Multicast Scheduling for Input-Queued Switches

Balaji Prabhakar
BRIMS
Hewlett-Packard Labs, Bristol.
balaji@hplb.hpl.hp.com

Nick McKeown and Ritesh Ahuja
Dept of Elec Engg/Comp Sc
Stanford University.
nickm@ee.stanford.edu, ritesh@cs.stanford.edu

Abstract

This paper presents the design of the scheduler for an $M \times N$ input-queued multicast switch. It is assumed that: (i) Each input maintains a single queue for arriving multicast cells, and (ii) Only the cell at the head of line (HOL) can be observed and scheduled at one time. The scheduler is required to be: (i) Work-conserving, which means that no output port may be idle as long as there is an input cell destined to it, and (ii) Fair, which means that no input cell may be held at HOL for more than a fixed number of cell times. The aim of our work is to find a work-conserving, fair policy that delivers maximum throughput and minimizes input queue latency, and yet is simple to implement in hardware. When a scheduling policy decides which cells to schedule, contention may require that it leave a *residue* of cells to be scheduled in the next cell time. The selection of where to place the residue uniquely defines the scheduling policy. Subject to a fairness constraint, we argue that a policy which always concentrates the residue on as few inputs as possible generally outperforms all other policies. We find that there is a tradeoff between concentration of residue (for high throughput), strictness of fairness (to prevent starvation), and implementational simplicity (for the design of *high-speed* switches). By mapping the general multicast switching problem onto a variation of the popular block-packing game, Tetris, we are able to analyze, in an intuitive and geometric fashion, various scheduling policies which possess these attributes in different proportions. We present a novel scheduling policy, called TATRA, which performs extremely well and is strict in fairness. We also present a simple weight based algorithm, called WBA, that is simple to implement in hardware, fair, and performs well when compared to a concentrating algorithm.

1 Introduction

Due to an exponential growth in the number of users of the Internet, the demand for network bandwidth has been growing at an enormous rate. As a result, recent years have witnessed an increasing interest in high-speed, cell-based, switched networks such as ATM. In order to build such networks, a high performance switch is required to quickly deliver cells arriving on input links to the desired output links. A switch consists of three parts: (i) Input queues to buffer cells arriving on input links, (ii) Output queues to buffer the cells going out on output links, and (iii) A switch fabric to transfer cells from the inputs to the desired outputs. The switch fabric operates under a scheduling algorithm which arbitrates among cells from different inputs destined to the same output. A number of approaches have been taken in designing these three parts of a switch [9, 20, 19, 17, 14, 16], each with its own set of advantages and disadvantages.

It is well known that when FIFO queues are used, the throughput of an input-queued switch with unicast traffic can be limited due to HOL blocking [4], [5]. So the standard approach has been to abandon input queueing and instead to use output queueing - by increasing the bandwidth of the fabric, multiple cells can be forwarded at the same time to the same output, and queued there for transmission on the output link. However this approach requires that the output-queues and the internal interconnect have a bandwidth equal to M times (for an $M \times N$ switch) the line rate. Since memory bandwidth is not increasing as fast as the demand for network bandwidth, this architecture becomes impractical for very high-speed switches. Moreover, numerous papers have indicated that by using non-FIFO input queues and by using good scheduling policies, much higher throughputs are possible [9, 10, 11, 12, 13, 14, 16, 17]. Therefore, input-queued switches are finding a growing interest in the research and development community.

An increasing proportion of traffic on the Internet is multicast, with users distributing a wide variety of audio and video material. This dramatic change in the use of the Internet has been facilitated by the MBONE [1, 2, 3]. A number of different architectures and implementations have been proposed for multicast switches [6, 7, 8]. However, since we are interested in the design of very high-speed ATM switches, we restrict our attention to input-queued architectures. This input-queued switch should schedule multicast cells so as to maximize throughput and minimize latency. It is important that it be simple to implement in hardware. For example, a switch running at a line rate of 2.4Gbps (OC48c) must make 6 million scheduling decisions every second.

In this paper we consider the performance of different multicast scheduling policies for input-queued switches. Several researchers have studied the Random scheduling policy [9, 18, 21, 22] in which each output selects an input at random from among those subscribing to it. But, as may be expected, we find that the Random scheduling policy is not the optimum policy. We introduce three new scheduling algorithms; the Concentrate algorithm, TATRA and WBA (a weight based algorithm). We show that the Concentrate algorithm leads to high throughput and low delay. It achieves this by concentrating the cells that it leaves behind on as few inputs as possible. Unfortunately, Concentrate has two drawbacks that make it unsuitable for use in an ATM switch; it can starve input queues indefinitely, and is difficult to implement in hardware. But Concentrate serves as a useful upper-bound on throughput performance against which we can compare heuristic approximations. One such approximation, TATRA, is motivated by Tetris, the popular block-packing game. TATRA avoids starvation by using a strict definition of fairness, while comparing well to the performance of Concentrate. The second algorithm, WBA is designed to be very simple to implement in hardware, and allows the designer to balance the tradeoff between fairness and throughput.

2 Background

2.1 Assumed Architecture

It is assumed that the switch has M input and N output ports and that each input maintains a single FIFO queue for arriving multicast cells. The input cells are assumed to contain a vector indicating which outputs the cell is to be sent to. For an $M \times N$ switch, the destination vector of a multicast cell can be any one of $2^N - 1$ possible vectors. We assume that each input has a single queue and that the scheduler only observes the first cell in the queue.

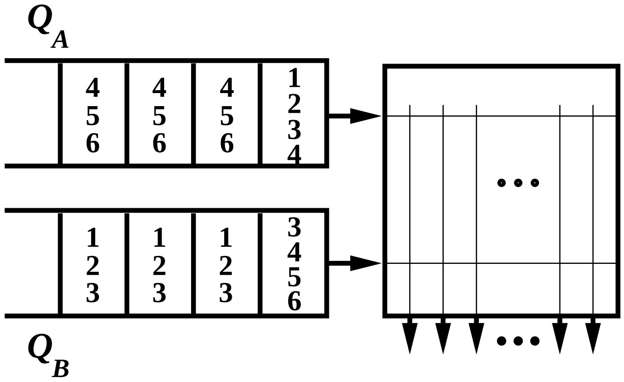

Figure 1: *2×N multicast crossbar switch with single* FIFO *queue at each input.*

As a simple example of our architecture, consider the 2 input and N output switch shown in Figure 1. Queue Q_A has an input cell destined for outputs $\{1, 2, 3, 4\}$ and queue Q_B has an input cell destined for outputs $\{3, 4, 5, 6\}$. The set of outputs to which an input cell wishes to be copied will be referred to as the *fanout* of that input cell.[1] For clarity, we distinguish an arriving *input* cell from its corresponding *output* cells. In the figure, the single input cell at the head of queue Q_A will generate four output cells.

We assume that an input cell must wait in line until all of the cells ahead of it have departed. A simple way to service the input queues is to replicate the input cell over multiple cell times, generating one output cell per cell time. However, this approach has two disadvantages. First, each input must be copied multiple times, increasing the required memory bandwidth. Second, input cells contend for access to the switch multiple times, reducing the bandwidth available to other traffic at the same input. Higher throughput can be attained if we take advantage of the natural multicast properties of a crossbar switch. So instead, we assume that one input cell can be copied to any number of outputs in a single cell time for which there is no conflict.

There are two different service disciplines that can be used. Following the description in [18], the first is *no fanout-splitting* in which all of the copies of a cell must be sent in the same cell time. If any of the output cells loses contention for an output port, none of the output cells are transmitted and the cell must try again in the next cell time. The second discipline is *fanout-splitting*, in which output cells may be delivered to output ports over any number of cell times. Only those output cells that are unsuccessful in one cell time continue to contend for output ports in the next cell time[2].

