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.

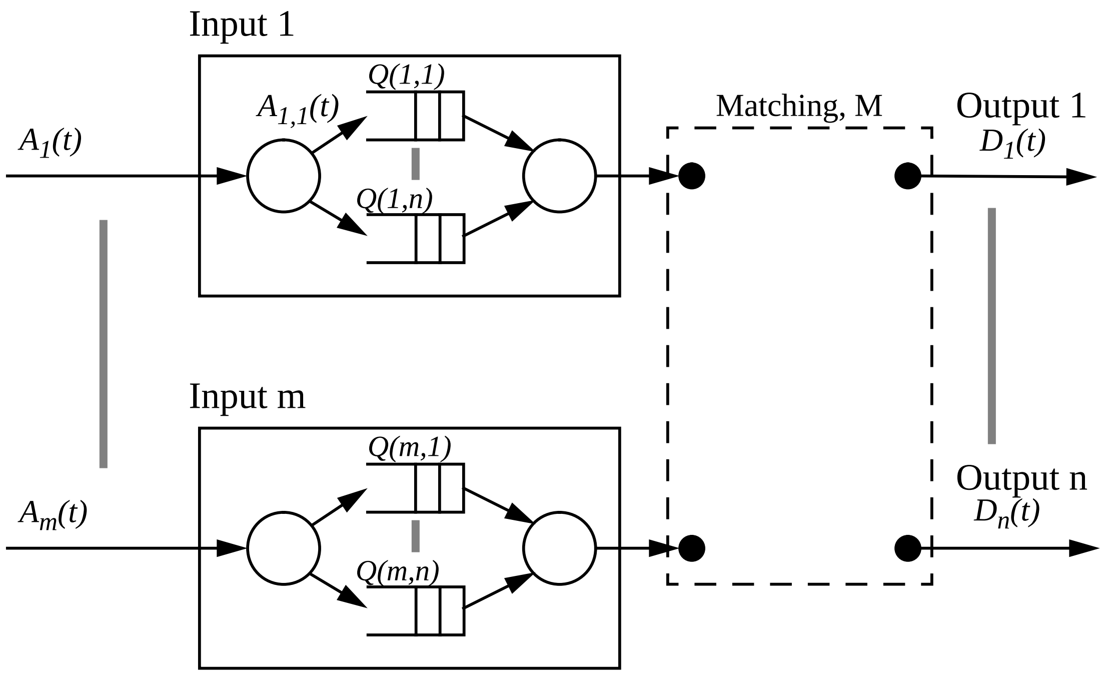


Figure 1.1 Components of an Input-Queued Cell-Switch.

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 $|I| = m = |J| = n = N$, 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

