HEWLETT PACKARD

Multicast Scheduling for Input-queued Switches

Balaji Prabhakar, Nick McKeown[1]; Ritesh Ahuja[2]
Basic Research Institute in the Mathematical Sciences
HP Laboratories Bristol
HPL-BRIMS-96-16
May 1996

multicast; high-speed switch; Tetris models; schedules

The demand for network bandwidth is growing much faster than the increase in commercially available memory bandwidth, causing a growing interest in input-queued switches. Furthermore, an increase in the proportion of multicast traffic in today's networks makes it important that they support such traffic efficiently. This paper presents the design of the scheduler for an M x 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 {\em 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, it is demonstrated that a policy which always concentrates the residue on as few inputs as possible outperforms all other policies. There is a trade-off between concentration of residue (for high throughput), strictness of fairness (to prevent starvation), and implementationl simplicity (for the design of {\em 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] Department of EE & CS, Stanford University
[2] Department of Computer Science, Stanford University

MULTICAST SCHEDULING FOR INPUT-QUEUED SWITCHES

BALAJI PRABHAKAR
BRIMS
Hewlett-Packard Labs, Bristol.
Email: balaji@hplb.hpl.hp.com

NICK MCKEOWN
Departments of EE & CS
Stanford University.
Email: nickm@ee.stanford.edu

RITESH AHUJA
Department of Computer Science
Stanford University.
Email: ritesh@cs.stanford.edu

Abstract

The demand for network bandwidth is growing much faster than the increase in commercially available memory bandwidth, causing a growing interest in input-queued switches. Furthermore, an increase in the proportion of multicast traffic in today's networks makes it important that they support such traffic efficiently. This paper presents the design of the scheduler for an M×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, it is demonstrated that a policy which always concentrates the residue on as few inputs as possible outperforms all other policies. There is a tradeoff between concentration of residue (for high throughput), strictness of fairness (to prevent starvation), and implementionl 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 enourmous growth in the use of Internet and an increasing need for the networking of computers, the demand for network bandwidth has been growing at a phenomenal 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.

The longstanding view has been that input-queued switches are impractical because of poor performance. It is well known that when FIFO queues are used, the throughput of an input-queued switch with unicast traffic can be limited to just 58% under relatively benign conditions due to HOL blocking [4]. When arrivals are correlated, the throughput can be even lower [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×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]. It seems inevitable that the volume of multicast traffic will continue to grow for some time to come. So, if ATM switches are to find widespread use in the Internet, either as standalone switches, or as the core of high performance routers, it is important that they be able to handle multicast traffic efficiently. 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 also important that it be simple to implement in hardware. For example, a switch running at a line rate of 2.4Gbps (OC48) must make 6 million scheduling decisions every second.

In this paper we consider the performance of different multicast scheduling policies. 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). The Concentrate algorithm, we believe, is the multicast scheduling algorithm that maximizes throughput. It achieves this by concentrating the cells that it leaves behind on as few inputs as possible. We show by way of simulation that Concentrate achieves a high thoughput. 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. We introduce two heuristic algorithms that compare favorably with the performance of Concentrate, and yet do not lead to starvation. The first algorithm, 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.

The rest of the paper is organized as follows. In Section 2 we introduce the basic model and define various terms used throughout the paper. After a discussion of the requirements of an algorithm, Section 3 introduces the heuristic of residue concentration. A proof of the optimality of residue concentration for a 2×N switch is presented and the heuristic is further strengthened with the aid of simulation results. In Section 4, we show how the general M×N multicast problem can be described and analyzed with the aid of Tetris models. Within this framework, Section 5 describes an efficient and fair scheduler, TATRA. Some of the attractive salient features of TATRA are explored. A discussion of the tradeoff between fairness, throughput and implementation simplicity then sets the stage for the introduction, in Section 6, of the easily implementable WBA. Section 7 presents simulations comparing the performance of various scheduling algorithms, and finally Section 8 discusses the relative complexity involved in their hardware implementation.

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. The assumption of a single FIFO queue at each input introduces HOL blocking just as in the case of unicast traffic. For an M×N switch, the destination vector of a multicast cell can be any of $2^N - 1$ possible vectors. In order to completely eliminate HOL blocking, maintaining a separate queue for each possible destination vector at each input is therefore impracticable[1]. In light of this, we assume that each input has a single queue and that the scheduler only observes the first cell in the queue.

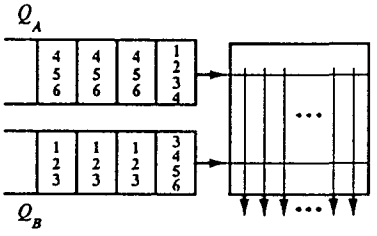

Figure 1: *2×N multicast crossbar switch with single* FIFO *queue at each input.*

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. In the literature the term fanout is often also used to denote the cardinality of this set. We adopt this and, for lack of better terminology, the term fanout will be used throughout this paper to denote both the constitution and the cardinality of the input vector. For example, in the figure, the input cell at the head of each queue is said to have a fanout of four. For clarity, we distinguish an arriving *input* cell from its corresponding *output* cells. In the figure, the

[1]For example, to completely eliminate HOL blocking in a 32 × 32 switch more than 4 billion queues are required at each input.

single input cell at the head of queue Q_A will generate four output cells.