Because fanout-splitting is work conserving, it enables a higher switch throughput [21] for little increase in implementation complexity. For example, Figure 2 compares the average cell latency (via simulations) with and without fanout-splitting of the Random scheduling policy for an 8×8 switch under uniform loading on all inputs and an average fanout of four. The figure demonstrates that fanout-splitting can lead to approximately 40% higher throughput.

[1] We use the term fanout throughout this paper to denote both the constitution and the cardinality of the input vector. For example, in Figure 1, the input cell at the head of each queue is said to have a fanout of four.

[2] It might appear that *fanout-splitting* is much more difficult to implement than *no fanout-splitting*. However this is not the case. In order to support *fanout-splitting*, we need one extra signal from the scheduler to inform each input port when a cell at its HOL is completely served.

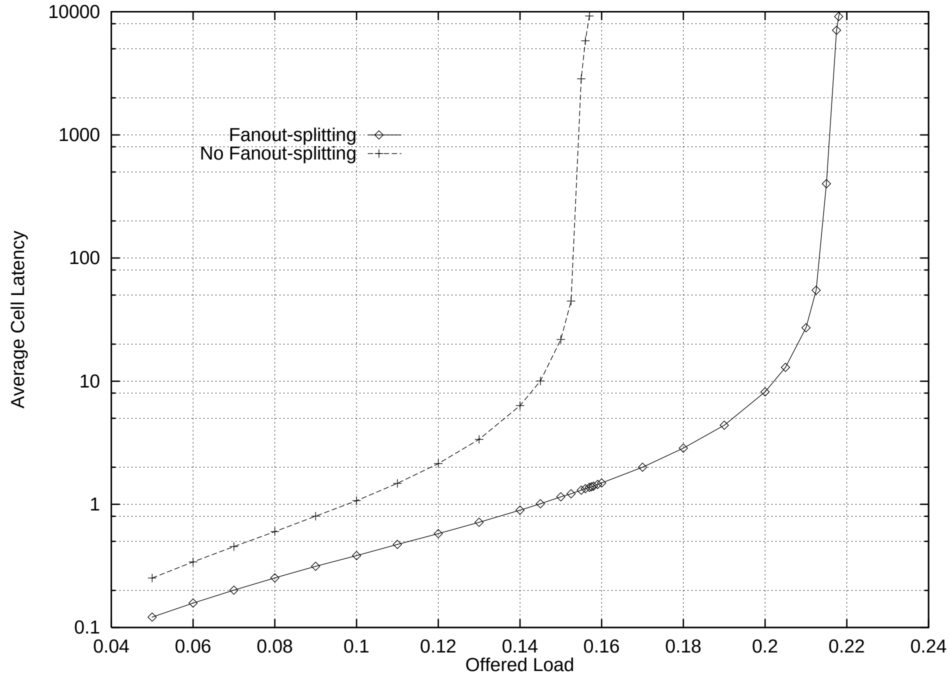

Figure 2: *Graph of average cell latency (in number of cell times) as a function of offered load for an 8×8 switch (with uniform input traffic and average fanout of four). The graph compares Random scheduling policy with and without fanout-splitting.*

2.2 Definition of Terms

Here we make precise some of the terminology used throughout the paper. Some terms have already been loosely defined, but a few new ones are introduced.

Definition 1 (Residue): *The* residue *is the set of all output cells that lose contention for output ports and remain at the* HOL *of the input queues at the end of each cell time.*

It is important to note that given a set of requests, every work-conserving policy will leave the same residue. However, it is up to the policy to determine how the residue is distributed over the inputs.

Definition 2 (Concentrating Policy): *A multicast scheduling policy is said to be* concentrating *if, at the end of every cell time, it leaves the residue on the smallest possible number of input ports.*

Definition 3 (Distributing Policy): *A multicast scheduling policy is said to be* distributing *if, at the end of every cell time, it leaves the residue on the largest possible number of input ports.*

Definition 4 (A Non-concentrating Policy): *A multicast scheduling policy is said to be* non-concentrating *if it does not always concentrate the residue.*

Definition 5 (Fairness Constraint): *A multicast scheduling policy is said to be* fair *if each input cell is held at the* HOL *for no more than a fixed number of cell times (this number* *can be different for different inputs). This fairness constraint can also be thought of as a starvation constraint.*

2.3 Requirements of an Algorithm

Before describing the details of various scheduling algorithms, we first look at some requirements.

1. **Work conservation:** The algorithm *must* be work conserving, which means that no output port may be idle as long as it can serve some input cell destined to it. This property is necessary for an algorithm to provide maximum throughput.

2. **Fairness:** The algorithm *must* meet the fairness constraint defined above, i.e. it must not lead to the starvation of any input.

3 The Heuristic of Residue Concentration

In this section, we describe two algorithms - the Concentrate algorithm and the Distribute algorithm, which represent the two extremes of residue placement. We present an intutive explanation for why it is best to concentrate residue in order to achieve a high throughput.

Algorithm: Concentrate. Concentrate always concentrates the residue onto as *few* inputs as possible. This is achieved by performing the following steps at the beginning of each cell time.

1. Determine the residue.
2. Find the input with the most in common with the residue. If there is a choice of inputs, select the one with the input cell that has been at the HOL for the shortest time. This ensures some fairness, though not in the sense of the definition in Section 2.2 (see remark below).
3. Concentrate as much residue onto this input as possible.
4. Remove the input from further consideration.
5. Repeat steps (2)-(4) until no residue remains.

Remark: Since an input cell can remain at HOL indefinitely, this algorithm does not meet the fairness constraint. The purpose of this algorithm is to provide us with a basis for comparing the performance of other algorithms, since it achieves the highest throughput. This is demonstrated by our simulation results in Section 7.

Algorithm: Distribute. Distribute always distributes the residue onto as *many* inputs as possible.

1. Determine the residue.
2. Find the input with at least one cell but otherwise the least in common with the residue. If there is a choice of inputs, select the one with the input cell that has been at the HOL for the shortest time.
3. Place one output cell of residue onto that input.
4. Remove the input from further consideration.
5. Repeat steps (2)-(4) until no inputs remain.
6. If residue remains, consider all the inputs again and start at step (2).

Let us look at an example to see how these two algorithms work. Referring to Figure 1, consider the options faced by a work-conserving scheduling algorithm at this time (t_1). Note that whatever decision the algorithm makes, the residue will be the same. The scheduling algorithm just determines where to place the residue. If at time t_1, the algorithm concentrates the residue on Q_B then all of a_1's (also see figure 12) output cells will be sent and cell a_2 will be brought forward at time t_2. At time t_2, the algorithm selects between a_2 *and* the residue left over from t_1. If, on the other hand, the algorithm distributes the residue over both input queues at t_1, then at t_2 the algorithm can only schedule the residue left over from t_1. No new cells can be brought forward.

From the example above, we can make the following intuitive argument: it is more likely that Concentrate will bring new work forward sooner, thus increasing the diversity of its choice. This enables more output cells to be scheduled in the following cell time. For the case of a $2 \times N$ switch, the following Theorem is true.

Theorem 1 *A scheduling policy for a $2 \times N$ multicast switch that always concentrates residue at every possible instant subject to the fairness condition of Definition 12, performs better than any other fair policy when subjected to static inputs.*

Proof: See Appendix A.□

4 Tetris Models for $M \times N$ Switches

This section presents a unified approach to the design and analysis of schedulers for an $M \times N$ multicast switch. It is shown that the general multicast scheduling problem can be mapped onto a variation of the popular block-packing game Tetris. Within this common framework, one is able to describe and analyze any multicast scheduling policy in an intuitive and geometric fashion. The presentation in this section follows earlier work presented in [24] and [25].

We first describe the class of scheduling policies to be considered in this section, all of which are required to satisfy the following fairness constraint.

