A Quantitative Comparison of Iterative Scheduling Algorithms for Input-Queued Switches*

Nick McKeown Thomas E. Anderson

Abstract

In this paper we quantitatively evaluate three iterative algorithms for scheduling cells in a high-bandwidth input-queued ATM switch. In particular, we compare the performance of an algorithm described previously — parallel iterative matching (PIM) — with two new algorithms: *iterative round-robin matching with slip* (*i*SLIP) and *iterative least-recently used* (*i*LRU). We also compare each algorithm against FIFO input-queueing and perfect output-queueing. For the synthetic workloads we consider, including uniform and bursty traffic, *i*SLIP performs almost identically to the other algorithms. Cases for which PIM and *i*SLIP perform poorly are presented, indicating that care should be taken when using these algorithms. But, we show that the implementation complexity of *i*SLIP is an order of magnitude less than for PIM, making it feasible to implement a 32×32 switch scheduler for *i*SLIP on a single chip.

1 Introduction

The past few years has seen increasing interest in arbitrary topology cell-based local area networks, such as ATM [5]. In these networks, hosts are connected together by an arbitrary graph of communication links and switches, instead of via a shared medium, as in Ethernet[24], or a ring, as in FDDI[4]. One reason for the popularity of arbitrary topology networks is that they offer a number of potential advantages relative to other approaches[2]: (i) aggregate throughput that can be much larger than that of a single link, (ii) the ability to add throughput incrementally as the workload changes by simply adding extra links and switches, (iii) improved fault tolerance

*Nick McKeown was supported by Pacific Bell, Bellcore and California State MICRO Program Grant 93-152. He is with the Department of Electrical Engineering, Stanford University, CA 94305. By email nickm@ee.stanford.edu. Tom Anderson is supported by the Advanced Research Projects Agency (N00600-93-C-2481), the National Science Foundation (CDA-8722788), California MICRO, Digital Equipment Corporation, the AT&T Foundation, Xerox Corporation, and Siemens Corporation. He was also supported by a National Science Foundation Young Investigator Award and a Sloan Research Fellowhsip. Anderson is with the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, Berkeley, CA 94720.

by allowing redundant paths between hosts, (iv) and reduced latency because control over the entire network is not needed to insert data.

But to realize the potential advantages of arbitrary topology networks, a high performance switch is needed to take data arriving on an input link and quickly deliver it to the appropriate output link. (In this paper, we will only consider switches that deal in fixed-length packets, or cells.) A switch requires three functions: a switch fabric, scheduling to arbitrate when cells arrive on different inputs destined for the same output, and (optionally) queueing at inputs or outputs to hold those cells that lose the arbitration. A large number of alternative switch designs have been proposed with varying approaches to these three functions [2, 7, 9, 13, 16, 25, 26], each with their own set of advantages and disadvantages.

The goal of this paper is to make a quantitative comparison of scheduling policies for input-queued switches, considering both hardware complexity and switch performance. We observe that while it can be tempting to throw hardware at improving switch performance, that strategy may be counter-productive in practice, since simpler solutions can be easier to implement at high speed.

Input queueing is attractive for very high bandwidth ATM switches — this is because no portion of the internal datapath needs to run faster than the line rate. The longstanding view has been that input-queued switches are impractical because of poor performance; if FIFO queues are used to queue cells at each input, only the first cell in each queue is eligible to be forwarded. As a result, FIFO input queues suffer from *head of line (HOL) blocking* [15] – if the cell at the front of the queue is blocked, other cells in the queue cannot be forwarded to other unused inputs. HOL blocking can limit the throughput to just 58%. Because HOL blocking has very poor performance in the worst case [19], the standard approach has been to abandon input queueing and instead use output queueing – by increasing the bandwidth of the internal interconnect, multiple cells can be forwarded at the same time to the same output, and queued there for transmission on the output link.

Our work is motivated by a different approach. HOL blocking can also be eliminated by maintaing at each input, a separate queue for each output [1, 2, 15, 27], where any cell queued at an input is eligible for forwarding across the switch. This technique is sometimes called *virtual output queueing*. Although this method is more complex than FIFO queueing, note that each input queue does not require increased bandwidth; each input must receive and transmit at most one cell per time slot. The central problem with random-access input queues is the need for fast *scheduling* – quickly finding a conflict-free set of cells to be forwarded across the switch, such that each input is connected to at most one output, and vice versa. As observed in [2], this is equivalent to *matching*[1] inputs to outputs [28].

In this paper we evaluate fast, parallel, iterative algorithms for scheduling cells in an input-queued switch. In particular, we compare DEC's parallel iterative matching (PIM) algorithm with two novel algorithms that we have devised: *i*SLIP and *i*LRU.

We simulate the behavior of these algorithms under a variety of synthetic workloads, comparing them on queueing delay, and fairness. We also compare the hardware complexity of two of the

[1]Or more precisely, bipartite graph matching.

more promising approaches.

Our study has two main results. First, we describe a novel switch scheduling policy, called *i*SLIP, which performs as well as the best previously proposed algorithm. We find that there are some special workloads for which *i*SLIP will perform better and some for which it will perform worse. *i*SLIP, however, has the advantage of being much simpler to implement. We demonstrate that an *i*SLIP scheduler for a moderate-scale switch ($\leq$ 32 inputs/outputs) can fit onto a single chip; this appears not to be the case with other approaches. Single chip implementations will be crucial if we are to be able to keep up with increasing link speeds – a 32 way ATM switch with 10 Gbit/sec links would need to make more than 25 million scheduling decisions per second.

Second, we demonstrate that seemingly trivial changes in switch scheduling policy can have a large impact on performance. *i*SLIP and *i*LRU are very similar, except for the ordering of the entries in the schedules. This small difference can lead to very different performance characteristics.

The remainder of this paper discusses these issues in more detail. Section 2 describes the various switch scheduling policies which we consider. Section 3 outlines our methodology for comparing the policies. Section 4 considers hardware implementation complexity; Section 5 presents performance results and Section 6 summarizes our comparative study.

2 Alternative Algorithms

In this section, we describe the scheduling algorithms that we examine in this paper, outlining the key design issues for each algorithm. We defer until afterwards the quantitative comparison of performance.

We assume that the switch synchronously forwards cells from all inputs at the same time. In an input-queued switch without any internal speedup, an input can only send one cell into the switch fabric in a cell time, or slot. Likewise, each output can only receive one cell in a slot. A conflict-free match connects inputs to outputs of the switch without violating these two requirements. Each algorithm has the same goal: to find a conflict-free match of inputs to outputs in a single cell time.

2.1 FIFO Scheduling

The simplest approach to managing queues and scheduling cells is to maintain a single FIFO queue at each input; the queues hold cells which have not yet been transmitted through the switch. To schedule the crossbar for the next cell time, the scheduling algorithm examines the cells at the head of each FIFO queue. The problem is that two or more FIFO queues may have cells destined for the same output. In this case, the scheduling algorithm must select just one cell from among the contending inputs. This is necessary to meet our requirement that each output can only accept one cell per slot for an input-queued switch with no internal speed-up.

