi}_3, \ldots$. We do not repeat these (long) expressions here.

APPENDIX 3

Stability of 2x2 Switch

1 Stability of 2x2 Switch with 3 Active Flows (Theorem 4.2)

In this section, we find sufficient conditions on a scheduling algorithm for a 2x2 switch with 3 active flows such that the switch is stable under all admissible, arrival processes with i.i.d. interarrival times. The switch is illustrated in Chapter 4 Figure 4.6.

1.1 Definitions

Define the vector of queue occupancies

$$\underline{L}(n) = (L_{1,1}(n), L_{1,2}(n), L_{2,1}(n)) . \quad (1)$$

We now consider the single-step change in $\underline{L}(n)$ conditioned on whether the switch is in configuration *A* or *B*, as shown in Chapter 4 Figure 4.6:

$$\left.\begin{aligned} L_{1,1}(n+1) &= [L_{1,1}(n)-1]^{+} + \eta_1 \\ L_{1,2}(n+1) &= L_{1,2}(n) + \eta_2 \\ L_{2,1}(n+1) &= L_{2,1}(n) + \eta_3 \end{aligned}\right\} \text{Configuration } A$$

$$\left.\begin{aligned} L_{1,1}(n+1) &= L_{1,1}(n) + \eta_1 \\ L_{1,2}(n+1) &= [L_{1,2}(n)-1]^{+} + \eta_2 \\ L_{2,1}(n+1) &= [L_{2,1}(n)-1]^{+} + \eta_3 \end{aligned}\right\} \text{Configuration } B \tag{2}$$

where

$$\eta_i = \begin{cases} 1, & \text{if an arrival occurs at queue i, w.p. } \lambda_i \\ 0, & \text{else} \end{cases} \tag{3}$$

We define the quadratic Lyapunov function

$$V(\underline{L}(n)) = \underline{L}(n)^T Q \underline{L}(n) \geq 0\,, \quad \text{where } Q = [q_{ij}], \quad q_{ij} \geq 0\,. \tag{4}$$

1.2 Problem statement

If we can find a Q such that $\mathrm{E}[V(\underline{L}(n+1)) - V(\underline{L}(n)) \mid \underline{L}(n)] < 0$, then the queue occupancy has a downward drift and the switch is said to be *stable*.

1.3 Solution

As there is no systematic method for finding Q we must guess its form. We assume that Q is symmetric, i.e. $[q_{ij}] = [q_{ji}]$. Further, we guess that if the switch is stable under all admissible offered loads, then it will be marginally stable when $\lambda_2 = \lambda_3 = 1$ and $\lambda_1 = 0$. i.e.

$$\mathrm{E}[V(\underline{L}(n+1)) - V(\underline{L}(n)) \mid \underline{L}(n), \lambda_2 = \lambda_3 = 1, \lambda_1 = 0] = 0\,. \tag{5}$$

This leads us to the guess

$$Q = a \begin{bmatrix} 4 & 2 & 2 \\ 2 & 1 & 1 \\ 2 & 1 & 1 \end{bmatrix}, \text{ for any integer, } a\,. \tag{6}$$

This matrix Q leads to a stable switch under the following conditions.

Conditioned on $L_{1,1}(n) > 0, L_{1,2}(n) > 0, L_{2,1}(n) > 0$:

$$\begin{aligned} \mathrm{E}\left[V(\underline{L}(n+1)) - V(\underline{L}(n)) \mid \underline{L}(n), A\right] &= (-2 + 2\lambda_1 + \lambda_2 + \lambda_3) \\ &\quad (-2 + 2\lambda_1 + \lambda_2 + \lambda_3 + 4L_{1,1}(n) + 2L_{1,2}(n) + 2L_{2,1}(n)) \\ &< 0 \end{aligned} \tag{7}$$

$$\begin{aligned} \mathrm{E}\left[V(\underline{L}(n+1)) - V(\underline{L}(n)) \mid \underline{L}(n), B\right] &= (-2 + 2\lambda_1 + \lambda_2 + \lambda_3) \\ &\quad (-2 + 2\lambda_1 + \lambda_2 + \lambda_3 + 4L_{1,1}(n) + 2L_{1,2}(n) + 2L_{2,1}(n)) \\ &< 0 \end{aligned} \tag{8}$$

Similarly, conditioned on either $L_{1,2}(n) = 0$ or $L_{2,1}(n) = 0$:

$$\mathrm{E}\left[V(\underline{L}(n+1)) - V(\underline{L}(n)) \mid \underline{L}(n), A\right] < 0 \tag{9}$$

whereas $\mathrm{E}\left[V(\underline{L}(n+1)) - V(\underline{L}(n)) \mid \underline{L}(n), B\right]$ maybe greater than 0.