The input queues are necessary because cells at different inputs may wish to be copied to the same output port. At the end of each cell time, a scheduling policy decides which input cells to copy to which output ports. The policy selects a conflict-free match between input and output ports such that each output receives exactly one cell. Thus, at the end of every cell time, the scheduling policy discharges some output cells, possibly leaving behind some residual output cells at the head-of-line (HOL) of the input buffers. For example, in the situation depicted in Figure 1 the *discharge* will consist of output cells for outputs $\{1, 2, 3, 4, 5, 6\}$, and the *residue* will consist of output cells for outputs $\{3, 4\}$. Observe that the decision of which inputs to serve is equivalent to the decision of which inputs *not* to serve, making scheduling equivalent to deciding on residue placement. One can therefore define a scheduling policy in terms of how it elects to place the residue on different inputs. A scheduling policy may elect to place the residue on both inputs (i.e., it "distributes the residue"), or it may place the residue on either Q_A only or on Q_B only (i.e., it "concentrates the residue"). It is the purpose of this paper to argue, based on theoretical results and simulations, that a scheduling policy which always "concentrates the residue" performs better (improves output utilization, reduces input queue latency, etc.) than one that does not always concentrate residue.

To reduce implementation complexity, we assume that an input cell must wait in line until all of the cells ahead of it have gained access to all of the outputs that they have requested. Perhaps the simplest 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 service discipline does not take advantage of the multicast properties of the 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 4. It is clear from the figure that the Random policy with no fanout-splitting fares much worse than the same policy with 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.

[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.

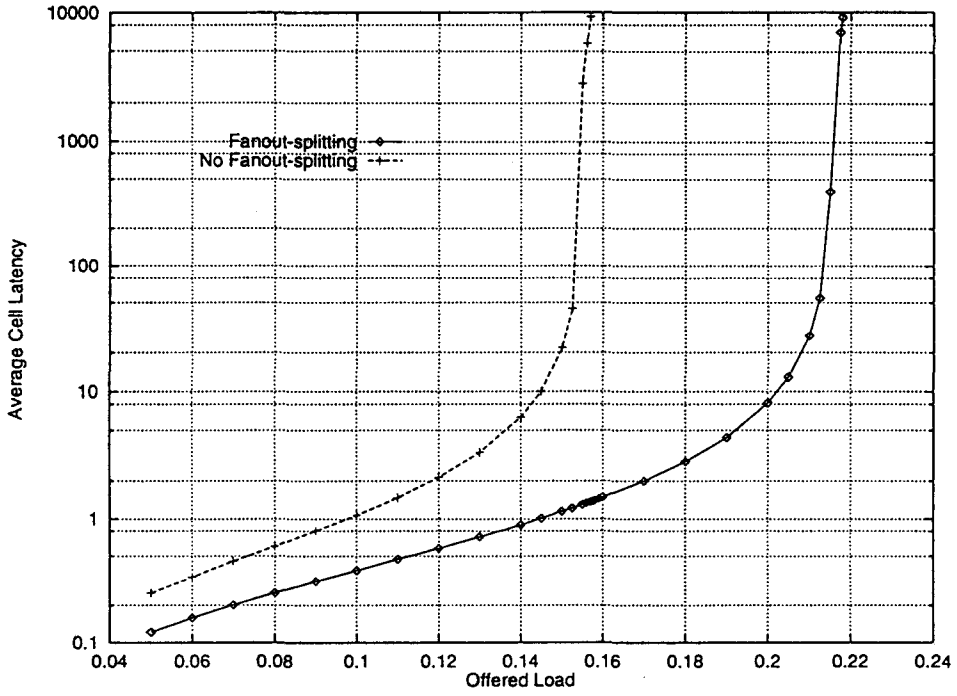

Figure 2: *Graph of average cell latency as a function of offered load for an 8×8 switch (with uniform input traffic and average fanout of 4). The graph compares Random scheduling policy with and without fanout-splitting.*

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. The* residue at an input port *is the set of output cells that originated from this input port and have lost contention for output ports.*

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). The fairness constraint ensures that inputs do not starve, and occassionally we refer to it as a "starvation constraint".*

For an M×N switch, it is not possible for the above bound to be less than M (simply consider the case where all new cells at HOL contend for the same set of outputs). However, it is possible that some input cells may be "opportunistically served" in less than M cell times because they do not contend with other input cells for outputs. In the case of a 2×N switch, if we require that each input cell be held at HOL for no more than two cell times, then this definition of fairness means that the residue alternates between the two inputs.

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. Following this, we present a proof of the optimality of residue-concentration for 2×N switches.

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 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×N switch, the previous argument can be rigorously proved.