Definition 6 (Fairness Constraint for $M \times N$ Switches): *A scheduling policy π for a $M \times N$ switch is said to be* fair *if no cell, from any input, is held at* HOL *for more than M cell times.*

The class of policies considered: In addition to requiring that policies be fair and work-conserving, we also require that they assign departure dates to input cells once the cells advance to HOL. This *departure date* (DD) is some number between 1 and M specifying how long, from the current cell time, the input cell will be held at HOL before being discharged. The DD of a cell is decremented by 1 at the end of each cell time. Clearly, this class of policies is smaller than the class of fair and work-conserving policies, since fairness allows one to reassign departure dates to input cells at HOL (but not beyond M cell times).

We use the "static input assumption" to describe the Tetris models. As will become clear, this description holds equally well for "dynamic inputs" since scheduling is based only on cells at HOL without look-ahead.

4.1 Tetris models: a sketch

Every input cell is mapped onto a Tetris block, each of which is an amalgamation of smaller blocks, one per output cell. Upon assignment of DDs, the input cells at HOL are dropped into a compartmentalized box of size $M \times N$, see Figure 3. Each of the N columns of the box holds cells destined to a specific output; i.e., column j holds cells destined to output j. The label on a cell denotes the input port from which it has arrived; all output cells with the same label result from the same input cell. The cells in the bottom-most row of the box in Figure 3 at columns 1, 3 and 5 are all identical copies of a cell from input 1 destined to outputs 1, 3 and 5. Similarly, the cell at the HOL of input 2 will be delivered to outputs 2, 3, 4 and 5.

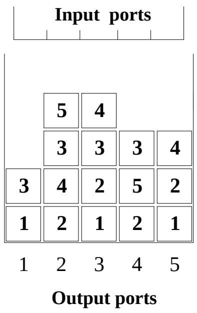

Figure 3: *An example. Cells from inputs 1, 2, 3, 4 are assigned DDs 1, 2, 3, 4 respectively, while the cell from input 5 is assigned a DD of 4.*

Suppose that, at time n, the switch is to schedule k input cells which have advanced to HOL. After the scheduler has assigned DDs to these input cells, they are dropped into the box which currently holds the cells or residues at the HOL of the other $(M - k)$ inputs[3]. Each new output cell may occupy any position in its appropriate output slot as long as (1) it does not alter the DD of any other cell, and (2) it does not leave any slots beneath it unoccupied. Note that there are no unoccupied slots between cells in any output column.

At the end of time n, all output cells at the bottom-most layer of the box are discharged and are assumed to be served. For the example in Figure 3, input 1 is completely served and can advance a new cell to HOL at time 2. Input 2 discharges cells to outputs 2 and 4 and is left with a residue for outputs 3 and 5. Note that the *discharge* at any time is the set of output cells in the bottom-most layer and the *residue* is everything that's left behind. It should now be clear that we do not allow unoccupied slots in output columns because of the restriction to policies which are work-conserving.

At the beginning of time $n + 1$, all residue cells drop down one level and their DDs are decremented by one. Those inputs which have been completely served in the previous cell time advance a new cell to the HOL. These cells are assigned DDs, and the cycle continues.

This is reminiscent of Tetris where blocks are dropped into a bin and the aim is to achieve maximum packing. The main difference here is that Tetris blocks are rigid and

[3]The order in which the scheduler assigns DDs to the k new cells is important, because if the cells contend for the same outputs it may not be possible to assign them DDs in parallel. For example, suppose that two of the new cells have a fanout of 1 and are the only cells contending for a specific output. Then, deciding who goes first is important since no two cells in an output column can have the same DD. In general, the order of assignment of DDs can either be pre-fixed or made to depend upon some criterion (e.g., size of fanout). However, for ease of exposition, we will assume a pre-fixed ordering.

cannot be decomposed. Note also that *there are never more than M input cells in the box.* Thus when an input cell is dropped into the box, it is *guaranteed to depart within M cell times*, since input cells arriving in the future do not alter its departure date. This automatically ensures fairness.

4.2 Tetris models: the details

We now make the description of Tetris models mathematically precise. If a plurality of cells advance to HOL at the beginning of a cell time, we choose, for simplicity, the following fixed ordering: for $i < j$ the new cell at input i will be assigned its DD before the new cell at input j. Before proceeding to define a scheduling algorithm, we make the following definitions.

Definition 7 (Tetris Box): *The* Tetris box *is specified by a matrix $T_{i,j}, 1 \leq i \leq M, 1 \leq j \leq N$, where the rows are numbered from* bottom to top *and the columns are numbered from left to right. Thus $T_{1,1}$ is the bottom-left slot of the box and $T_{M,N}$ is the top-right slot.*

Definition 8 (Occupancy Set): *The* occupancy set *of the cell or residue at the* HOL *of input l at time n is given by $O_l(n) = \{T_{i,j} :$ an output cell of l resides at $T_{i,j}$ at time $n.\}$*

Definition 9 (Peak Cell and Departure Date): *An output cell belonging to input l is said to be a* peak cell *at time n if it occupies a slot in the row whose number is given by* $\max\{i : T_{i,j} \in O_l(n)\}$. *The corresponding row number is the* departure date (DD) *of the input cell at time n.*

That is, the peak cell of an input is one which is furthest from the bottom of the box and the distance from the bottom is its departure date. Note that there may be more than one peak cell for a given input.

Definition 10 (Scheduling Policy): *Given $k \leq M$ new cells $c_1, c_2, \cdots, c_k$ at the* HOL *of inputs $i_1 < i_2 < \cdots < i_k$ at time n, a* scheduling policy Π *is given by a sequence of decisions $\{\pi(n), n \in \mathcal{Z}^+\}$, where $\pi(n)$ associates to each of $c_1, c_2, \cdots, c_k$ (in that order) the corresponding occupancy sets $O_{c_1}(n), O_{c_2}(n), \cdots, O_{c_k}(n)$ subject to the following rules.*

1) No cell should change the DD of a cell that is already scheduled. This means that no peak cells should be raised or lowered.

2) For every $i > 1$ and j, if any *of $T_{i,j}$ is occupied, then so are $T_{1,j} \cdots T_{i-1,j}$; i.e., there should be no gaps in the output columns.*

Algorithm for Π. Given the above definitions, the algorithm for implementing a policy Π just requires a specification for transitioning from one cell time to the next. The following steps enumerate the procedure.

a) At the end of time n, all output cells occupying slots in the set $\{T_{1,j}, 1 \leq j \leq N\}$ are discharged. In particular, input cells (or residues thereof) with DDs $= 1$ are completely served.

b) Each output cell occupying slot $T_{i,j}$ for i and j in the set $\{2 \leq i \leq M, 1 \leq j \leq N\}$ is assigned to the slot $T_{i-1,j}$. The occupancy set, peak cell(s), and the departure dates of the residue are recomputed. For example, the occupancy set of the residue at input l is given by $O_l(n+1) = \{T_{i,j} : T_{i+1,j} \in O_l(n)\}$. From this peak cells and DDs are easily computed.

c) New cells advancing to HOL are then scheduled according to $\pi(n+1)$.

Consider the example of Figure 3 again. The input cells are scheduled in the order 1, 2, 3, 4 and 5. The occupancy sets, peak cells and departure dates at time 1 are given in the table below.