On its own, the above description isn't enough to specify how FIFO works. If there are two

inputs both with cells for the same output, which input will be selected to send its cell first? One possibility would be to maintain a fixed priority and, for example, always select the lowest numbered input. This would be fast and simple to implement, but would be unfair among input-output pairs: in fact, it can be so unfair that one or more flows[1] may be permanently starved of service.

Alternatively, the scheduling algorithm could *randomly* select among the contending inputs. This leads to a fair allocation among flows, but is hard to implement at high speed: the algorithm must perform a random number lookup uniformly over a varying set of inputs. Yet we could meet the same objective — starvation avoidance — with a rotating priority. This method has the advantage of not requiring a random number lookup, but we must vary the priority carefully to maintain fairness and avoid starvation. This choice between random and time-varying priorities is found in all of the approaches we consider.

Before considering other scheduling approaches, we need to examine a serious performance problem caused by maintaining FIFO queues at the inputs: *head of line (HOL) blocking* [15]. The problem is that when a cell is blocked at the head of an input queue (due to contention with a cell from a different input destined for the same output) it also blocks all the cells behind it in the queue *even though those cells may be destined for an output that is currently idle.* HOL blocking has been shown to limit the aggregate throughput of the switch to just $2-\sqrt{(2)} \approx 58\%$ of its maximum for independent Bernoulli arrivals and destinations uniformly distributed over outputs [15]. With less benign traffic patterns, the situation can be much worse [19].

Several methods have been proposed for reducing HOL blocking [12, 15, 16, 25, 27], but a simple way described in [15, 16] to eliminate HOL blocking is to maintain a separate FIFO queue at each input for each output.[1] This is illustrated in Figure 1. If cells for one output are blocked because the output is busy, the scheduling algorithm may select a cell for some other output. This method was implemented in [2], where they showed that the scheme is not only straightforward to build, but yields large performance gains. Note that despite the increased complexity at the input buffer, the memory bandwidth need not increase: each input still receives and transmits at most one cell per time slot.

Now that we have arranged the buffering to eliminate HOL blocking, we must change the flow scheduling algorithm: the algorithm must now consider the cells at the head of all N^2 input FIFOs and determine a pattern of conflict-free flows that will provide a high throughput. (The algorithm is an example of bipartite graph matching [28] and is also an example of the "rooks on a chessboard problem" [1]).

Recently it has been shown that 100% throughput can be achieved in an input-queued switch if an algorithm considers the occupancy of each of the N^2 input queues [22]. However, as the authors point out, it is infeasible for this algorithm to perform quickly in hardware. The algorithms that

[1]Throughout this paper we will refer to the stream of cells flowing between an input and output as a "flow". A flow in general contains multiple ATM virtual circuits.

[1]In practice, we would maintain a separate FIFO for each flow (in ATM terms, each virtual circuit). When a particular input-output flow is selected, we would then choose among those flows in a random, pre-determined or time-varying manner. But as we are only considering the scheduling of input-output flows in this paper, we shall assume that each input maintains only N output FIFOs.

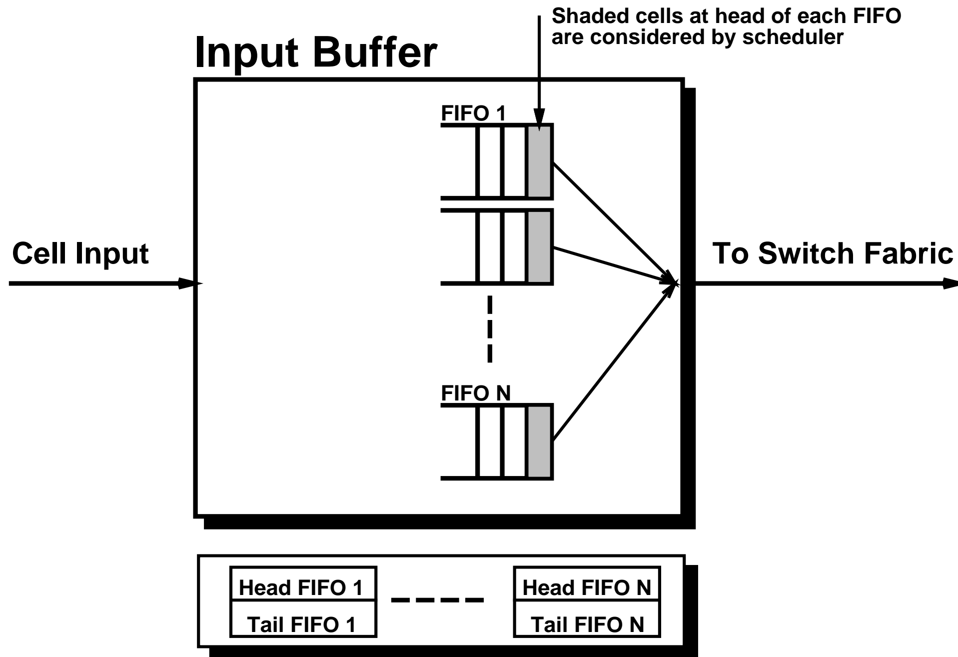

Figure 1: A single input queue containing N separate FIFOs managed within a single buffer memory.

we describe here only consider whether an input queue is empty or non-empty — they do not consider the occupancy of the queue. This simplifies the algorithms considerably, making them feasible to implement in hardware.

2.2 Maximum Size Matching

A desirable algorithm would be one that finds the maximum number of matches between inputs and outputs. This would provide the highest possible instantaneous throughput in a given time slot.

There are several algorithms for achieving a maximum size match [8, 11, 17, 28], but these are not suitable for this application for two reasons. First, a maximum size match can take a long time to converge and, second, it can lead to starvation of an input-output flow under certain traffic patterns. Referring to Figure 2 we see that even a simple traffic pattern can lead to starvation for a maximum size matching algorithm. Input 2 will never be able to send cells to output 1 as this would lead to a match of only one flow.

We have a dilemma: the maximum number of cells per time slot will be delivered using a maximum size match; but the maximum size match can take too long to compute and will potentially starve some flows.

Each of the algorithms that follow attempt to achieve a maximal, not a maximum size match, by iteratively adding flows to fill in the missing flows left by the previous iteration. Because the flows made in previous iterations may not be removed, this technique does not always lead to a

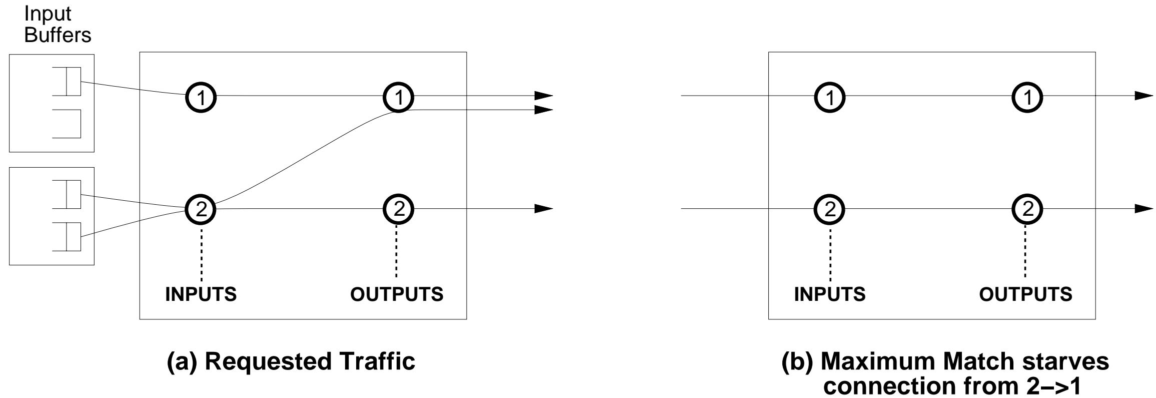

Figure 2: Starvation for a 2x2 switch using a Maximum size Matching scheduling algorithm.

maximum size match. However, we shall see that it is possible to achieve a close approximation to the maximum for many traffic patterns. Furthermore, by introducing randomness or time-varying priorities, we shall see that starvation can be avoided.