3.1 Proof of Optimality for 2×N Switches

We now present a proof to show that for a 2×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 6 (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×N switches and a criterion used to judge the performance of two scheduling policies.

Definition 7 (Fairness Constraint for 2×N Switches): *A scheduling policy π for a 2×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 8 (Performance Criterion): *A fair scheduling policy π^1 for a 2×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.*

Given the conditions stated above, the following result concerning the optimality of the fair residue concentrating policy for 2×N switches was proved in [23]. For the sake of completeness, a brief sketch of the proof is included here.

Theorem 1 *A scheduling policy for a 2×N multicast switch that always concentrates residue at every possible instant subject to the fairness condition of Definition 7, performs better than any other fair policy when subjected to static inputs.*

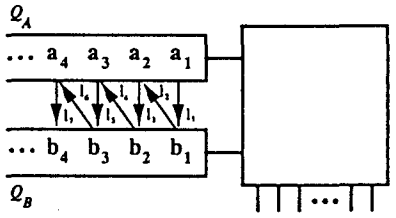

Figure 3: *2×N multicast crossbar switch. The links show the order in which cells are released.*

Sketch of Proof: 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 $\{a_i\}_{i=1,2,...}$ and $\{b_i\}_{i=1,2,...}$ respectively (Figure 3).

As a consequence of Definition 7, *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 3. 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.

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.*

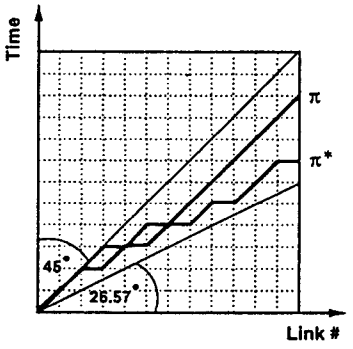

Figure 4: *Time-link graphs of a non-concentrating policy, π, and the concentrating policy, π^*.*

In light of Fact 2, Theorem 1 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 4. Each plot is a "time-link graph" showing the time a policy releases a certain link. Note that in Figure 4 the diagonal line (of slope 1) is the time-link graph of the worst policy - that is, this policy releases precisely one link per unit time. Similarly, the line of slope 1/2 is the time-link graph of the best policy - one that always releases 2 links per unit time (this is only possible if there is no contention at all). Clearly, the time-link graphs of all policies lie between these two extremes.

Thus, proving Theorem 1 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.

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 1) may be found in [23].

4 Tetris Models for M×N Switches

This section presents a unified approach to the design and analysis of schedulers for an M×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].

Before intuitively motivating the connection between Tetris models and multicast scheduling, we first describe the class of scheduling policies to be considered in this section. The policies will be required to satisfy the following fairness constraint.

Definition 9 (Fairness Constraint for M×N Switches): *A scheduling policy π for a M×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. Once assigned, the DD of a cell cannot be increased, and 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 and there is no look-ahead.

4.1 Tetris models: a sketch

We map the operation of an M×N multicast switch onto a Tetris-like game in the following fashion. It is imagined that every input cell is constituted of a number of identical copies

of itself, each of these copies is thought of as an output cell. This allows us to map input cells onto Tetris blocks, each block will then be an amalgamation of smaller blocks, one per output cell. Upon assignment of DDs, the input cells at HOL will be dropped into a compartmentalized box of size M×N - each compartment or slot capable of holding one output cell. 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. Refer to the example of a 5×5 switch shown in Figure 5. The labels on the cells denote the input port from which the cell has arrived; all output cells with same label have resulted from the same input cell. The cells in the bottom-most row of the box in Figure 5 at columns 1, 3 and 5 are all identical copies of a cell from input 1 destined to outputs 1, 3 and 5 (note that this input cell has a fanout of 3). Similarly, the cell at the HOL of input 2 wishes to access outputs 2, 3, 4 and 5.

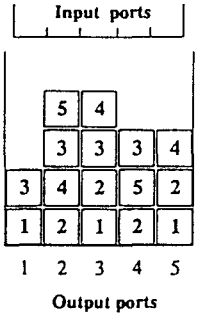

Figure 5: *An example. Cells from inputs 1, 2, 3, 4 are assigned DDs 1, 2, 3, 4 respectively, while the cell from input 5 is assigned a DD of 4.*

Suppose that, at time n, the switch is to schedule k input cells which have advanced to HOL; i.e., there have been precisely k departures in the previous cell time. 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. Again, referring to Figure 5, note that there are no unoccupied slots between cells in any output column. The reason for this will become clear momentarily.

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 5, 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; i.e., these gaps may lead to an idling of the output in the future.

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

[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.