Input Port l	Occupancy Set $O_l(1)$	Peak Cells $PC_l(1)$	Departure Date $DD_l(1)$
1	$\{T_{1,1}, T_{1,3}, T_{1,5}\}$	$O_1(1)$	1
2	$\{T_{1,2}, T_{2,3}, T_{1,4}, T_{2,5}\}$	$\{T_{2,3}, T_{2,5}\}$	2
3	$\{T_{2,1}, T_{3,2}, T_{3,3}, T_{3,4}\}$	$O_3(1) - \{T_{2,1}\}$	3
4	$\{T_{2,2}, T_{4,3}, T_{3,5}\}$	$\{T_{4,3}\}$	4
5	$\{T_{4,2}, T_{2,4}\}$	$\{T_{4,2}\}$	4

As a final remark, the discussion in this section presents a unified framework for thinking about multicast scheduling policies. We have seen how constraints like fairness and work-conservation translate into rules for placing Tetris blocks in the box. This general framework allows us to design and evaluate the performance of specific scheduling algorithms.

5 TATRA: A multicast scheduling algorithm

Motivated by the Tetris models of the previous section, we now describe a specific multicast scheduling algorithm, TATRA, first introduced in [24] and discuss some of its salient features.

Again we assume that the switch has been idle prior to time 0 and that the "static input assumption" holds. We denote by $\Pi^* = \{\pi^*(n), n \in \mathcal{Z}^+\}$ the policy TATRA. Since TATRA schedules input cells solely based on their DDs, we assume that this number is stamped upon all the output cells belonging to a specific input cell (both peak and non-peak cells).

For time $n \geq 1$, the algorithm is specified by the following steps.

(1) At the beginning of time n, $\pi^*(n)$ assigns a DD to each new cell at HOL according to the formula given in Equation 1 below. The order in which the DD is assigned when there is a plurality of new cells is in increasing order of their input port numbers.

(2) Each new output cell is dropped to the lowest possible level in the appropriate output slot, without getting ahead of another cell whose DD is less than or equal to its own.

Remark: It follows that a non-peak cell cannot be ahead of a peak cell unless it has the same DD as the peak cell. If such a non-peak cell exists, we call it a *pseudo-peak cell* (an example of a pseudo-peak cell is given below). Corresponding to each output slot, there is thus a (possibly empty) column of peak/pseudo-peak cells. This column is called the *peak column.*

(3) Cells in the bottom-most row are discharged. New DDs are computed for the residue cells. Time is advanced to $n+1$.

Using the terminology introduced in the remark above, and from the constitution of a new input cell its DD is computed as follows

$$\text{DD} = \max\{\text{height of peak columns across fanout}\} + 1 \tag{1}$$

5.1 An Example

By applying the above algorithm to the example of Figure 3, it is easy to see that TATRA schedules the cells as shown in Figure 4a. Assuming that at the end of time 1 the two new cells at inputs 1 and 5 wish to access ouputs $\{1,5\}$ and $\{2\}$ respectively, Figure 4b shows how TATRA schedules them. Observe that in Figure 4b, the cell from input 3 at position $T_{1,1}$ is a pseudo-peak cell because the cell at input 3 has a DD equal to 2 which is the same as the cell from input 1. Therefore, the height of the peak column corresponding to output 1 in Figure 4b is equal to 2.

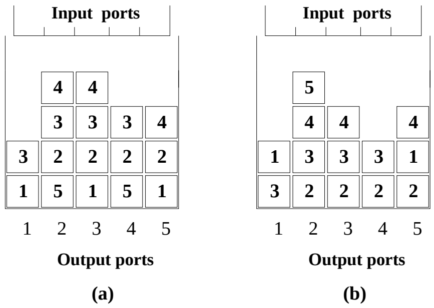

Figure 4: *TATRA schedules:* (a) *the cells of Figure 3*, (b) *the new cells from inputs 1 and 5 at time 2.*

5.2 Properties of TATRA

In this subsection we discuss some desirable properties of TATRA. For brevity, the properties are stated and only briefly explored.

Property 1: Under TATRA an input cell is guaranteed to be a discharged every cell time. This is equivalent to the statement that there is a peak cell in every row of the Tetris box. To see this, merely observe that (1) under every peak cell there is a column of peak (or pseudo-peak) cells, and (2) the cell furthest from the bottom of the box must be a peak cell.

Property 2: Residue Concentration. Suppose that we are given the occupancy sets, $O_l(n)$ and $O_m(n)$, of two input cells l and m. If $T_{i,j} \in O_l(n)$ and $T_{i+k,j} \in O_m(n)$ for some j and for some $k > 0$, then it is impossible that there exists an output $j' \neq j$ such that $T_{i',j'} \in O_l(n)$ and $T_{i'-k',j'} \in O_m(n)$, where $k' > 0$. That is occupancy sets cannot "criss-cross". This follows from the fact that output cells are arranged in output columns according to a monotonic increase of DDs. The "no criss-crossing" property corresponds to residue concentration.

6 WBA

Although it performs well and is simple to describe, there are two disadvantages to TATRA. First, it is difficult to implement, since the assignment of DDs at inputs requires a collective effort and this process cannot be parallelized. Second, the definition of fairness is both rigid (i.e., no input cell should be held at HOL for more than M cell times), and uniformly the same for all inputs. Treating all inputs uniformly does not help when the inputs are non-uniformly loaded or when some inputs request a higher priority.

These issues motivate us to look for an algorithm that (i) is simple to implement in hardware, (ii) is fair and achieves a high throughput, and (iii) is able to cope with non-uniform loading and/or provide different priorities to inputs. A weight based algorithm, called WBA, is introduced in this section and is shown to meet the above requirements.

It is worth mentioning that if one merely wishes to achieve a high throughput without regard to fairness, then it is best to always achieve the highest residue concentration. But this can lead to the starvation of some inputs. For example, in the Concentrate algorithm, an input cell with maximum fanout may wait forever without being served. Conversely, if an algorithm aims to be fair, it may not achieve the best possible residue concentration and therefore sacrifices throughput.

WBA: The Weight Based Algorithm

This algorithm works by assigning weights to input cells based on their age and fanout at the beginning of every cell time. Once the weights are assigned, each output chooses the heaviest input from among those subscribing to it. It is clear that a positive weight should be given to age in order to achieve fairness. We claim that to maximize throughput, fanout should be weighted negatively. To see this, recall that at the end of each cell time the output cells in the bottom-most row of the Tetris box are discharged and all other cells are left behind as residue. To improve residue-concentration we must therefore ensure that as many *input cells* as possible can be placed in the bottom-most row at every cell time. This automatically ensures that the residue is concentrated on fewer inputs. Since the bottom-most row can only take N output cells one has to choose input cells with the smallest fanout to place on this row. Thus, the weight of an input cell should vary inversely as its fanout.

Algorithm: WBA.

1. At the beginning of every cell time, each input calculates the weight of the new cell/residue at its HOL based on:
   (a) The age of the cell/residue: The older, the heavier.
   (b) The fanout of the cell/residue: The larger, the lighter.
2. Each input then submits this weight to all the outputs that the cell/residue at its HOL wishes to access.
3. Each output grants to the input with the highest weight, independently of other outputs, ties being broken randomly.

By making a suitable choice of weights based on these two quantities (age and fanout), one arrives at a compromise between the extremes of pure residue concentration and of strict fairness. Simple calculations show that if we give weight a to the age of the cell, and weight $(-f)$ to the fanout, the bound on the time for which a cell has to wait at HOL is simply $(M + \frac{f}{a}N - 1)$ cell times. In particular, if we give equal weight to age and to fanout, no cell waits at the HOL for longer than $(M + N - 1)$ cell times. And if the negative weight of fanout is twice the weight of the age then one increases residue concentration and decreases fairness, allowing a cell to wait at the HOL upto $(M + 2N - 1)$ cell times.

Many variations of the WBA are possible. In particular, one can use other features to assign weights to the cells. For example, one can take into account input queue occupancy while computing weights, or keep track of the utilization of each output link and use negative weight to discourage inputs subscibing to heavily loaded outputs. When dealing with non-uniform loading or when offering different priorities to different inputs, one can use different formulae to compute weights at different inputs. However, these weights should be within the proper range to ensure stability, i.e. the range of possible values for the weights should be the same on all the inputs.