2.3 Parallel Iterative Matching (PIM)

The first iterative algorithm that we shall describe is called parallel iterative matching [2]. It uses *randomness* to (i) reduce the number of iterations needed, and (ii) avoid starvation.

PIM attempts to quickly converge on a conflict-free match in multiple iterations where each iteration consists of three steps. All inputs and outputs are initially unmatched and only those inputs and outputs not matched at the end of one iteration are eligible for matching in the next. The three steps of each iteration operate in parallel on each output and input and are as follows:

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

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

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

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

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

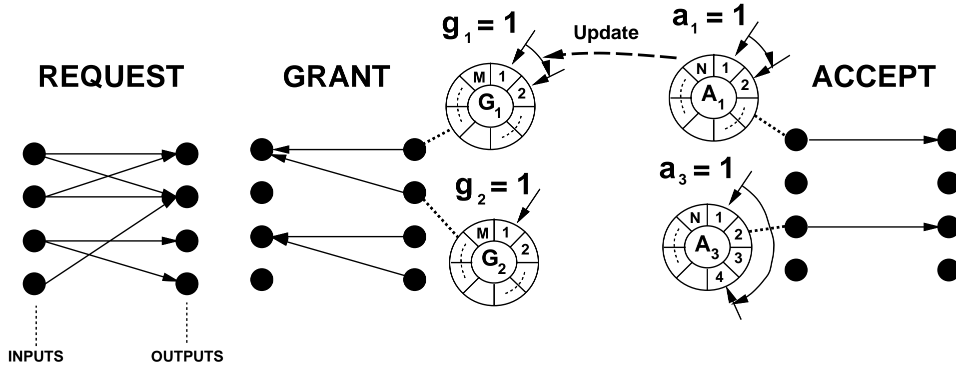

Figure 3: One iteration of the three phase *i*SLIP scheduling algorithm.

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

But using randomness comes with its own problems. First, it is difficult and expensive to implement at high speed (we shall consider this more later) and second, it can lead to unfairness between flows. This will be discussed more when we present simulation results in Section 5.

The two algorithms that we shall consider next (*i*SLIP and *i*LRU), avoid the use of randomness when making selections. Instead they use a *varying priority* based on state information that is maintained from one time slot to the next.

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

*i*SLIP, like PIM, is an iterative algorithm and was first described in [21]. The following three steps are iterated for an $N \times N$ switch:

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

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

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

Time Slot 1				
	a_i	g_i	Output line that receives cell.	In iteration
Line 1	1	1	1	1
Line 2	1	1	2	2
Line 3	1	1	3	3
Line 4	1	1	4	4
Time Slot 2				
	a_i	g_i	Output line that receives cell.	In iteration
Line 1	2	2	2	1
Line 2	1	1	1	1
Line 3	1	1	3	2
Line 4	1	1	4	3
Time Slot 3				
	a_i	g_i	Output line that receives cell.	In iteration
Line 1	3	3	3	1
Line 2	2	2	2	1
Line 3	1	1	1	1
Line 4	1	1	4	2
Time Slot 4				
	a_i	g_i	Output line that receives cell.	In iteration
Line 1	4	4	4	1
Line 2	3	3	3	1
Line 3	2	2	2	1
Line 4	1	1	1	1

Table 1: An example of four successive time slots for a 4×4 switch, with continuously occupied queues.

An example of one iteration of the three phases is illustrated in Figure 3. In the example, input 1 has one or more cells for outputs {1,2}, input 2 has one or more cells for outputs {1,2} and so on. The grant arbiters are shown for outputs 1 and 2, and accept arbiters are shown for inputs 1 and 3. At the end of the time slot, g_1 and a_1 are incremented to 2 and a_3 is incremented to 4; input 2 will match output 2 in the next iteration, but this will have no effect on the priorities.[1]

An example of how cells are served in successive cell times is shown in Table 1. This shows a 4×4 switch with every input queue always full of cells for every output. The table illustrates the evolution over four cell times.

[1] The reason for only updating the pointer on the first iteration is to avoid starvation. If the pointer can move past an unsuccessful first choice, the first choice may never be served. An alternative method, albeit slightly more complex, is to maintain a separate pointer for each iteration.

So, each output arbiter decides among the set of ordered, competing requests using a rotating priority. When a requesting input is granted and the input accepts that grant, the input will have the lowest priority at that output in the next cell time. Also, the input with the highest priority at an output will continue to be granted during each successive time slot until it is serviced. This ensures that a flow will not be starved: the highest priority flow at an output will always be accepted by an input in no more than N cell times.

We highlight four main properties of the *i*SLIP scheduling algorithms:

Property 1. Flows matched in the first iteration become the lowest priority in the next cell time.

Property 2. No flow is starved. Because of the requirement that pointers are not updated after the first iteration, an output will continue to grant to the highest priority requesting input until it is successful.

Property 3. The algorithm will converge in at most N iterations. Each iteration will schedule zero, one or more flows. If zero flows are scheduled in an iteration then the algorithm has converged: no more flows can be added with more iterations. Therefore, the slowest convergence will occur if exactly one flow is scheduled in each iteration. At most N flows can be scheduled (one to every input and one to every output) which means the algorithm will converge in at most N iterations.

Property 4. The algorithm will not necessarily converge to a maximum sized match. At best, it will find a maximal match: the largest size match without removing flows made in earlier iterations.

Moving the pointers not only prevents starvation, it tends to *desynchronize* the grant arbiters. This desynchronization of the grant arbiters is the reason that *i*SLIP performs so well, and is able to converge quickly upon a maximal match.[1] As we shall see later, the *i*SLIP algorithm leads to surprisingly large matches particularly under high offered load, and even for a single iteration.

The high performance of iSLIP is a consequence of Step 2 — an output arbiter may only be updated if a successful match is made in Step 3.

We now explain why this simple rule leads to high performance. Consider a switch that is heavily loaded with traffic. Intuitively, if each output arbiter favors a different input, then every output is likely to grant to a different input and a large match will be achieved in a single iteration. So how does *i*SLIP encourage the (independent) output arbiters to favor a different input? To see how this happens, notice that each input connects to at most one output arbiter and each output arbiter connects to at most one input arbiter. When a successful matched output arbiter moves its pointer to one position beyond the input that it is connected to, it must be the *only* output that moves its pointer to that position.[2] In other words, the set of output pointers that

[1] Refer to [21] for a more detailed description.