Finally, conditioned on $L_{1,1}(n) = 0$:

$$\mathrm{E}\left[V(\underline{L}(n+1)) - V(\underline{L}(n)) \mid \underline{L}(n), B\right] < 0 \tag{10}$$

whereas $\mathrm{E}\left[V(\underline{L}(n+1)) - V(\underline{L}(n)) \mid \underline{L}(n), A\right] > 0$.

1.4 Stable Algorithms

The value of Q above enables us to define the following algorithm that will be stable under all admissible traffic with i.i.d. arrivals for a 2x2 switch with three active flows:

1. If $L_{1,1}(n) = 0$, set crossbar to configuration *B*.
2. Else, if either $L_{1,2}(n) = 0$ or $L_{2,1}(n) = 0$, set crossbar to configuration *A*.
3. Else, set crossbar configuration to either *A* or *B*.

2 Relative Queue Sizes (Theorem 4.3)

In this section we prove that if or any arrival process to a 2x2 switch, if for some *n*

$$\left| \{L_{1,1}(n) + L_{2,2}(n)\} - \{L_{1,2}(n) + L_{2,1}(n)\} \right| \le 3 \tag{11}$$

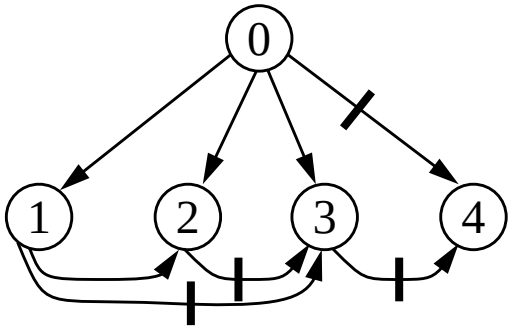


FIGURE A3.1 All possible single-step increases in $|D(n)|$. Arrows marked: ⟶ require two arrivals, which means that both queues in $L_M(n)$ are non-empty in the next cell time.

then for all $n' \geq n$

$$\left| \{L_{1,1}(n') + L_{2,2}(n')\} - \{L_{1,2}(n') + L_{2,1}(n')\} \right| \leq 4 . \tag{12}$$

For convenience, define

$$L_A(n) = L_{1,1}(n) + L_{2,2}(n) \quad , \qquad L_B(n) = L_{1,2}(n) + L_{2,1}(n) \tag{13}$$

and assume without loss of generality that $L_A(n) \geq L_B(n)$, i.e.

$$L_A(n) = L_B(n) + D(n) \tag{14}$$

for some $D(n) \geq 0$.

Finally, define

$$L_M(n) = \max(L_A(n), L_B(n)) \tag{15}$$

$$L_m(n) = \begin{cases} L_A(n) & \text{if } L_M(n) = L_B(n) \\ L_B(n) & \text{if } L_M(n) = L_A(n) \end{cases} . \tag{16}$$

Theorem A3.1: *All possible single step increases in* $|D(n)|$ *are shown in Figure 3.1.*

Proof:

Case (i): $D(n) = 0$. First we consider all possible values of $|D(n+1)|$ when $D(n) = 0$, i.e. $L_A(n) = L_B(n)$. We shall assume, without loss of generality that the two queues that contribute to $L_A(n)$ are served at time *n*.

At most two cells can arrive to the switch in a cell time, which means that

$$L_B(n+1) = L_B(n) + \begin{cases} 0 & : 0 \text{ arrivals} \\ 1 & : 1 \text{ arrival} \\ 2 & : 2 \text{ arrivals} \end{cases} \tag{17}$$

and that at most two cells can depart from the switch in a cell time, which means that

$$L_A(n+1) = L_A(n) + \begin{cases} -2 & : (2 \text{ dep. and } 0 \text{ arr.}), \\ -1 & : (2 \text{ dep. and } 1 \text{ arr.}), \text{ or } (1 \text{ dep and } 0 \text{ arr.}) \\ 0 & : (2 \text{ dep. and } 2 \text{ arr.}), \text{ or } (1 \text{ dep and } 1 \text{ arr.}) \\ 1 & : (1 \text{ dep. and } 2 \text{ arr.}), \text{ or } (0 \text{ dep and } 1 \text{ arr.}) \end{cases} . \tag{18}$$

Note that there can only be 0 departures if $L_A(n) = 0$ and only 1 departure if either $L_{1,1}(n) = 0$ or $L_{2,2}(n) = 0$.