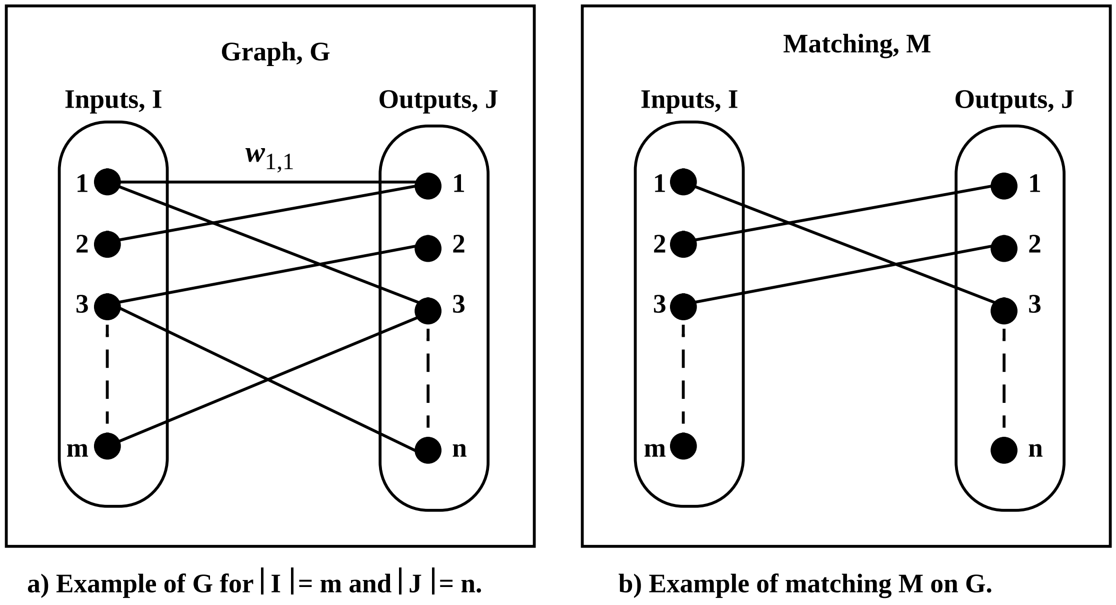


Figure 1.2 Define G = [V,E] as an undirected graph connecting the set of vertices V with the set of edges E. The edge connecting vertices *i*, $1 \le i \le m$ and *j*, $1 \le j \le n$ has an associated weight denoted $w_{i,j}$. Graph G is bipartite if the set of inputs I = {i: $1 \le i \le m$} and outputs 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

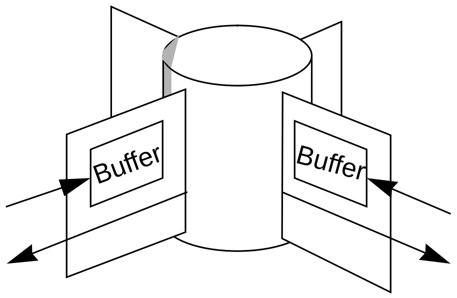
The scheduling algorithms described in this thesis are applicable to all input-queued switches. However, the work was motivated by a single goal: to find a simple algorithm that can schedule cells in a high-speed, parallel input-queued crossbar switch. We may partition such a switch into two main components: 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

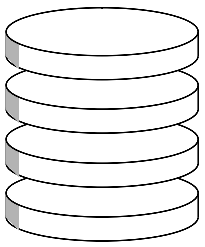
An example of a high-speed datapath is shown in Figure 1.3. This switch is shown to illustrate that it is feasible to build a small switch with extremely high aggregate bandwidth in current CMOS technology. Figure 1.3(a) shows the general structure of the switch: switch port cards connect to a central 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:

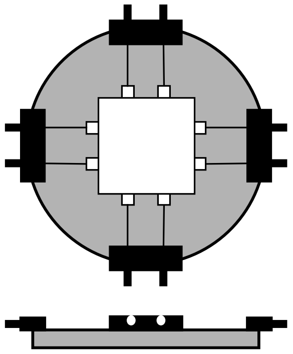
- The bit-slice is extremely simple. No leads need to cross, reducing crosstalk and allowing the slice to be constructed from a single layer printed circuit board.
- The lead lengths connecting each port card to the central switch are all of identical and minimum length. This reduces skew and the effect 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



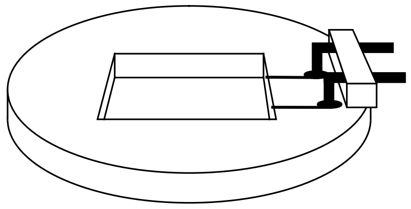
a) The switch consists of a central, vertical hub. Each interface card connects radially into the hub. This example shows a 4-port switch.



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.