[2] The output arbiter may of course clash with another output arbiter that already points to the same input, but was not matched in the previous cell time. However, this indicates that there is no backlog on this flow and can be ignored.

Input Choice	Output Choice		
	Random	Rotating	LRU
Random	PIM [2]		
Rotating		*i*SLIP	
LRU			*i*LRU

Table 2: Alternative Selection methods for Input and Output arbiters.

move, do not clash with each other in the next cell time. The heavier the load, the more flows are made and so the smaller the number of clashes in the next cell time.

We say that two clashing arbiters are *synchronized* with respect to each other, and we refer to their tendency to move apart as *desynchronization*.

Because of the round-robin movement of the pointers, we can also expect the algorithm to provide a fair allocation of bandwidth among competing flows. the algorithm will visit each competing flow in turn, so that even if a burst of cells for the same output arrives at the input, the burst will be spread out in time if there is competing traffic. We will examine fairness further when we present our simulation results in Section 5.

2.5 Iterative Least Recently Used (*i*LRU)

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

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

Thus, for each of the iterative algorithms that we have described, there are several alternatives for choosing among contending inputs and outputs. Table 2 shows the alternative ways of making selections at the input (accept) and output (grant) arbiters; we only examine the symmetric form of *i*SLIP and *i*LRU in which the same scheduling scheme is used at the input and output, but the other combinations are possible.

3 Methodology

To evaluate the performance of the iterative PIM, *i*SLIP and *i*LRU scheduling algorithms, we have compared them to each other and to FIFO queueing, perfect output queueing and maximum size matching. FIFO queueing is used as a lower bound: it is the simplest to implement, but suffers from HOL blocking. On the other extreme, perfect output queueing only has queueing caused by contention for the output links. This provides an upper bound on the performance that any switch can attain. Maximum size matching indicates the best throughput that could be achieved in a given time slot for an input queued switch.

For each of the iterative algorithms we need to decide how many iterations to perform within a single cell time. For extremely high bandwidth switches, limited only by the link technology, there may only be time to perform a single iteration of the scheduling algorithm. For this reason, we compare the algorithms for a single iteration.

In other applications, there may be time for more than one iteration, particularly if the scheduler can be integrated on a single chip. In [2], the authors were able to achieve four iterations of the PIM algorithm within a single 53-byte ATM cell time (420ns) at link speeds of 1Gbits/sec. For comparison, we also consider the PIM, *i*SLIP and *i*LRU algorithms for four iterations.

3.1 Bases for Comparison

There are a number of factors that would lead a switch designer to select one scheduling algorithm over another. First, it must be simple to implement. A more complex implementation is not only more expensive in area and power, but is likely to provide less performance—particularly if each iteration requires off-chip communication. Second, the algorithm should provide high throughput and avoid starvation for any flow pattern. This is because real network traffic is rarely uniformly distributed over inputs and outputs. Fourth, the algorithm should provide high throughput for bursty traffic. Real network traffic is highly correlated from cell to cell [18] and the scheduling algorithm should not be unduly affected by the burstiness. However, we can expect all algorithms to perform worse under more bursty traffic. This is not necessarily because of the algorithm, but rather because of the temporary increase in queue length that occurs when the arrival rate momentarily exceeds the service rate.

In the next two sections, we will first consider the implementation complexity of the algorithms before considering their relative performance. Using simulation, we examine the performance for uniform and asymmetric work loads as well as for non-bursty and bursty traffic.

4 Complexity Comparison

For PIM, *i*SLIP and *i*LRU, the scheduler chip needs to know the status of all the input queues (empty/non-empty). This takes N^2 bits of state within the scheduler. To maintain this state for unicast traffic, each input must indicate $1 + \log_2 N$ bits to the scheduler per cell time. $\log_2 N$ bits are required to identify the input; the additional bit indicates if the value is valid. When

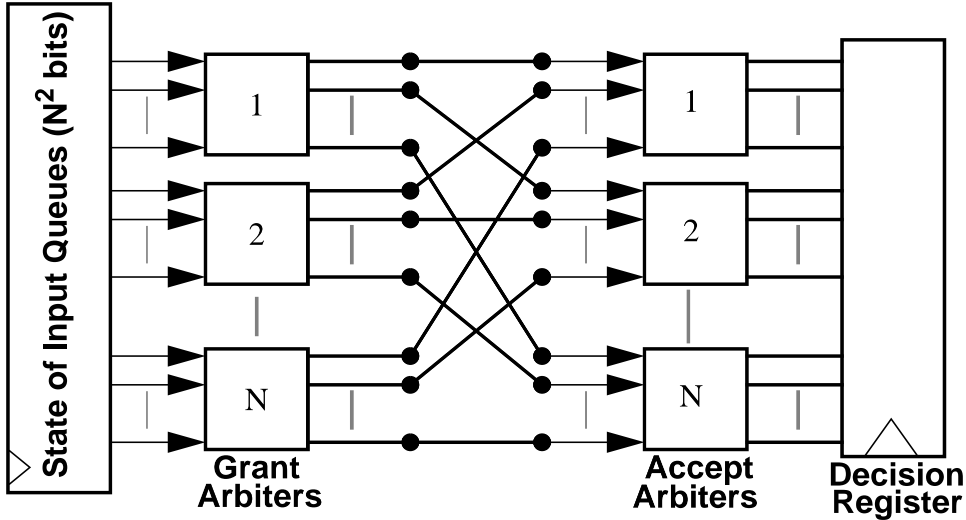

Figure 4: An *i*SLIP scheduler consists of $2N$ identical arbiters.

the scheduler completes its matching decision, it informs each input which output, if any, it can send a cell to.

A PIM scheduler is difficult to implement—random selection is an expensive operation, particularly for a variable number of input requests. In the implementation described in [2], the scheduler filled approximately 1.5 Xilinx 3190 [31] devices for each port [29]. For a 16×16 switch, this is approximately 64,000 *complex* gates. If we assume that each gate is approximately eight 2-input gate equivalents this totals 512,000 simple gates.

An *i*SLIP scheduling chip contains $2N$ arbiters: N grant arbiters to arbitrate on behalf of the outputs, and N accept arbiters to arbitrate on behalf of the inputs. Each arbiter is an N input priority encoder with a programmable priority level. A schematic of the complete scheduler is shown in Figure 4; the details of one arbiter is shown in Figure 5. Using the "misII" tools from the Berkeley Octtools VLSI design package[6], we have determined that the scheduler for a 16×16 switch will require approximately 35,000 2-input gate equivalents. This is an order of magnitude less complex than for PIM for the same sized switch. Further, since a custom VLSI chip has room for at least 200,000 gates, this indicates that a single chip version of *i*SLIP would be feasible— but not a single chip version of PIM. We are currently completing the implementation of a 32×32 *i*SLIP scheduler as part of the Tiny Tera project at Stanford [23].