7 Simulation Results

In order to validate our claims in the previous sections, we compare different scheduling policies through simulation. The switch behaviour is simulated by using a discrete-event simulator written for the purpose. Our simulated switch is assumed to have infinite buffers at the inputs so that no cells are dropped due to lack of buffer space. In each simulation run, there is a sufficient warmup period (typically half of the total simulation time) to allow the input buffers to be filled up with cells before statistics about the queue lengths and cell latencies are collected. The simulation continues for a fixed amount of simulation time (typically 1 million cell times) unless the switch becomes unstable (i.e. it reaches a stage where it is unable to sustain the offered load).

7.1 Traffic Types

We assume that the stream of arrivals at the inputs are independent of each other. We compare each scheduling policy for two different arrival processes:

Uncorrelated Arrivals: At the beginning of each cell time, a cell arrives at each input with probability p (the "arrival rate") independently of whether a cell arrived during the previous cell time.

Correlated Arrivals: Cells are generated using a 2-state Markov process which alternates between BUSY and IDLE states. The process remains in the busy and idle states for a geometrically distributed number of cell times, with expected duration $E[B]$ and $E[I]$ respectively. $E[B]$ is fixed at 16 cells for all the simulations.[4] When in this state, cells arrive at the beginning of every cell time and all with the same set of destinations. The arrival rate, $p = E[B]/(E[B] + E[I])$.

For both types of traffic, each arriving multicast cell has a multicast vector that is uniformly distributed over all possible multicast vectors. However, the destination vector of all *zeroes* is not allowed. As a result, for an $M \times N$ switch, the average fanout is $(N+1)/2$. The offered load is the fraction of link bandwidth used at each input by the incoming traffic. Since the average fanout is $(N+1)/2$ on *each* of the M inputs, the total traffic load of all outputs combined is $p * M * (N+1)/2$. Since this traffic is uniformly divided among all the outputs, the load as seen on each of the output links is $(pM/2)(1+1/N)$. Thus for a $2 \times N$ switch, the offered load shown in the graphs is the actual load, whereas for an 8×8 switch, the total switch load is approximately *four* times the offered load shown in the graphs.

For comparison, we also show the performance of an algorithm, *Random*, in which each output randomly selects one input from among those that request it. This algorithm is motivated by the work of Hayes et al. in [18], which is the multicast version of the unicast

[4] The choice of an expected duration of 16 cells per burst is arbitrary, but is representative. The same qualitative results are obtained for different burst lengths.

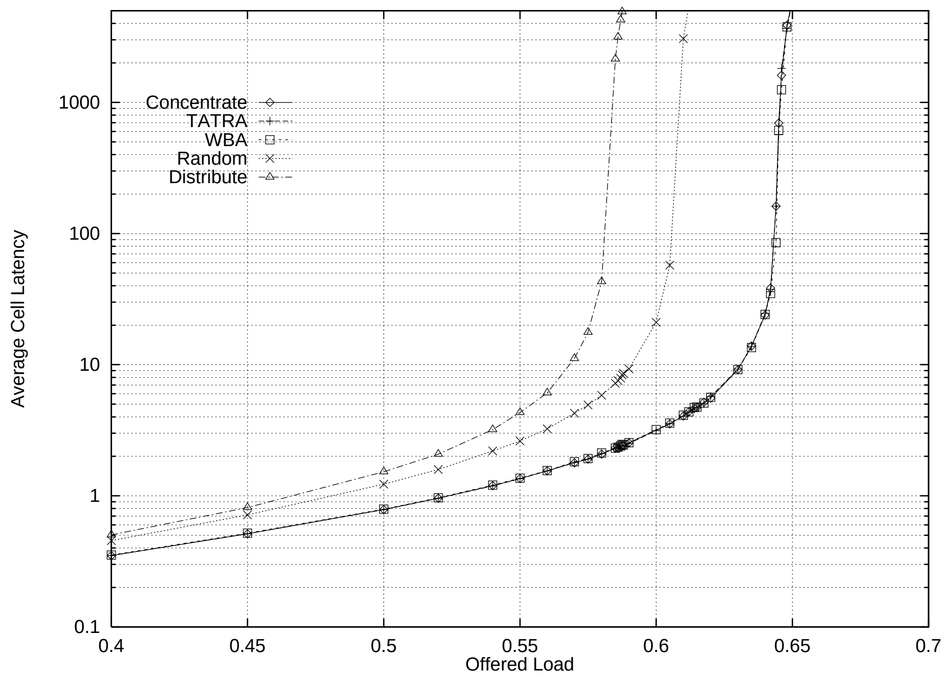

Figure 5: *Graph of average cell latency (in number of cell times) as a function of offered load for a 2×8 switch (Uncorrelated arrivals with an average fanout of four).*

algorithm described in [4]. The WBA plots are obtained by using a fanout weight equal to twice the weight for cell age.

7.2 2×8 Switch

Figures 5 and 6 compare the different scheduling policies for a 2×8 switch, with uncorrelated and correlated arrivals respectively. As predicted by Theorem 1, the *Concentrate* algorithm leads to an average cell latency that is lower than for the *Distribute* algorithm.

The algorithms also differ in the maximum possible throughput sustainable by the switch. As expected, the algorithm that leads to lower cell latency through the switch also provides higher throughput.

7.3 8×8 Switch

Figures 7 and 8 compare the different scheduling policies for an 8×8 switch, with uncorrelated and correlated arrivals, respectively. Once again, the *Concentrate* algorithm leads to an average cell latency that is much lower than for the *Distribute* algorithm.

Note that for an 8×8 switch TATRA performs worse than *Concentrate*. This is because it does not necessarily concentrate the residue on the minimum number of inputs. WBA performs a little worse than TATRA for uncorrelated arrivals even though TATRA provides a stricter bound on the HOL latency. The reason for this relatively poor performance of WBA is that the outputs make their decision independently. So, if two or more inputs have the same weight, different outputs will not concentrate the residue onto the same input. As a result, WBA is not as effective in concentrating residue as TATRA. Thus WBA sacrifices

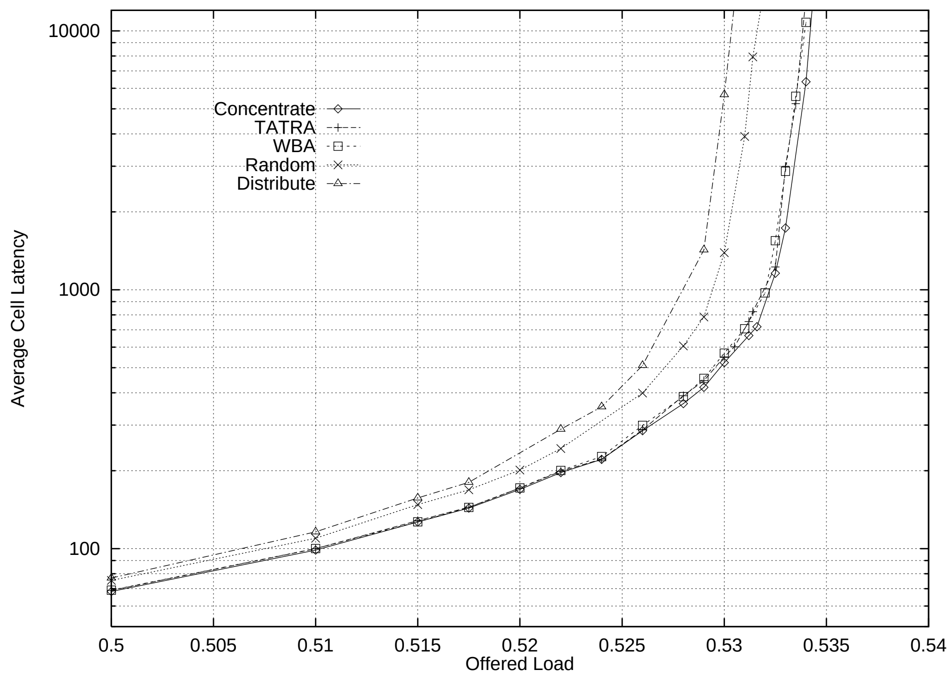

Figure 6: *Graph of average cell latency (in number of cell times) as a function of offered load for a 2×8 switch (Correlated arrivals with an average fanout of four).*