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 whereas Tetris blocks are rigid and cannot be decomposed, work-conservation will at times require that the various output cells constituting an input cell depart at different cell times. 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. As mentioned earlier, if a plurality of cells advance to HOL at the beginning of a cell time, it is important to determine the order in which they are assigned DDs. In general, it is better to allow this ordering to depend on the constitution of the current residue and of the new cells. However, for simplicity, we choose 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 10 (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 11 (Occupancy Set): *The* occupancy set *of the cell or residue at the* HOL *of input* l *at time* n *is given by* $O_l(n) = \{T_{i,j} : \text{ an output cell of } l \text{ resides at } T_{i,j} \text{ at time } n.\}$

Definition 12 (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 13 (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 *and* j, *if* $T_{i,j}$ *and* $T_{i+2,j}$ *are occupied, then so is* $T_{i+1,j}$; *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 $\pi(n+1)$.

4.3 An Example

Consider the example of Figure 5 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

At the end of cell time 1, input 1 is completely served and advances a new cell to HOL. Suppose that this new cell wishes to access outputs 1 and 5. Figure 6 shows two different ways of scheduling the new cell.

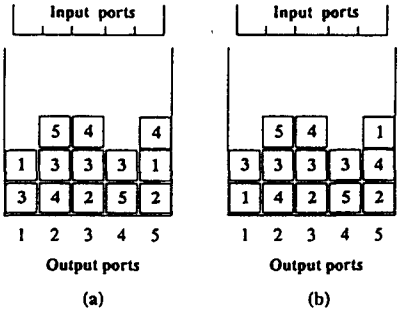

Figure 6: *Two ways of scheduling a new cell at input 1.*

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 5, it is fairly easy to see that TATRA schedules the cells as shown in Figure 7a below. 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 7b shows how TATRA schedules them. Observe that in Figure 7b the cell from input 3 at position $T_{1,1}$ is a pseudo-peak cell because this cell has a DD equal to 2, which is the same as the DD of the cell from input 1 at position $T_{1,2}$. Therefore, the height of the peak column corresponding to output 1 in Figure 7b is equal to 2.

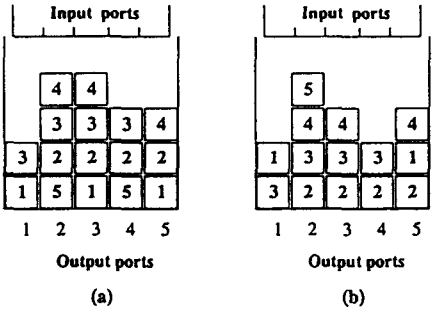

Figure 7: *TATRA schedules:* (a) *the cells of Figure 5,* (b) *the new cells from inputs 1 and 5 at time 2.*

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 there is guaranteed to be a discharge 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. To see this, note that for the example of Figure 6a, cells from inputs 4 and 5 are criss-crossing each other. At the end of the current cell time, one output cell from each of inputs 4 and 5 is discharged leaving a residue on both. This distribution of residue can be prevented by avoiding criss-cross; i.e., by simply swapping the positions of cells from inputs 4 and 5 in output column 2.

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 by distributing it over different inputs. 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.

Throughput vs Fairness: a discussion

An algorithm that maximizes residue concentration, without regard to fairness, may achieve a high throughput but it can lead to the starvation of some inputs. For example, in the Concentrate algorithm, an input requesting all the outputs may wait for ever without being served. Conversely, if an algorithm aims to be fair, it may not achieve the best possible residue concentration and therefore sacrifices throughput. For example, TATRA has to sacrifice some throughput in order to meet its very strict fairness constraint. The more an algorithm tries to concentrate the residue, the less strict it becomes in terms of fairness.

WBA: The Weight Based Algorithm

The above discussion of the tradeoff between throughput and fairness suggests that, in order to choose the desired operating point between the two extremes of residue concentration and strict fairness, an algorithm needs to decide on their relative importance. This leads us to a simple weight based algorithm, WBA.

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, in the interests of achieving fairness, a positive weight should be given to age. 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. In other words, by discharging more inputs one automatically ensures that the residue is concentrated on few inputs. Since the bottom-most row can only take N output cells, to accomodate the maximum number of *input cells* in this row one must choose those input cells which have the least fanout. That is, *the lower the fanout, the heavier the input cell.*

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, allowing one to use other features to assign weights to 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 subscribing 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.

7 Simulation Results

7.1 Traffic Types

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 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 each period for a geometrically