Although we did not design a full implementation for *i*LRU, it is much more complex than *i*SLIP. Like *i*SLIP, the implementation requires $2N$ identical arbiters. Each arbiter must keep an ordered list indicating how recently an input or output was selected. This requires keeping an ordered set of N registers each of width $\log_2 N$. Furthermore, N comparators are required to determine which is the highest priority that was selected. This hardware is significantly more complex than for PIM or *i*SLIP, making it impractical to place on a single chip.

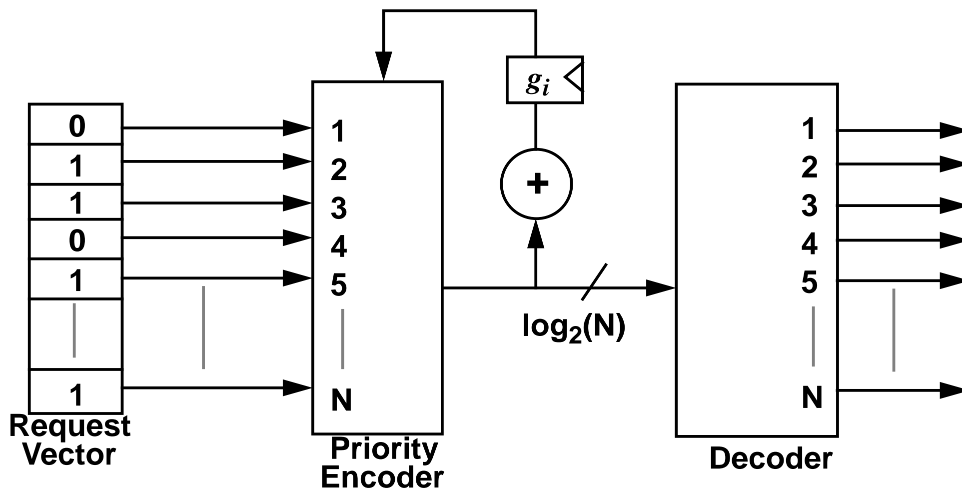

Figure 5: An *i*SLIP arbiter is a programmable priority encoder. The priority level P indicates the lowest priority. (P+1) mod N is the highest priority.

5 Performance Results

In this section, we compare the performance results by simulation for PIM, *i*SLIP and *i*LRU against FIFO, maximum size matching and output queueing for a 16×16 switch. In each case, simulation results are reported for a given traffic pattern after the switch has reached steady-state. The switch is assumed to have infinite sized buffers. We start in Section 5.1 by considering the performance of one and four iterations of each algorithm with a Bernoulli arrival process (i.e. in each cell time, a random choice is made for each input whether a cell is to arrive in that time slot) and destinations uniformly distributed over outputs. This is followed by a discussion of some bad performance cases for PIM and *i*SLIP under asymmetric work loads. Finally, we consider performance for bursty traffic.

5.1 Performance Under Uniform Traffic

5.1.1 A Single Iteration

We consider first the case where cells arrive as a Bernoulli process with cell destinations distributed uniformly over all outputs. Figure 6 shows the average queueing delay (measured in cells) versus the average offered load for PIM, *i*SLIP and *i*LRU with one iteration contrasted with FIFO, maximum matching and perfect output queueing.

Because PIM, *i*SLIP and *i*LRU all avoid HOL blocking we expect them to achieve a higher maximum offered load than FIFO queueing. This is the case: PIM, *i*SLIP and *i*LRU achieve maximum offered load of 63%, 100% and 64% respectively, whereas FIFO queueing achieves slightly more than 58%.[1] PIM is limited to less than 63% because this is the probability that

[1]FIFO is limited to 58% for a switch with an infinite number of inputs and outputs. For finite $N = 16$ the maximum offered load is slightly higher than 58%.

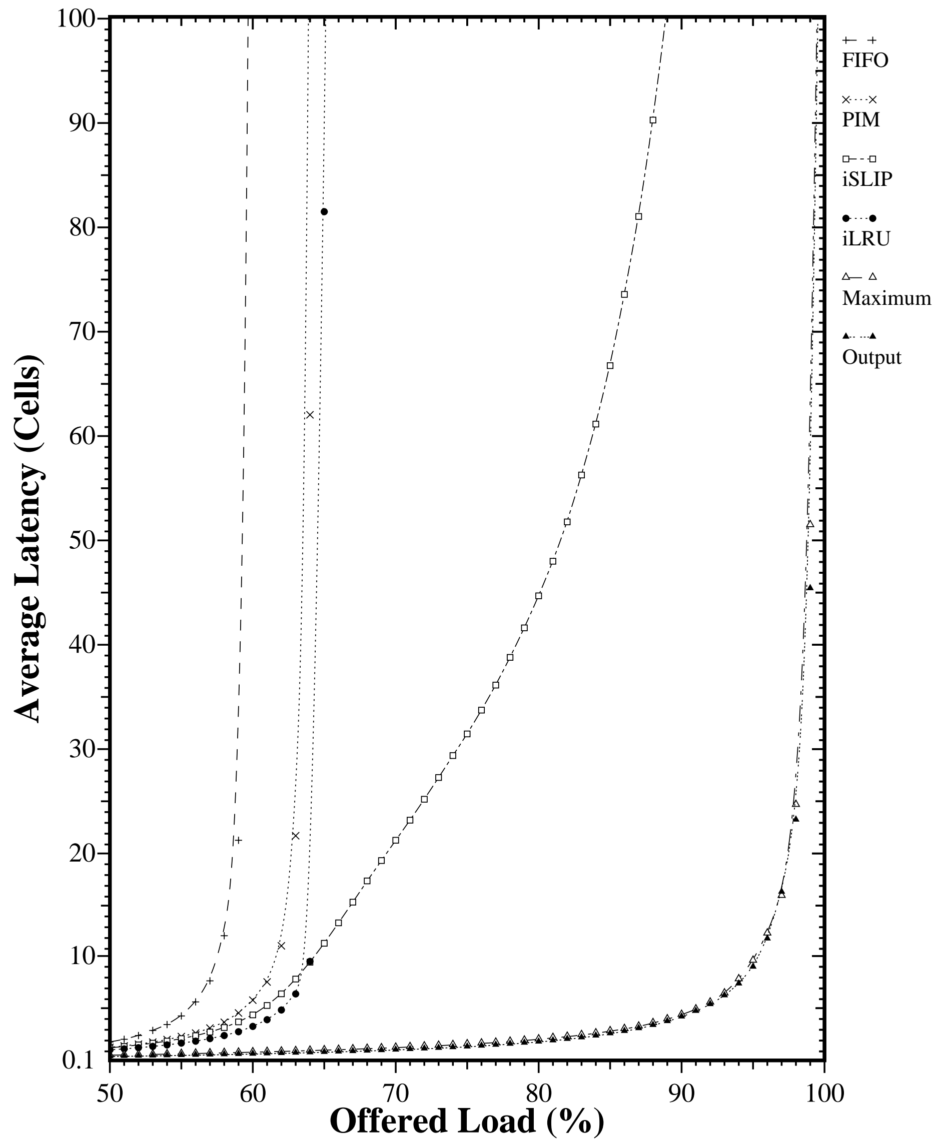

Figure 6: Delay vs. offered load for PIM, iSLIP, iLRU, FIFO, for a single iteration with uniform workload and Bernoulli arrivals.