From Eq. 17 and Eq. 18 we find the following possible increases $D(n)$ when $D(n) = 0$.

$$|D(n+1)| = \begin{cases} 1: \ (L_A(n) \to L_A(n)-1), \ (L_B(n) \to L_B(n)) \\ 2: \ (L_A(n) \to L_A(n)-1), \ (L_B(n) \to L_B(n)+1) \\ 3: \ (L_A(n) \to L_A(n)-2), \ (L_B(n) \to L_B(n)+1) \\ 4: \ (L_A(n) \to L_A(n)-2), \ (L_B(n) \to L_B(n)+2) \end{cases} \tag{19}$$

Note that $|D(n)| = 4$ if and only if two arrivals occur. This means that both queues that contribute to $L_M(n+1)$ are non-empty.

Case (ii)-(iv): $|D(n)| = 1, 2, 3$. By enumerating all transitions, as for $|D(n)| = 0$, we find the transitions from 1, 2, 3 in Figure 3.1.

Case (v): $|D(n)| = 4$. In cases (i)-(iv) we found that transitions into $|D(n)| = 4$ require that two arrivals occur and that both of the queues that contribute to $L_M(n)$ when $|D(n)| = 4$ are non-empty. As a result, the matching at time *n* serves two queues, hence

$$L_M(n+1) \le L_M(n). \tag{20}$$

$L_m(n)$ is not served and so cannot decrease, therefore,

$$L_m(n+1) \ge L_m(n). \tag{21}$$

Finally,

$$|D(n+1)| \le 4 \tag{22}$$

❒.

The transitions in Figure 3.1 indicate

1. For any queue occupancy such that $|D(n)| \le 3$, the next state is bounded by $|D(n+1)| \le 4$.
2. If $|D(n+1)| = 4$, then both queues in $L_M(n+1)$ are non-empty.
3. If both queues in $L_M(n)$ are non-empty and $|D(n)| = 4$, then $|D(n+1)| \le 4$.

Hence, if for some n, $|D(n)| \le 3$, then for all $n' \ge n$, $|D(n')| \le 4$ which proves the theorem.

APPENDIX 4

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

1 Definitions

In this appendix we use the following definitions for an *N*x*N* switch:

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

$$\underline{L}(n) \equiv (L_{1,1}(n), \ldots, L_{1,N}(n), \ldots, L_{N,1}(n), \ldots, L_{N,N}(n)) \ . \tag{1}$$

2. The (constant) arrival rate matrix:

$$\Lambda \equiv [\lambda_{i,j}], \quad \text{where: } \sum_{i=1}^{N} \lambda_{i,j} \le 1, \sum_{j=1}^{N} \lambda_{i,j} \le 1, \lambda_{i,j} \ge 0 \tag{2}$$

and associated rate vector:

$$\underline{\lambda} \equiv (\lambda_{1,1}, \ldots, \lambda_{1,N}, \ldots, \lambda_{N,1}, \ldots, \lambda_{N,N}) \ . \tag{3}$$

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

$$\mathbf{A}(n) \equiv [A_{i,j}(n)], \quad \text{where:} \quad A_{i,j}(n) \equiv \begin{cases} 1 & \text{if arrival occurs at } Q(i,j) \text{ at time } n \\ 0 & \text{else} \end{cases} \tag{4}$$

and associated arrival vector:

$$\underline{A}(n) \equiv (A_{1,1}(n), \ldots, A_{1,N}(n), \ldots, A_{N,1}(n), \ldots, A_{N,N}(n)). \quad (5)$$

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

$$\mathbf{S}(n) \equiv [S_{i,j}(n)], \quad \text{where: } S_{i,j}(n) = \begin{cases} 1 & \text{if } Q(i,j) \text{ is served at time } n \\ 0 & \text{else} \end{cases} \quad (6)$$

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

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

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

We define the associated service vector:

$$\underline{S}(n) \equiv (S_{1,1}(n), \ldots, S_{1,N}(n), \ldots, S_{N,1}(n), \ldots, S_{N,N}(n)), \quad (7)$$

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

5. The *approximate* next-state vector:

$$\tilde{\underline{L}}(n+1) \equiv \underline{L}(n) - \underline{S}(n) + \underline{A}(n) \ , \quad (8)$$

which approximates the exact next-state of each queue

$$L_{i,j}(n+1) = [L_{i,j}(n) - S_{i,j}(n)]^{+} + A_{i,j}(n). \quad (9)$$

2 Main Theorem

Theorem A4.1: *An NxN switch is stable for the LQF algorithm under i.i.d. arrivals.*

3 Proof

Before proving this theorem, we first prove the following theorems.