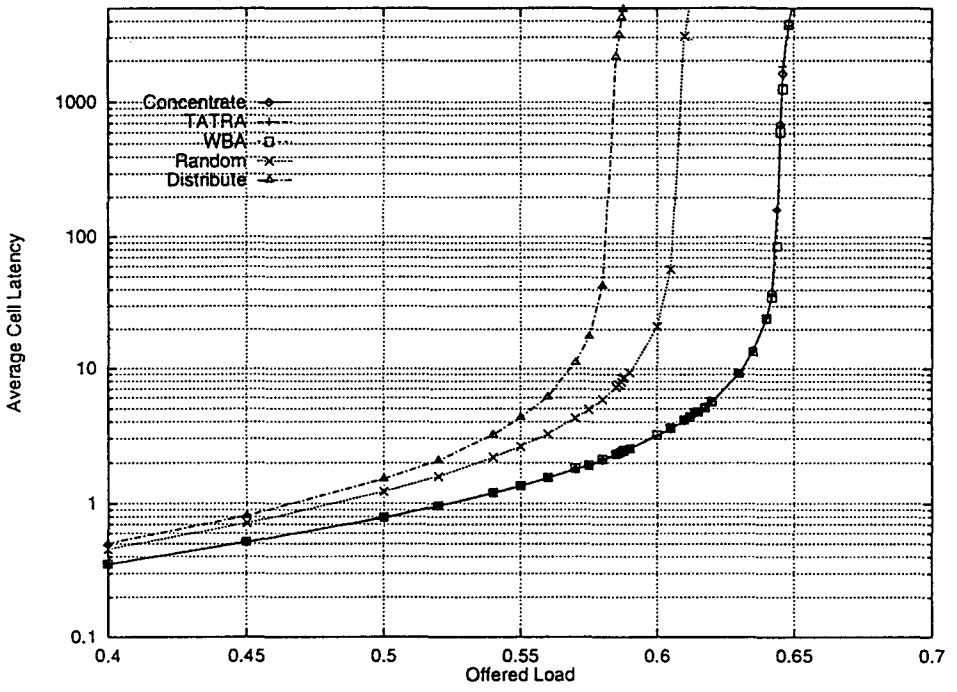

Figure 8: *Graph of average cell latency as a function of offered load for a 2×8 switch (Uncorrelated arrivals with an average fanout of 4).*

distributed number of cell times. The expected duration of the BUSY state is fixed at 16 cells[4]. When in this state, cells arrive at the beginning of every cell time and all with the same set of destinations. No cells arrive during the IDLE state.

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×N switch, the average fanout is slightly larger than N/2.

For comparison, we also plot the performance of the algorithm *Random*, in which each output randomly selects one input from among those requesting it. This algorithm is motivated by the work of Hayes et al. in [18], which is the multicast version of the unicast algorithm described in [4]. The WBA plots are obtained by using twice the negative weight for fanout as for the age of the cell at HOL.

7.2 2×8 Switch

Figures 8 and 9 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 much lower than for the *Distribute* algorithm. Note that TATRA performs identically to *Concentrate*, as expected for a 2×8 switch. This is because, for a 2×N switch, the residue of each input is the same and in every cell time the scheduling algorithm decides which input to place this residue on. Placing the residue on the younger cell (one which has been at HOL for the shortest period of time), as is done by

[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.

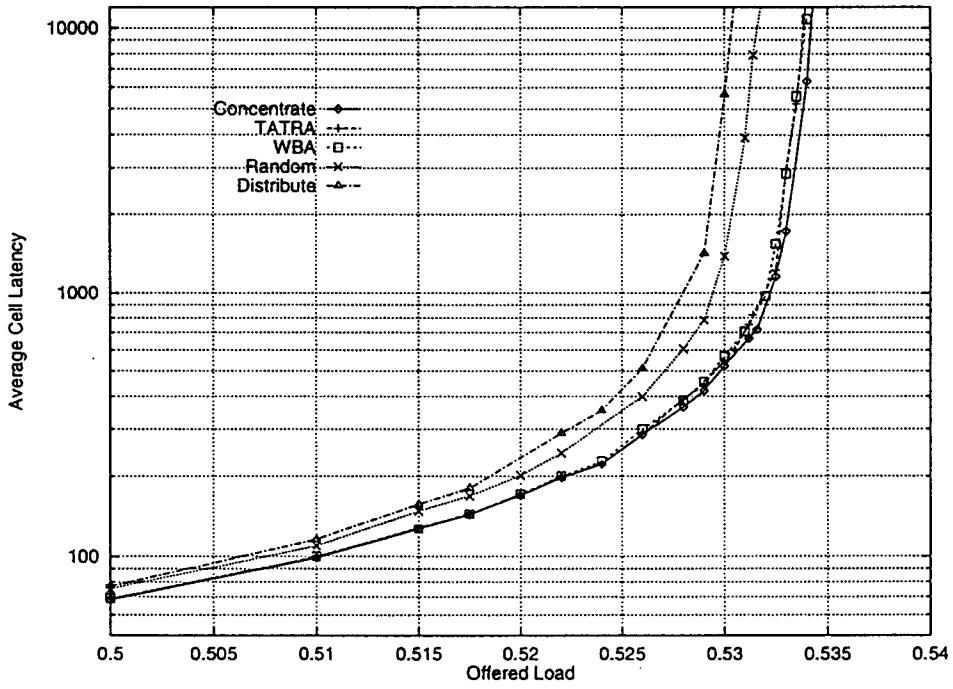

Figure 9: *Graph of average cell latency as a function of offered load for a 2×8 switch (Correlated arrivals with an average fanout of 4).*

Concentrate, is same as allowing the older cell to go first, as is done by TATRA. Note also from the figures that WBA also performs very well, whereas *Random* performs much worse.

7.3 8×8 Switch

Figures 10 and 11 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 lower than that of other algorithms, supporting our argument that the *Concentrate* algorithm outperforms other algorithms.