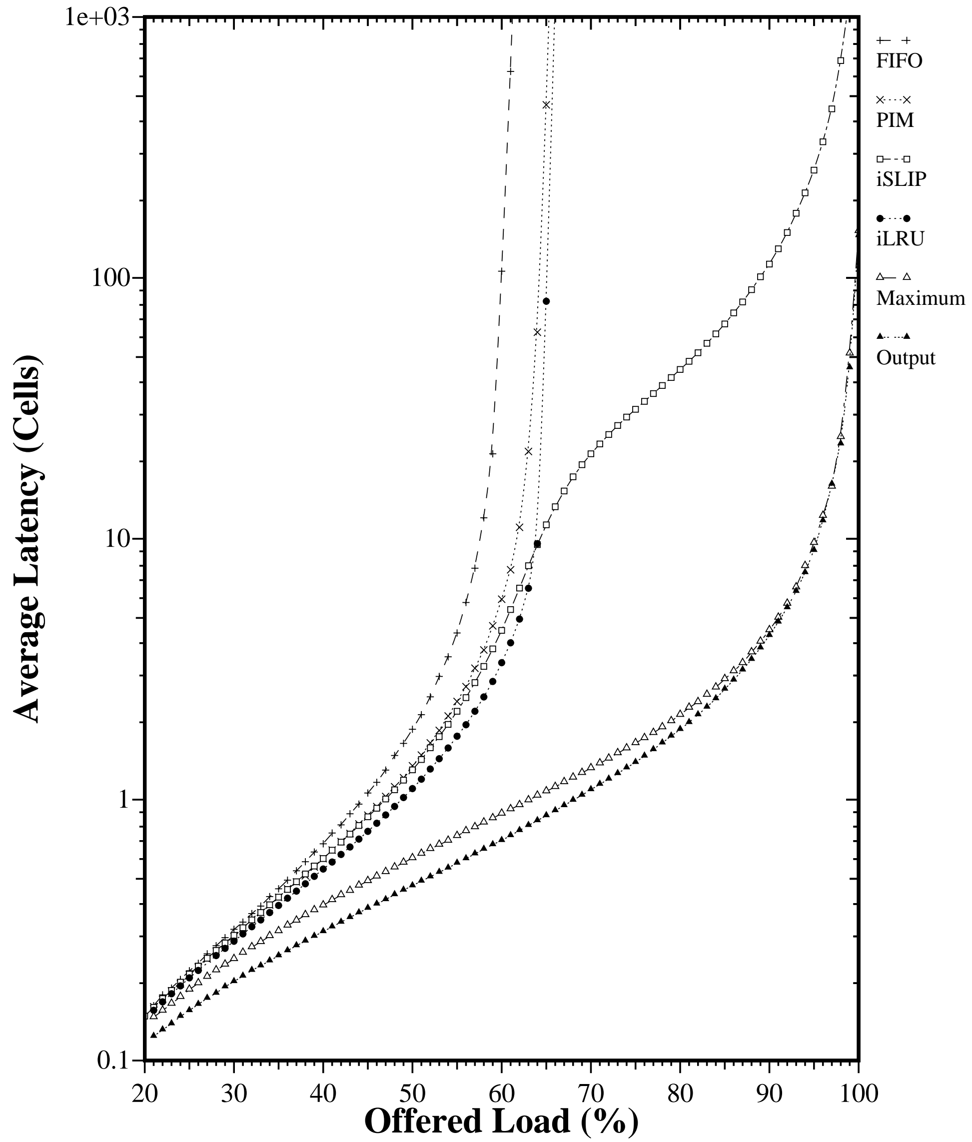

Figure 7: Delay vs. offered load of PIM, *i*SLIP, *i*LRU, FIFO, for a single iteration with uniform workload and Bernoulli arrivals. Log(latency) vs. offered load.

any given input will be able to connect to an output when all its input queues are occupied. The probability that no output will grant to a particular input is $((N-1)/N)^N$ which, for $N=16$, limits the probability that a flow will be made to approximately 63%.[2]

*i*SLIP can deliver throughput asymptotic to 100% of each link, but with a much higher average queueing delay than for either maximum matching or perfect output queueing. *i*SLIP achieves the high offered load because of the *desynchronization* discussed in Section 2.4. If all the input queues become non-empty, then the arbiters "slip" into a time division multiplexing configuration that is 100% efficient until the backlog is cleared. This behavior is illustrated in Figure 8 which compares the performance of *i*SLIP with a fixed time division multiplexor.

The tendency of the arbiters to desynchronize is illustrated in Figure 9. This graph shows the average number of arbiters that clash with another arbiter (i.e. are synchronized) at the beginning of each time slot. Note that as the load increases, the number of synchronized arbiters decreases to zero. In other words, the algorithm converges to time division multiplexing.

We might have expected *i*LRU to perform as well as *i*SLIP. But as we can see, it performs significantly worse. This is because the *desynchronization* of the arbiters does not happen as effectively as it does for *i*SLIP. Each schedule can become re-ordered at the end of each cell time which, over many cell times, tends to lead to a random ordering of the schedules. This in turn leads to a high probability that the pointers at two or more outputs will point to the same input: the same problem encountered by PIM. This explains why the performance for PIM and *i*LRU are very similar for the single iteration case.

The highest throughput attainable per time slot for input queueing is illustrated by the maximum size matching algorithm. This still leads to higher queueing delay than for perfect output queueing. With output queueing a cell is only competing for service with cells at the same output, whereas for input queueing a cell must compete with cells at the same input *and* output.

To clarify the difference between these algorithms over the whole offered load range, Figure 7 plots the logarithm of the latency versus the offered load. This shows that PIM and *i*SLIP are indistinguishable for a single iteration under low load, but separate as PIM becomes saturated. At approximately 70% offered load, the rate of increase of *i*SLIP with offered load decreases. This is due to the onset of desynchronization at higher offered loads.

5.1.2 Four Iterations

Figure 10 shows the average queueing delay (measured in cells) versus the average offered load for PIM, *i*SLIP and *i*LRU with *four* iterations and the same traffic pattern as before. We chose four iterations as this was the number of iterations performed in a single slot in the implementation of PIM [2]. There is a large improvement for PIM, *i*SLIP and *i*LRU for multiple iterations and their performance is now almost indistinguishable. All three algorithms are now significantly better than FIFO due to the elimination of HOL blocking and all approach the performance to maximum size matching. They still, of course, perform worse than output queueing: contention still exists at both inputs and outputs.

[2] As $N \to \infty$, maximum offered load for PIM with one iteration will tend to $1 - 1/e = 63.2\%$.