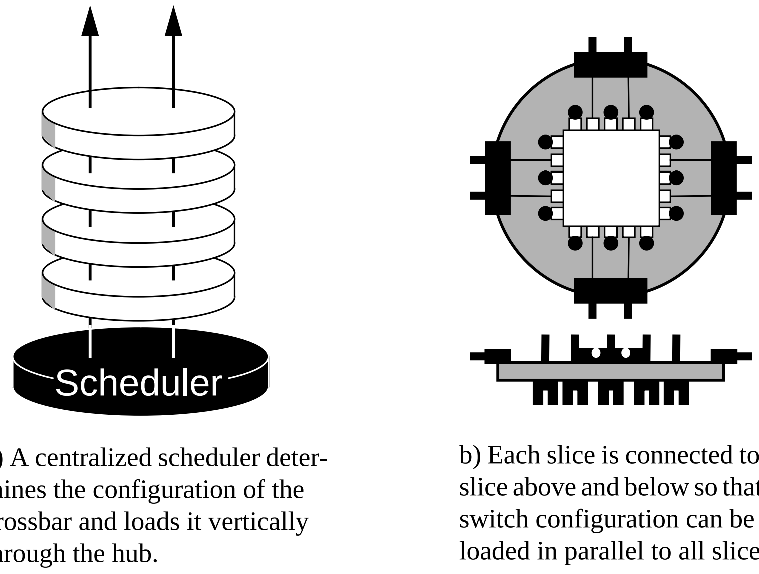
d) Detail of construction of bit-slice. This example only shows a single edge connector.

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.

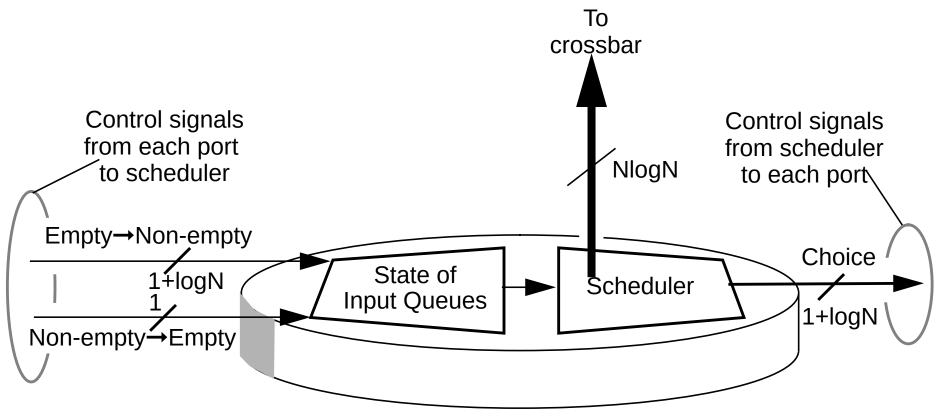


Figure 1.5 Connections to and from each port and a centralized scheduler.

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.