Theorem A4.2: *The doubly stochastic matrices, Λ, form a convex set, C, with the set of extreme points equal to permutation matrices,* **S**.

Proof: The set C is clearly convex: for all rate matrices $\Lambda_1, \Lambda_2 \in C$ and for every real number α, $0 < \alpha < 1$, the point $\alpha\Lambda_1 + (1-\alpha)\Lambda_2 \in C$. A permutation matrix S is doubly stochastic and is therefore a member of the set C. Furthermore, there are no two distinct matrices $\Lambda_1, \Lambda_2 \in C$ such that $\alpha\Lambda_1 + (1-\alpha)\Lambda_2 = S$, for real α, $0 < \alpha < 1$. Hence, a permutation matrix is an extreme point of C. ❐

Theorem A4.3: $\underline{L}^T(n)\underline{\lambda} - \underline{S}^*(n) \le 0, \quad \forall (\underline{L}(n), \underline{\lambda})$, *where* $\underline{S}^*(n) = \max(\underline{L}^T(n)\underline{S}(n))$, *the service matrix selected by the LQF algorithm to maximize* $\underline{L}^T(n)\underline{S}(n)$.

Proof: Consider the linear programming problem:

$$\begin{aligned} &\max(\underline{L}^T(n)\underline{\lambda}) \\ &\text{s.t.} \quad \sum_{i=1}^{N} \lambda_{i,j} \le 1, \ \sum_{j=1}^{N} \lambda_{i,j} \le 1, \ \lambda_{i,j} \ge 0 \end{aligned} \tag{10}$$

which has a solution equal to an extreme point of the convex set, C. Hence,

$$\max(\underline{L}^T(n)\underline{\lambda}) \le \max(\underline{L}^T(n)\underline{S}(n)) \tag{11}$$

and so $\underline{L}^T(n)\underline{\lambda} - \max(\underline{L}^T(n)\underline{S}(n)) \le 0$. ❐

Theorem A4.4: $\mathrm{E}\,[\tilde{\underline{L}}^T(n+1)\tilde{\underline{L}}(n+1) - \underline{L}^T(n)\underline{L}(n) \mid \underline{L}(n)] \le 2N, \quad \forall\, \underline{\lambda}$.

Proof:

$$\begin{aligned} &\tilde{\underline{L}}^T(n+1)\tilde{\underline{L}}(n+1) - \underline{L}^T(n)\underline{L}(n) \\ &= (\underline{L}(n) - \underline{S}(n) + \underline{A}(n))^T (\underline{L}(n) - \underline{S}(n) + \underline{A}(n)) - \underline{L}^T(n)\underline{L}(n) \\ &= 2\underline{L}^T(n)(\underline{A}(n) - \underline{S}(n)) + (\underline{S}(n) - \underline{A}(n))^T (\underline{S}(n) - \underline{A}(n)) \\ &= 2\underline{L}^T(n)(\underline{A}(n) - \underline{S}(n)) + k, \end{aligned} \tag{12}$$

where $0 \le k \le 2N$. $k \ge 0$ because $\underline{S}(n) - \underline{A}(n)$ is a real vector, and $k \le 2N$ because $\|\underline{S}(n) - \underline{A}(n)\|^2 \le 2N$.

Taking the expected value:

$$\begin{aligned} E\,[\tilde{\underline{L}}^T(n+1)\tilde{\underline{L}}(n+1) - \underline{L}^T(n)\underline{L}(n)\,\big|\,\underline{L}(n)] &\le E\,[2\underline{L}^T(n)\,(\underline{A}(n) - \underline{S}(n))\,] + 2N \\ &= 2\underline{L}^T(n)\,(\underline{\lambda} - \underline{S}^*(n)) + 2N. \end{aligned} \tag{13}$$

From Theorem 4.4 we know that $2\underline{L}^T(n)\,(\underline{\lambda} - \underline{S}^*(n)) \le 0$, proving the theorem. ❒

Theorem A4.5: $\mathrm{E}\,[\tilde{\underline{L}}^T(n+1)\tilde{\underline{L}}(n+1) - \underline{L}^T(n)\underline{L}(n)\,\big|\,\underline{L}(n)] \le -\varepsilon\|\underline{L}(n)\| + 2N$, $\varepsilon > 0$,

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

Proof:

$$\begin{aligned} \underline{L}^T(n)\,(\underline{\lambda} - \underline{S}^*(n)) &\le \underline{L}^T(n)\,\{\underline{\lambda}_m - \underline{S}^*(n)\} - \underline{L}^T(n)\,(\beta\underline{\lambda}_m) \\ &\le 0 - \beta\|\underline{L}\,(n)\| \cdot \|\underline{\lambda}_m\| \cos\theta \end{aligned} \tag{14}$$