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, without any communication with other outputs. So, when two or more inputs have the same weight, different outputs may grant requests to different inputs, causing a distribution of residue. Therefore, WBA is not as effective in concentrating the residue as TATRA. If all the outputs were able to communicate with each other, and make an collective decision, WBA would perform better than TATRA because it has a less restrictive starvation constraint. However, that would increase the complexity of the algorithm making it impracticable to implement in hardware. Thus WBA sacrifices some residue concentration for simplicity. Note that for correlated arrivals, the performance of WBA is almost indistinguishable from TATRA (see Figure 11).

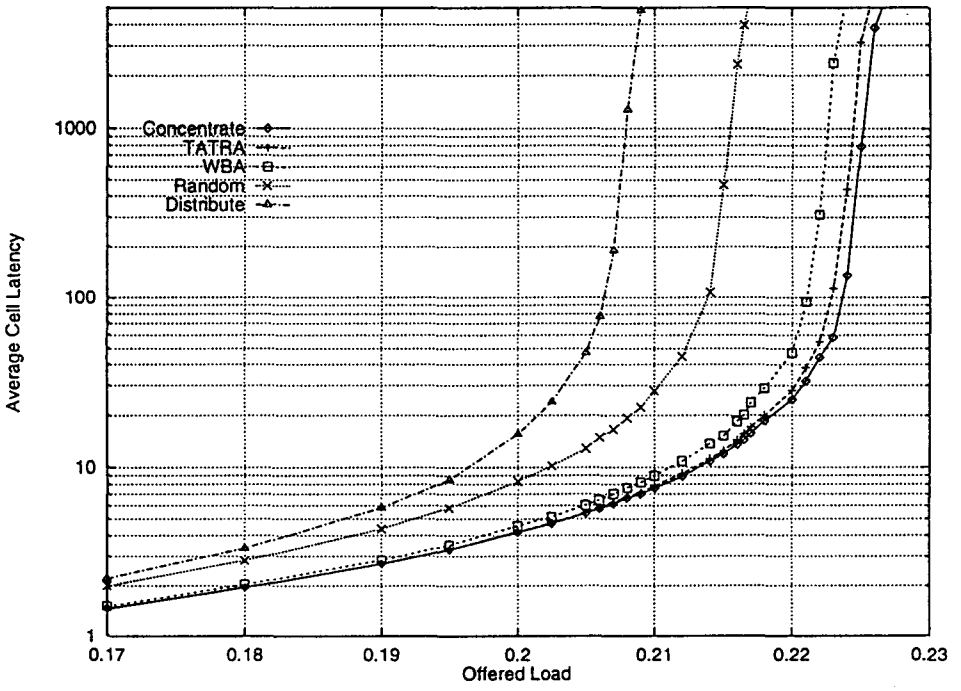

Figure 10: *Graph of average cell latency as a function of offered load for an 8×8 switch (Uncorrelated arrivals with an average fanout of 4).*

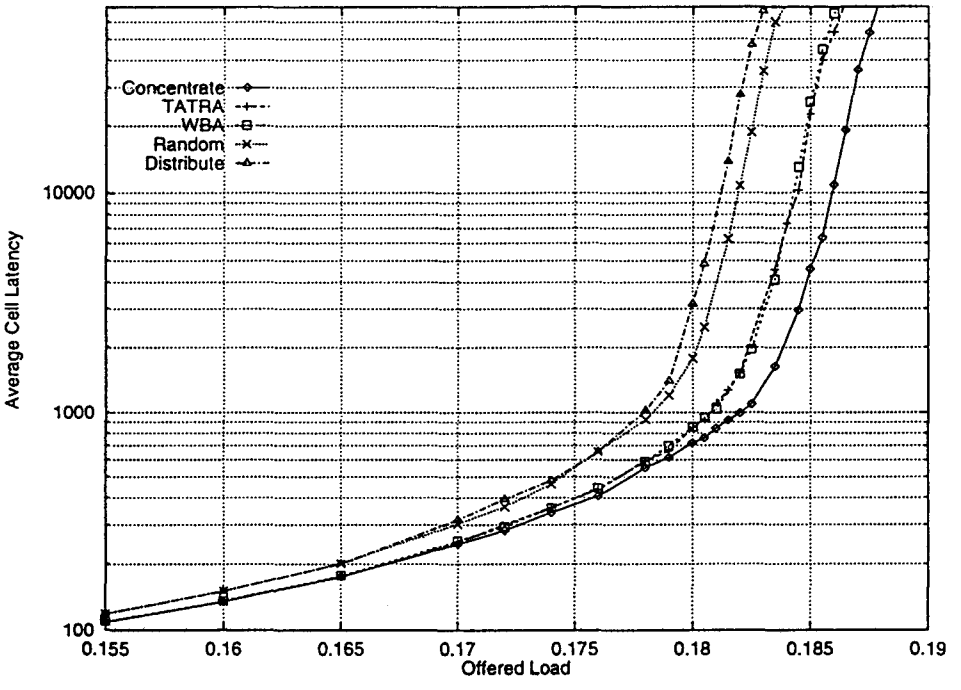

Figure 11: *Graph of average cell latency as a function of offered load for an 8×8 switch (Correlated arrivals with an average fanout of 4).*

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.