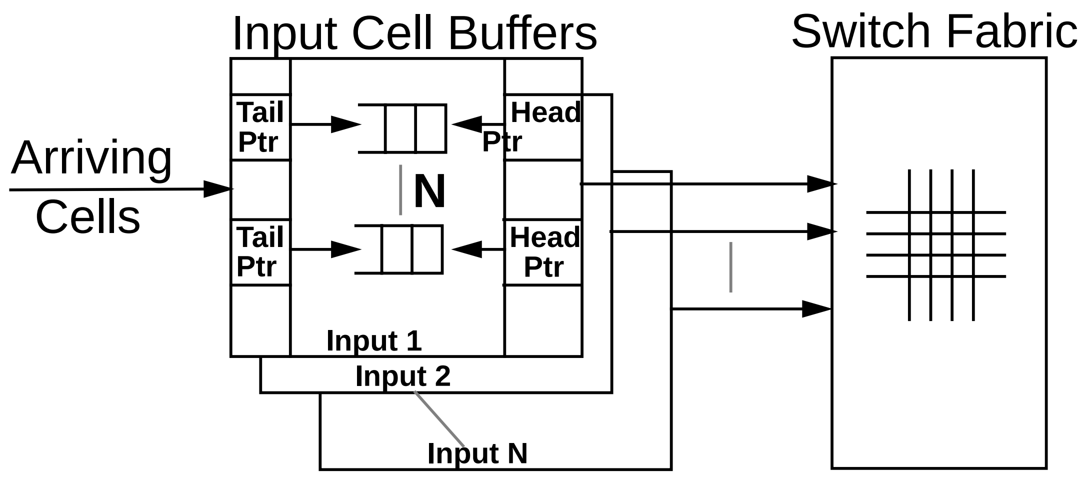


Figure 1.6 Head of line blocking can be eliminated by using a separate queue for each output at each input.

[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

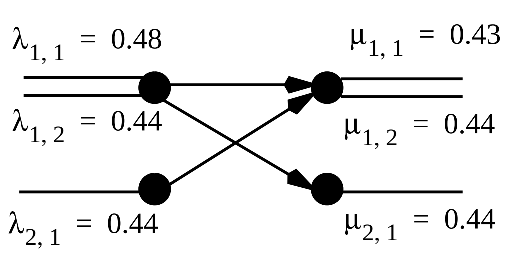
The maximum size matching for a bipartite graph can be found by solving an equivalent network flow problem [41]. We will call this algorithm *maxsize*. There exist many algorithms for solving these problems, the most efficient currently known converges in $O(n^{5/2})$ time and is described in [17].[2] 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.

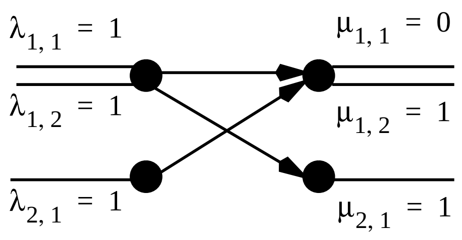
1. In some literature, the maximum *size* matching is called the maximum *cardinality* matching or just the maximum bipartite matching.

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]:

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

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.

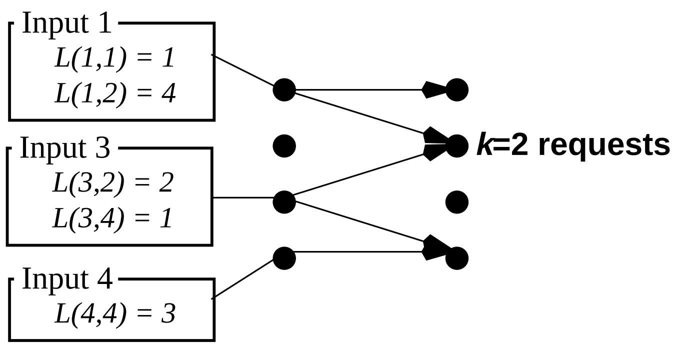
3.3.4 Parallel Iterative Matching

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.

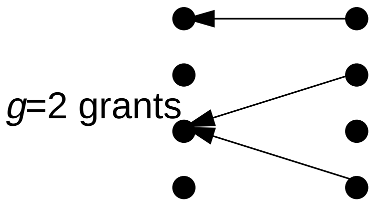
PIM uses *randomness* to avoid starvation, and to reduce the number of iterations needed to converge on a maximal matching. PIM attempts to quickly converge on a conflict-free match in multiple iterations, where each iteration consists of three steps. All inputs and outputs are initially unmatched and 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.

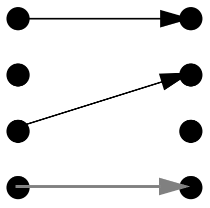
Step 2. *Grant*. If an unmatched output receives any requests, it grants to one by *randomly* selecting a request uniformly over all requests.