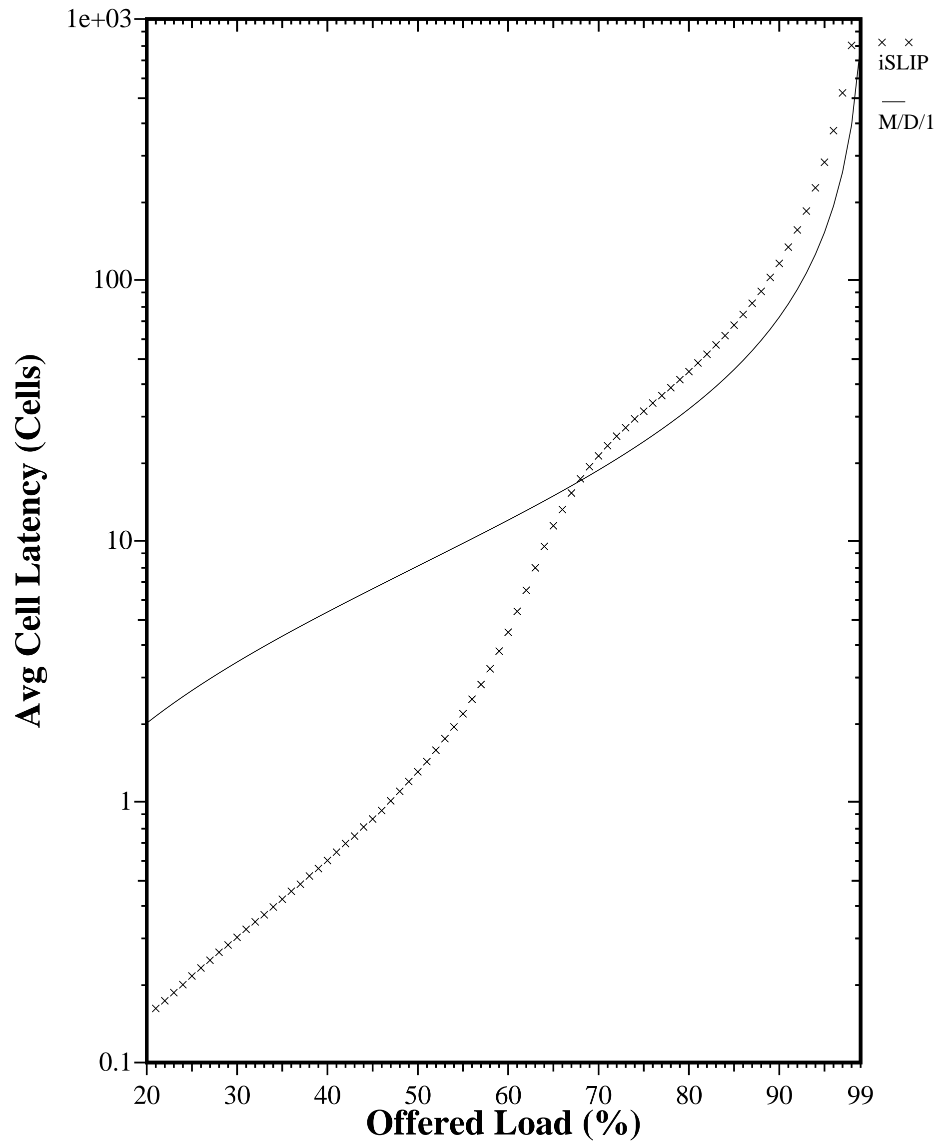

Figure 8: As the offered load increases, the performance of *i*SLIP approximates the performance of an M/D/1 time division multiplexor.

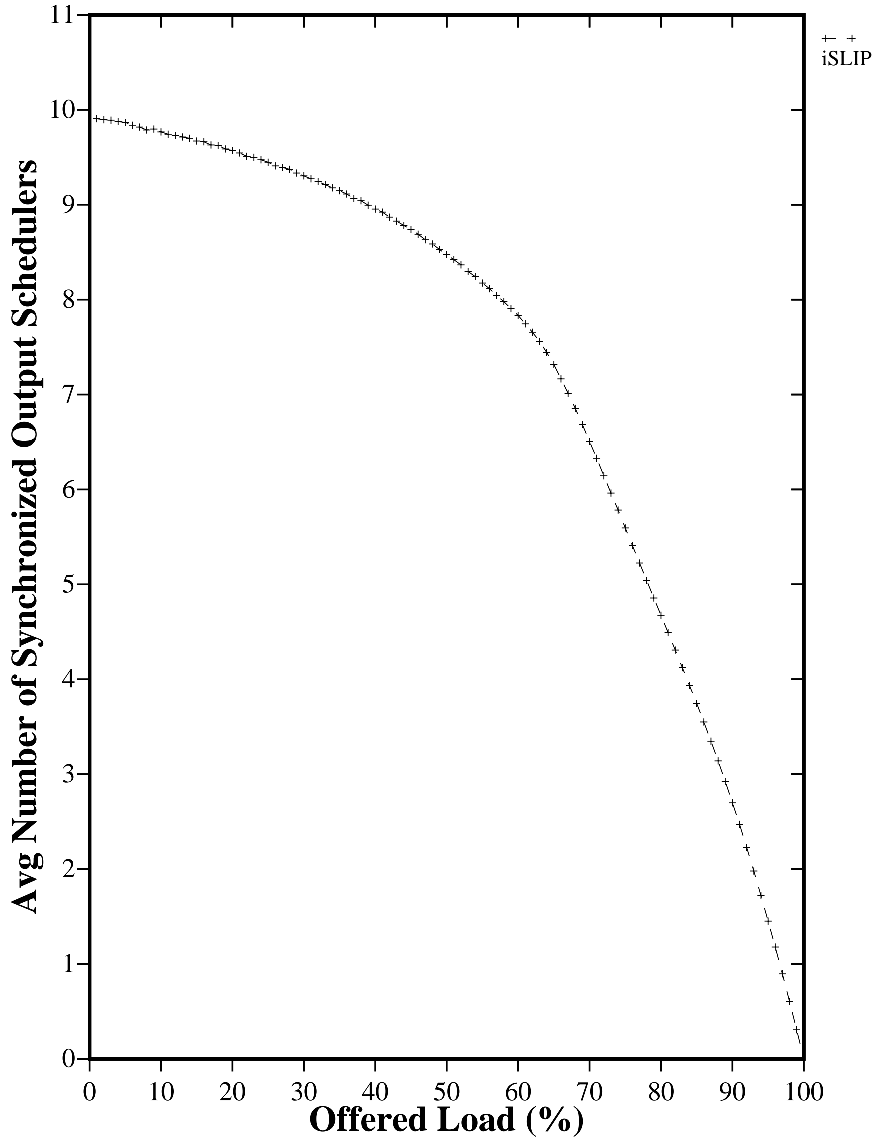

Figure 9: Under high offered load, the output arbiters become less and less sycnhronized. This graphs shows the average number of synchronized arbiters as a function of offered load for Bernoulli i.i.d. arrivals.