8.1 Concentrate

Even though the *Concentrate* algorithm provides the best throughput performance, it is not a practicable algorithm. First of all, it violates our starvation contraint; and in every cell time, the algorithm requires M iterations in the worst case, leaving a part of the residue on one cell in each iteration. Because of its $O(M)$ complexity this algorithm is very difficult to implement at high speeds.

8.2 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 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 stages: (1) peak cells are scheduled by examining the current profile (that is, their DDs are determined), and (2) non-peak cells can then be scheduled independently by each output. Unfortunately, up to M new input cells may need to be scheduled in a cell time. 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].

8.3 WBA

This algorithm can be divided into two main parts: (1) Every input computes its weight, with which it will request each of the outputs in its destination vector, and (2) Every output chooses the input making request with the highest weight. Since the weight computation of an input does not depend on the weight of any other input, this part can be done separately at each input in parallel. Similarly, each output just chooses the input with the highest weight *independently*, and this part of the algorithm can be performed in parallel at each output. Hence the complexity of WBA is $O(1)$. Not only is WBA well suited for parallel implementation, the logic required at each input (for part 1), and each output (for part 2) is relatively simple. An input just needs to subtract the fanout of the cell at HOL from its age, in order to compute its weight (Figure 12), and an output just needs a magnitude comparator (an M input magnitude comparator) to find the input with the highest weight (Figure 13). A WBA scheduler for an N×N switch can be constructed by using N input blocks and N output blocks as shown in Figure 14.

[5]However, we have an *approximation* to TATRA, in which the input cells can be dropped in parallel, leading to $O(1)$ complexity.

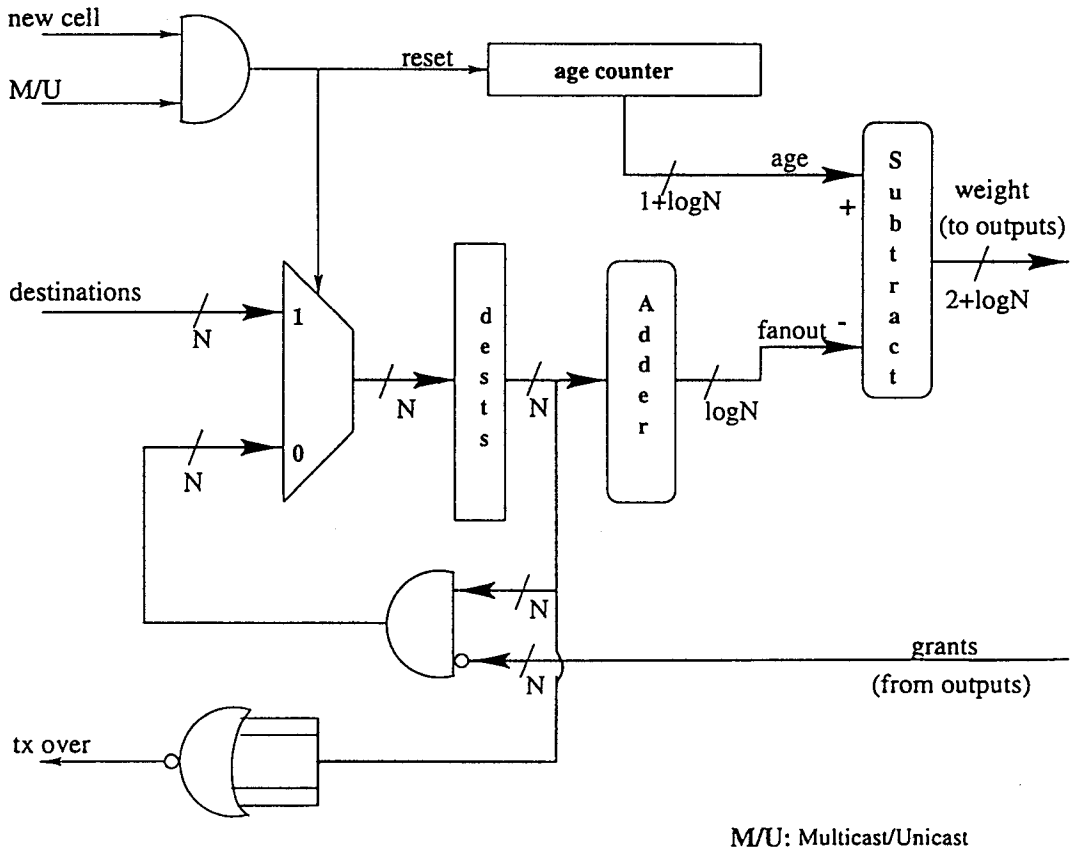

Figure 12: *The hardware required in* WBA *for computation of weight at each input. The age counter is reset when a new multicast cell comes to* HOL*, and is incremented every cell time thereafter. The bits corresponding to the outputs, which grant to this input, are selectively reset in every cell time until the entire destination vector 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.