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

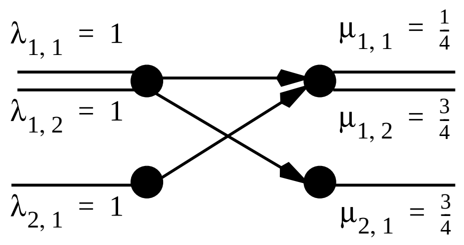

$\lambda_{1,1} = 1$
$\lambda_{1,2} = 1$
$\lambda_{2,1} = 1$
$\mu_{1,1} = \frac{1}{4}$
$\mu_{1,2} = \frac{3}{4}$
$\mu_{2,1} = \frac{3}{4}$

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.

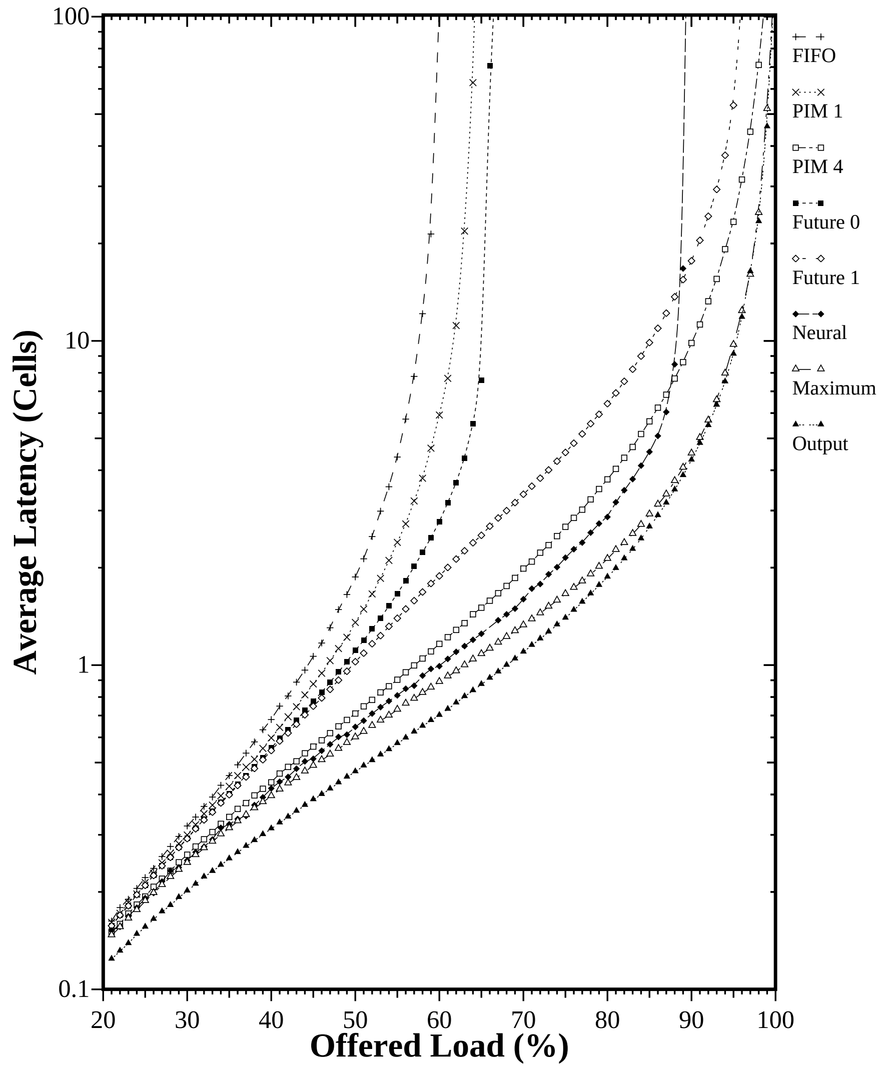


Figure 1.10 Comparison of latency as a function of offered load for several scheduling algorithms, using simulation. 16x16 switch, arrivals at each input are Bernoulli i.i.d. trials for each cell time. Cell destinations are uniformly distributed over all outputs. All arrival processes are independent.

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*.