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

We now show that $\cos\theta > \delta$ for some $\delta > 0$ whenever $\underline{L}(n) \neq \underline{0}$. First, we show that $\cos\theta > 0$. We do this by contradiction: suppose that $\cos\theta = 0$, i.e. $\underline{L}(n)$ and $\underline{\lambda}_m$ are orthogonal. This can only occur if $\underline{L}(n) = \underline{0}$, or if for some i, j, both $\lambda_{i,j} = 0$ and $L_{i,j}(n) > 0$, which is not possible: for arrivals to have occurred at queue $Q(i, j)$, $\lambda_{i,j}$ must be greater than zero. Therefore, $\cos\theta > 0$ unless $\underline{L}(n) = \underline{0}$. Now we show that $\cos\theta$ is bounded away from zero, i.e. that $\cos\theta > \delta$ for some $\delta > 0$. Because $\lambda_{i,j} > 0$ wherever $L_{i,j}(n) > 0$, and because $\|\underline{\lambda}\| \le \sqrt{N}$,

$$\cos\theta = \frac{\underline{L}^T(n)\underline{\lambda}}{\|\underline{L}(n)\|\,\|\underline{\lambda}\|} \ge \frac{L_{max}(n)\lambda_{min}}{\|\underline{L}(n)\|\sqrt{N}}, \tag{15}$$

where $\lambda_{min} = \min(\lambda_{i,j},\, 1 \le i, j \le N)$ and $L_{max}(n) = \max(L_{i,j}(n),\, 1 \le i, j \le N)$. Also,

$$\|\underline{L}(n)\| \le \left[N^2 L_{max}^2(n)\right]^{1/2} = NL_{max}(n), \tag{16}$$

and so $\cos\theta$ is bounded by

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

Therefore

$$\mathrm{E}\left[\tilde{\underline{L}}^T(n+1)\tilde{\underline{L}}(n+1) - \underline{L}^T(n)\underline{L}(n) \middle| \ \underline{L}(n)\right] \le -\frac{\beta\lambda_{min}}{\sqrt{N}}\|\underline{L}(n)\| + 2N \,. \ \square \tag{18}$$

Theorem A4.6: $\mathrm{E}\left[\underline{L}^T(n+1)\underline{L}(n+1) - \underline{L}^T(n)\underline{L}(n) \middle| \ \underline{L}(n)\right] \le -\varepsilon\|\underline{L}(n)\| + N^2 + 2N,\ \varepsilon > 0$

$\forall \ \underline{\lambda} \le (1-\beta)\,\underline{\lambda}_m(n), \quad 0 < \beta < 1 \,.$

Proof:

$$L_{i,j}(n+1) = \tilde{L}_{i,j}(n+1) + \begin{cases} 1 & \text{if } L_{i,j}(n) = 0, S_{i,j}(n) = 1 \\ 0 & \text{else} \end{cases}, \tag{19}$$

therefore

$$\underline{L}^T(n+1)\underline{L}(n+1) - \tilde{\underline{L}}^T(n+1)\tilde{\underline{L}}(n+1) \le N^2 \,, \tag{20}$$

and so

$$\mathrm{E}\left[\underline{L}^T(n+1)\underline{L}(n+1) - \underline{L}^T(n)\underline{L}(n) \middle| \underline{L}(n)\right] \le \mathrm{E}\left[\tilde{\underline{L}}^T(n+1)\tilde{\underline{L}}(n+1) - \underline{L}^T(n)\underline{L}(n) \middle| \underline{L}(n)\right] + N^2 \,. \tag{21}$$

Using Theorem 4.5 this concludes the proof. ❐

Theorem A4.7: *There exists a* $V(\underline{L}(n))$ *s.t.* $\mathrm{E}\left[V(\underline{L}(n+1)) - V(\underline{L}(n)) \middle| \underline{L}(n)\right] \le -\varepsilon\|\underline{L}(n)\| + k$, *where* $k, \varepsilon > 0$.

Proof: $V(\underline{L}(n)) = \underline{L}^T(n)\underline{L}(n)$ and $k = N^2 + 2N$ in Theorem 4.6.❐

We are now ready to prove the main theorem.

Proof of Main Theorem: $V(\underline{L}(n))$ in Theorem 4.7 is a quadratic Lyapunov function and, according to the argument of Kumar and Meyn [27], it follows that the switch is stable. ❐