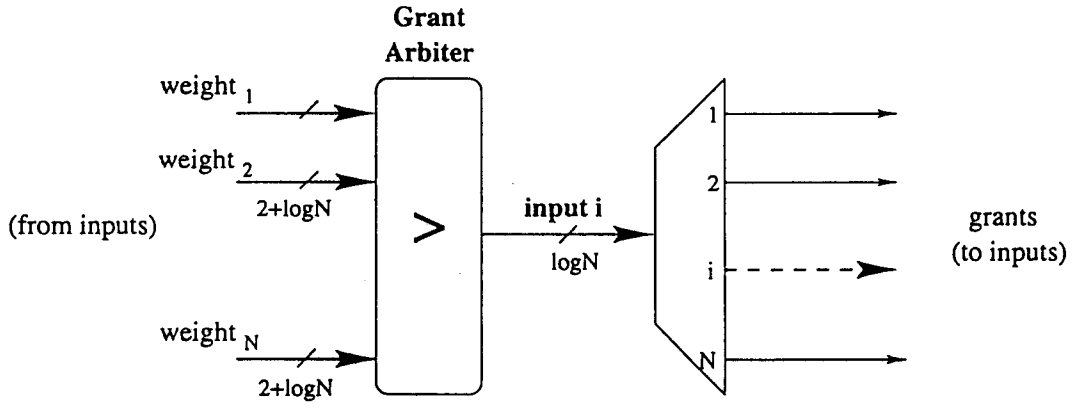

Figure 13: *The hardware required in* WBA *for determining the input to grant to, at each output. The input requesting with the highest weight is selected.*

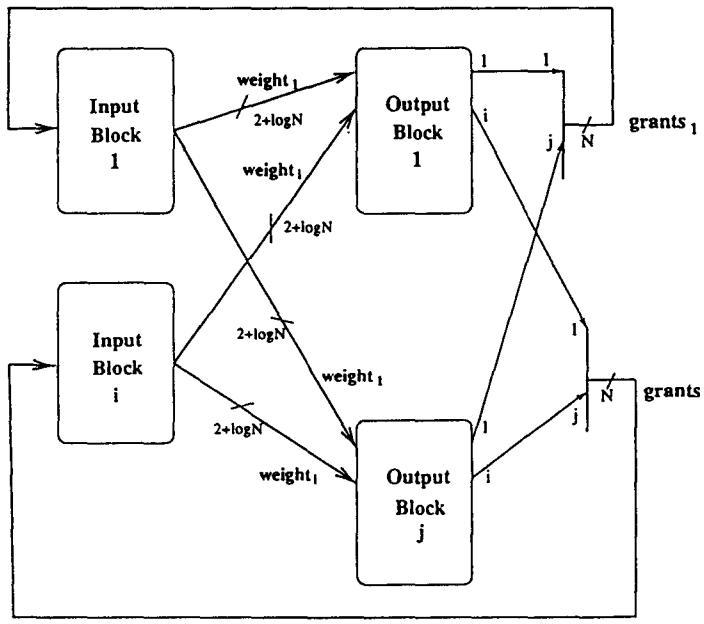

Figure 14: *Connecting N input blocks and N output blocks to form an N×N* WBA *scheduler.*

9 Conclusion

The increase in demand for network bandwidth creates a need for high-speed input-queued multicast switches. To help the designer of such switches, we have studied multicast scheduling policies. Our work leads to an understanding of the maximum achievable throughput for multicast switches, and introduces high-performance algorithms that are simple to implement in hardware.

In particular, we have observed that when designing a multicast scheduler, it is important to determine the placement of residue. This lead to the development of the following "residue-concentration heuristic": To achieve a high throughput, a scheduler should always concentrate the residue onto as few inputs as possible. The heuristic was supported by simulations and, for 2×N switches, a proof established the optimality of the residue-concentrating policy. However, concentrating residue at all times can be unfair and lead to the starvation of some inputs. We therefore concluded that designing a simple, fast and efficient multicast scheduler is an exercise in balancing the conflicting requirements of throughput maximization, ensuring fairness, and implementational simplicity.

We then imposed a fairness constraint on our scheduling policies and used Tetris models to study the general M×N multicast scheduling problem. This led to the development of a fair and efficient scheduling policy – TATRA, and some salient features of TATRA were explored. Though fair and efficient (in terms of high throughput and low latency), TATRA was found to be implementationally complex. To remedy this, we developed and studied a weight based algorithm called WBA which is easily implemented in hardware, ensures fairness and achieves good throughput.

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, 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.

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

[22] Mustafa K. Mehmet Ali, Shaying 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] Nick McKeown, Balaji Prabhakar "Scheduling Multicast Cells in an Input-Queued Switch", *INFOCOM '96*, pp.271-278.

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

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