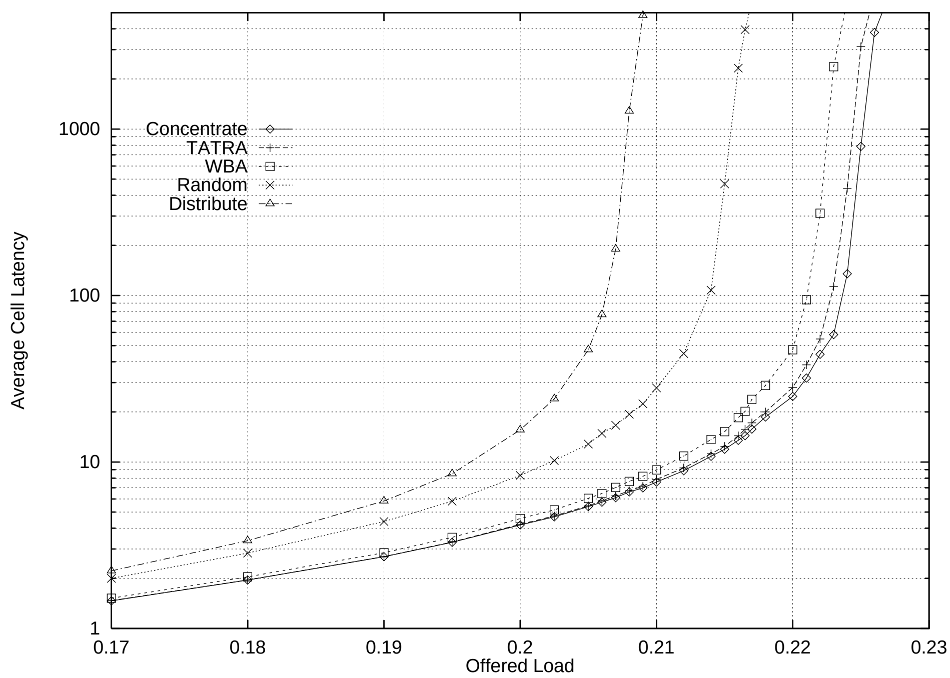

Figure 7: *Graph of average cell latency (in number of cell times) as a function of offered load for an 8×8 switch (Uncorrelated arrivals with an average fanout of four). Note that the total load on the switch is* four *times the offered load at the inputs.*

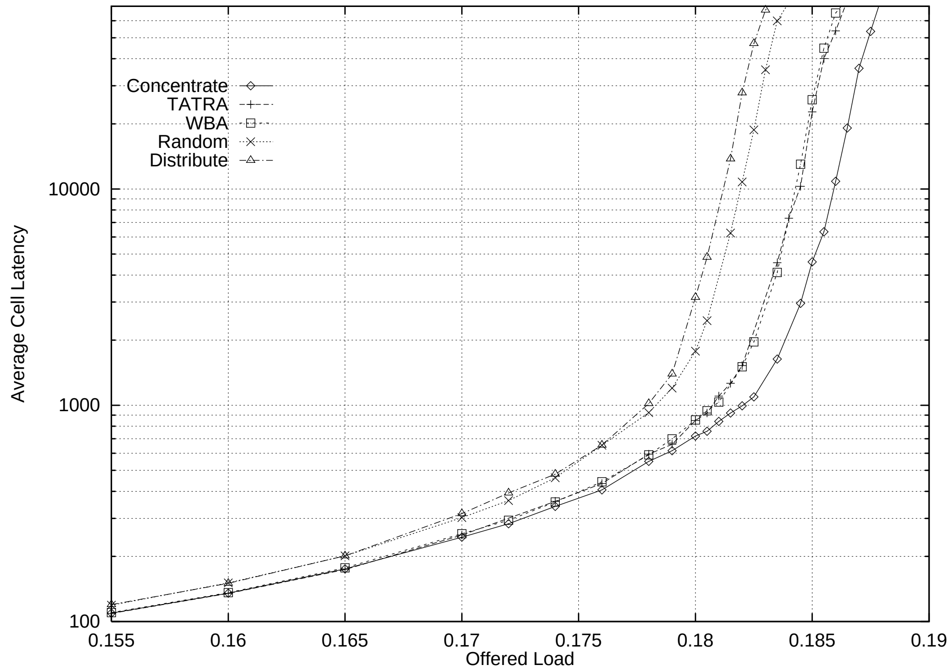

Figure 8: *Graph of average cell latency (in number of cell times) as a function of offered load for an 8×8 switch (Correlated arrivals with an average fanout of four).*

some residue concentration for simplicity. Note that for correlated arrivals, the performance of WBA is almost indistinguishable from TATRA (as seen in Figure 8).

8 Implementation Complexity

Since input-queueing architectures are interesting only at very high bandwidths, it is very important that the scheduling algorithm for an input-queued switch be simple enough to implement in hardware. Here, we compare the implementation complexity of the various scheduling algorithms we have considered.

Concentrate: Even though the *Concentrate* algorithm provides the best throughput performance, it is not a practicable algorithm. First of all, it could lead to the starvation of some inputs; and second, the algorithm requires up to M iterations per cell time to complete. This makes the algorithm difficult to implement at high speed.

TATRA: The TATRA algorithm is simpler to implement than the *Concentrate* algorithm, but still has a time complexity $O(M)$. To understand why this is so, consider a newly arriving HOL input cell. Scheduling the cell is equivalent to determining the position of its peak cell(s), and its non-peak cells. If only one input cell is scheduled per cell time, the scheduling decision can be broken down into two simple stages: (1) The peak cell is scheduled by examining the current profile, and (2) The non-peak cells are scheduled independently by each output. Unfortunately, up to M new input cells may need to be scheduled in a cell time where the positions of their non-peak cells are dependent on the non-peak cells at other outputs. This results in an algorithm of complexity $O(M)$.[5]

[5]However, we have designed an *approximation* to TATRA, in which the input cells can be dropped in

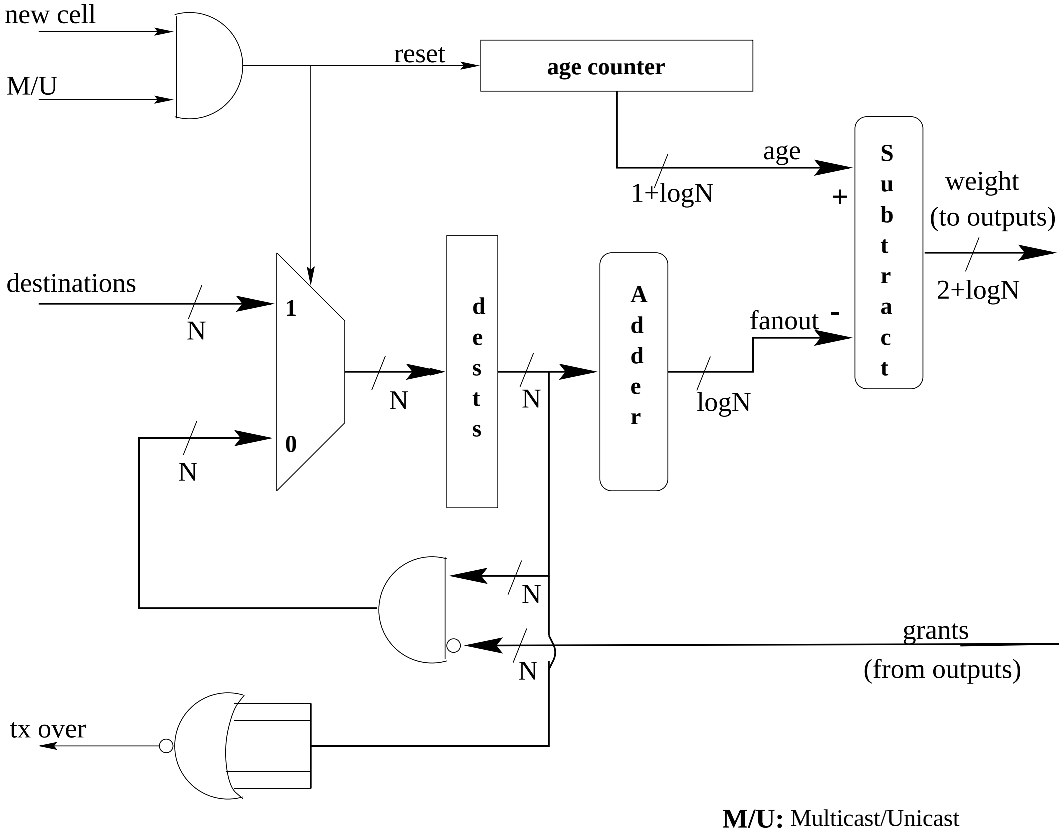

Figure 9: *The hardware required in* WBA *for computation of weight at each input. The* **age counter** *is reset when a new multicast cell comes to* HOL*, and is incremented every cell time thereafter. The bits corresponding to the outputs, which grant to this input, are selectively reset in every cell time until the entire destination vector (the register* **dests***) becomes zero. At this point, the input port is signalled that the transmission of the cell is over, so a new cell can come to* HOL*. Since the maximum age of a cell at* HOL *is* $2N - 1$ *for an* $N \times N$ *switch, the* **age counter** *needs to be* $1 + \log(N)$ *bits wide. Subtracting the fanout (which spans from* 1 *to* N*) from the age makes the total range of weights to be* $3N - 1$*, which can be represented by using* $2 + log(N)$ *bits.*

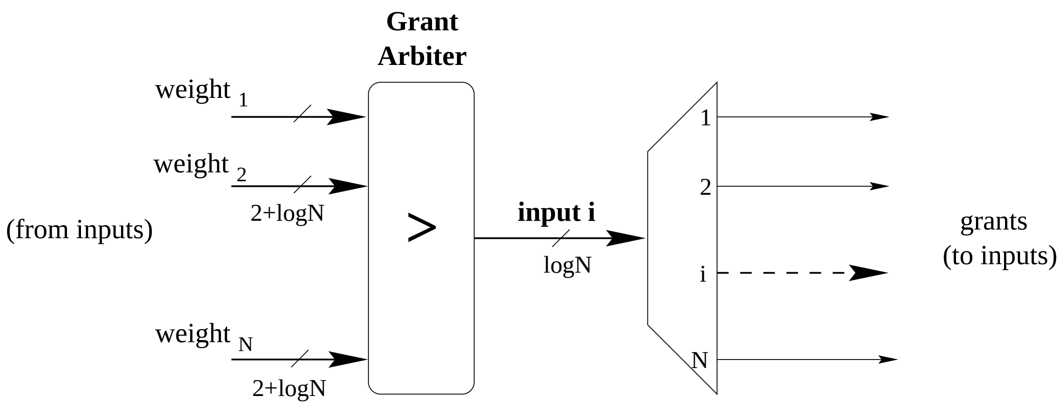

Figure 10: *The hardware required in* WBA *for determining the input to grant to, at each output. The input requesting with the highest weight is selected.*

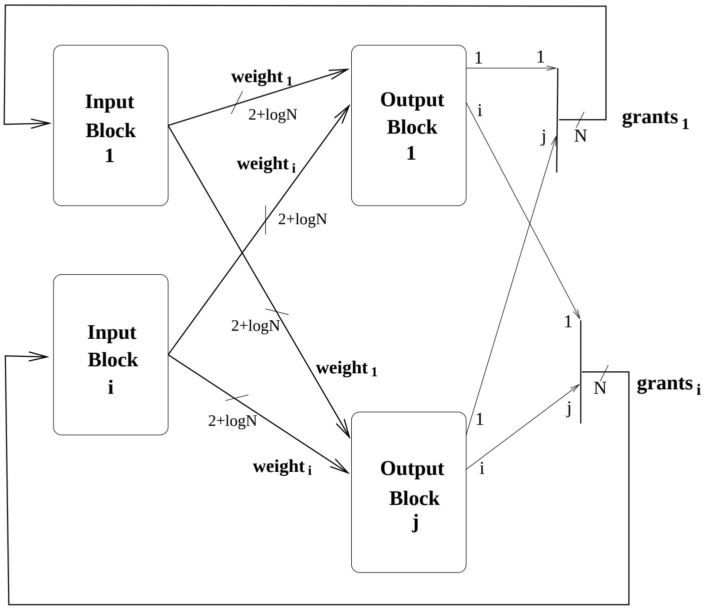

Figure 11: *Connecting N input blocks and N output blocks to form an $N \times N$* WBA *scheduler.*

WBA: This algorithm can be divided into two main parts: (1) Every input computes a request weight, and (2) Every output chooses the input making a request with the highest weight. Since calculating an input's weight does not depend on the weight of any other input, this may be performed in parallel. Similarly, each output may choose the input with the highest weight independently, and may be performed in parallel. Hence the complexity of WBA is $O(1)$. Not only is WBA well suited for parallel implementation, the logic required is relatively simple. To compute its weight, each input subtracts the fanout of the cell at HOL from its age, see Figure 9. Each output employs an M input magnitude comparator to select the input with the highest weight, see Figure 10. A WBA scheduler for an $N \times N$ switch can be constructed by using N input blocks and N output blocks as shown in Figure 11.

References

[1] V. Paxson: "Growth trends in wide-area TCP connections", *IEEE Network,* vol.8, (no.4):8-17. July-Aug 1994.

[2] H. Eriksson: "MBone: the Multicast Backbone", *Communications of the ACM,*vol.37, (no.8):54-60. Aug 1994.

[3] S. E. Deering, and D. R. Cheriton: "Multicast Routing in datagram internetworks and extended LANs", *ACM Transactions on Computer Systems,* vol.8, (no.2):85-110. May 1990.

[4] M. Karol, M. Hluchyj, and S. Morgan: "Input Versus Output Queueing on a Space Division Switch", *IEEE Trans. Comm*, 35(12) pp.1347-1356

[5] S.-Q. Li: "Performance of a nonblocking space-division packet switch with correlated input traffic", *IEEE Trans. Comm*, vol.40, (no.1):97-108. Jan 1992.

[6] T.T. Lee: "Nonblocking copy networks for multicast packet switching", *IEEE J. Select. Areas Comm.*, vol.6, pp.1455-1467. Dec 1988.

[7] J.S. Turner: "Design of a broadcast switching network", *Proc. IEEE INFOCOM '86*, pp.667-675.

parallel, leading to $O(1)$ complexity.

[8] A. Huang: "Starlite: A wideband digital switch," *Proc. IEEE GLOBECOM '84*, pp.121-125.

[9] T. Anderson, S. Owicki, J. Saxe, and C. Thacker: "High Speed Switch Scheduling for Local Area Networks", *Proc. Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* Oct 1992, pp. 98–110.

[10] N. McKeown, P. Varaiya, and J. Walrand: "Scheduling Cells in an Input-Queued Switch", *IEE Electronics Letters,* Dec 9th 1993, pp.2174-5.

[11] N. McKeown: "Scheduling Algorithms for Input-Queued Cell Switches", *PhD Thesis, University of California at Berkeley*, May 1995.

[12] M. Chen, N.D. Georganas: "A Fast Algorithm for multi-channel/port traffic scheduling", *Proc. IEEE Supercomm/ICC '94,* pp.96-100.

[13] H. Obara: "An Efficient Contention Resolution Algorithm for Input Queueing ATM Switches", *Intl. Jour. of Digital & Analog Cabled Systems,* vol. 2, no. 4, Oct-Dec 1989, pp. 261-267.

[14] H. Obara: "Optimum Architecture For Input Queueing ATM Switches", *Elect. Letters*, 28th March 1991, pp.555-557.

[15] N. McKeown, and B. Prabhakar: "Scheduling Multicast Cells in an Input-Queued Switch", *Technical Report: Computer Systems Lab, Stanford University.*

[16] H. Obara, S. Okamoto, and Y. Hamazumi: "Input and Output Queueing ATM Switch Architecture with Spatial and Temporal Slot Reservation Control", *Elect. Letters*, 2nd Jan 1992, pp.22-24.

[17] M. Karol, K. Eng, H. Obara: "Improving the Performance of Input-Queued ATM Packet Switches", *INFOCOM '92*, pp.110-115.

[18] J.F. Hayes, R. Breault, and M. Mehmet-Ali: "Performance Analysis of a Multicast Switch", *IEEE Trans. Commun., vol.39, no.4,* pp. 581-587. April 1991.

[19] K. Eng, M. Hluchyj, and Y. Yeh: "Multicast and Broadcast services in a Knockout packet switch", *INFOCOM '88*, 35(12) pp.29-34.

[20] J. Giacopelli, J. Hickey, W. Marcus, D. Sincoskie, and M. Littlewood: "Sunshine: A high-performance self-routing broadband packet switch architecture", *IEEE J. Selected Areas Commun.,* 9, 8, Oct 1991, pp.1289-1298.

[21] J.Y. Hui, and T. Renner: "Queueing Analysis for Multicast Packet Switching", *IEEE Transactions on Communications, vol.42, no.2/3/4,* pp.723-731, Feb 1994.

[22] M. Mehmet-Ali, and S. Yang: "Performance Analysis of a Random Packet Selection Policy for Multicast Switching", *IEEE Transactions on Communications, vol.44, no.3*, pp.388-398, Mar 1996.

[23] N. McKeown, B. Prabhakar: "Scheduling Multicast Cells in an Input-Queued Switch", *INFOCOM '96*, pp.271-278.

[24] B. Prabhakar, and N. McKeown: "Designing a Multicast Switch Scheduler", *Proc. of the 33rd Annual Allerton Conference*, Urbana-Champaign. 1995.

[25] B. Prabhakar, N. McKeown, and J. Mairesse: "Tetris Models for Multicast Switches", *Proc. of the 30th Annual Conference on Information Sciences and Systems*, Princeton. 1996.

A Proof of Optimality for $2\times N$ Switches

We now present a proof to show that for a $2\times N$ switch a residue concentrating algorithm, subject to a fairness constraint, outperforms all other fair algorithms. We use the following class of inputs for comparing scheduling policies.

Definition 11 (Static Input Assumption): *Following [23], we make the "static input assumption" for switches. That is, it is assumed that at time 0 an infinity of cells has been placed in each input buffer according to some (possibly random) configuration.*

The next two definitions give a fairness constraint for $2\times N$ switches and a criterion used to judge the performance of scheduling policies.

Definition 12 (Fairness Constraint for $2 \times N$ Switches): *A scheduling policy π for a $2 \times N$ switch is said to be* fair *if no cell, from either of the two inputs, is held at* HOL *for more than one cell time.*

Definition 13 (Performance Criterion): *A fair scheduling policy π^1 for a $2 \times N$ multicast switch is said to* perform better *than another fair policy π^2 if every input cell, belonging to either input, departs no later under π^1 than under π^2.*

Under the above conditions a proof of Theorem 1 was given in [23]. For the sake of completeness, a brief sketch of the proof is included here.

A sketch of the proof of Theorem 1: At time 0 we are given an infinity of packets in each input queue, placed according to some configuration. Fix one such configuration and label the cells at inputs 1 and 2 as $\{a_i\}_{i=1,2,\ldots}$ and $\{b_i\}_{i=1,2,\ldots}$ respectively (Figure 12).

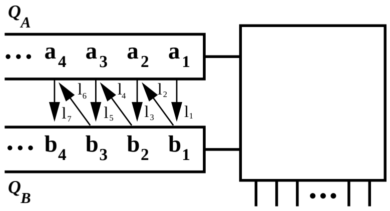

Figure 12: *$2 \times N$ multicast crossbar switch. The links show the order in which cells are released.*

As a consequence of Definition 12, *every* fair scheduling policy discharges the cell (or residue) at the HOL of each input buffer alternately. This orders all input cells according to their departure times as follows: (1) $a_1 \leq_d b_1 \leq_d a_2 \leq_d b_2 \cdots$ if a_1 is the first cell to depart, and (2) $b_1 \leq_d a_1 \leq_d b_2 \leq_d a_2 \cdots$ if b_1 is the first cell to depart. Here $a \leq_d b$ is to be read as "a departs no later than b".

Without loss of generality, we assume the first ordering for cells and *link* them in a vertical or oblique fashion as shown in Figure 12. The directions of the arrows on the links denote where the residue is to be concentrated, should a policy choose to concentrate residue at some time. The vertical link between a_i and b_i is labelled l_{2i-1} and the oblique link between b_i and a_{i+1} is labelled l_{2i}. The following facts now follow easily.

Fact 1 *All scheduling policies work their way through links $l_1, l_2, l_3, \ldots$ in that order. In one cell time, the policies release no links when there is contention between cells at* HOL *and residue is distributed, one link when there is contention between cells at* HOL *and residue is concentrated, or two links when there is no contention between cells at* HOL *in one cell time.*

Fact 2 *The time at which an input cell is completely served is exactly equal to the time at which the link emanating from it is released.*

In light of Fact 2, Theorem **??** is proved if we show that the fair concentrating policy π^* releases each link i no later than any other fair policy π. To this end, consider the plots in Figure 13. Each plot is a "time-link graph" showing the time a policy releases a certain link. Thus, proving Theorem **??** is equivalent to showing that the time-link graph of the residue concentrating policy π^* *lies below* that of any non-concentrating policy. In other words, it is sufficient to prove the following assertion.

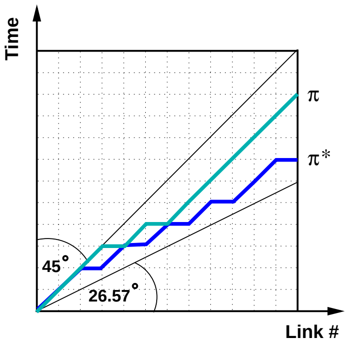

Figure 13: *Time-link graphs of a non-concentrating policy, π, and the concentrating policy, π^*.*

Assertion 1 *The time-link graph of the optimal scheduling policy π^* is never above that of any other scheduling policy.*

A proof of the above assertion (and a complete proof of Theorem **??**) may be found in [23].

Remark: The above proof sample path proof cannot be adapted to prove an analogous result for $M \times N$ ($M > 2$) switches. This is because cells at different inputs cannot be ordered in such a way that all fair, work-conserving policies release them in that specific order. Thus, the simple performance criterion used above cannot be used to compare policies for $M \times N$ switches when $M > 2$. Indeed, counter-examples suggest that by deliberately distributing residue at certain times it is possible for a non-concentrating policy to achieve a higher throughput than a concentrating policy (see [25]).