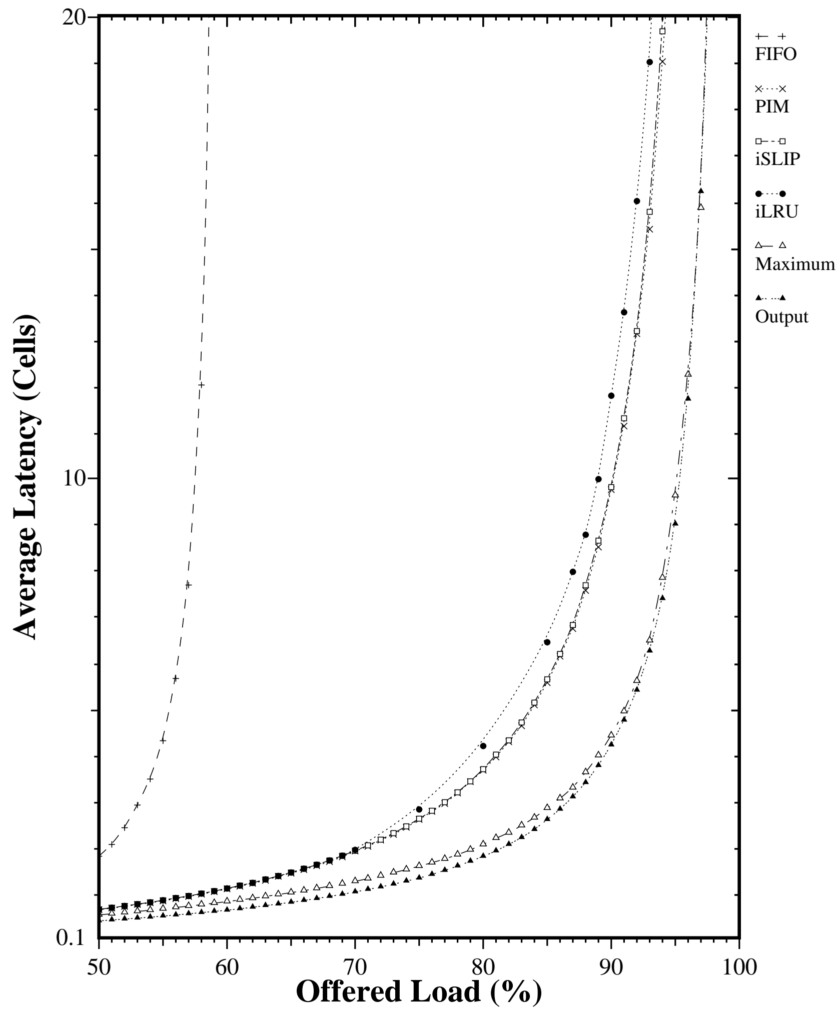

Figure 10: Delay vs. offered load for four iterations of PIM, *i*SLIP, *i*LRU, FIFO, maximum size matching and output queueing with uniform workload and Bernoulli arrivals.

	One iterations	N iterations
FIFO	58%	
PIM	63%	100%
***i*LRU**	63%	100%
***i*SLIP**	100%	100%
Output	100%	

Table 3: Comparison of asymptotic utilization for each algorithm, for Bernoulli arrivals.

In summary, it appears that for a single iteration, the determinism of *i*SLIP outperforms PIM and *i*LRU due to the desynchronization of its output arbiters. However, after just four iterations for a 16×16 switch, all three algorithms achieve almost maximum size matching and become indistinguishable.

Table **??** compares each algorithm for Bernoulli arrivals. The table shows the asymptotic utilization that is achievable for either one, or N iterations, where N is the number of switch ports. Note that while all of the maximal matching algorithms are able to achieve 100% throughput with N iterations, only *i*SLIP can do so in a single iteration.

5.2 Performance Under Asymmetric Workload

Real traffic is not usually uniformly distributed over all outputs, but is asymmetric. A common example of this would be a client-server work load in which a large number of clients communicate with a small number of servers. Although we cannot simulate all such work loads, we can highlight characteristics of each algorithm under certain patterns.

We have already seen in Figure 2 that maximum size matching can be so unfair under an asymmetric work load that some flows may receive no bandwidth at all. Although PIM, *i*SLIP and *i*LRU will not starve flows, all can perform badly under asymmetric work loads. Figure 11(a) is a workload for which PIM will allocate bandwidth *unfairly*. Ideally, every flow would receive 1/2 of a link bandwidth. However, in PIM if the input queues are non-empty, input 2 will be able to send cells on flows to outputs 1 and 2 *less often* than inputs 1 and 3. The algorithm tends to discriminate against inputs that have contention. The figure shows the actual bandwidth allocation for PIM with 2 or more iterations. With a single iteration, the values will be slightly different, but PIM is still unfair. It is straightforward to construct more complex examples for which PIM will allocate bandwidth even more unfairly.

On the other hand, *i*SLIP will allocate bandwidth fairly for this example: independent of the number of iterations. Whatever the starting conditions, so long as the input queues are occupied, *i*SLIP will slip into a round-robin schedule providing each flow with exactly 1/2 of the link bandwidth. But with just a single iteration, *i*SLIP has a different problem that is illustrated in Figure 11(b). In this example, if the arbiters start synchronized, they will visit each flow in turn allocating just two flows in each cell time (although up to $N-1$ flows could be made in

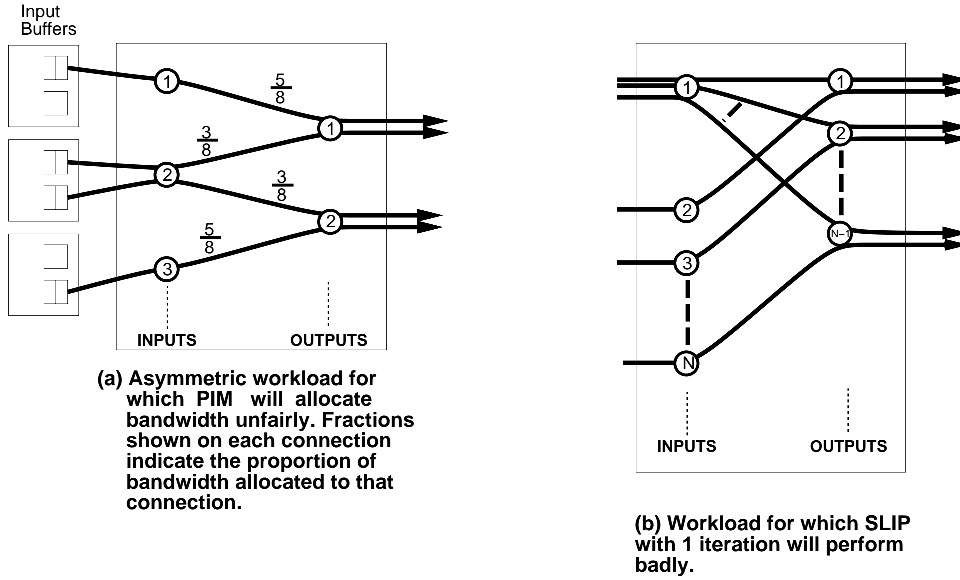

Figure 11: Examples of (a) unfairness for PIM and (b) poor performance for *i*SLIP.

more iterations). So, as a result of servicing each flow fairly and in turn, the algorithm does not slip into an efficient schedule with a high throughput. On the other hand, a single iteration of PIM will match about $(N-1)/2$ flows. Although this workload is poor for *i*SLIP with a single iteration, further iterations will efficiently find other flows.

5.3 Performance Under Bursty Traffic

Real traffic is not only asymmetric; it is also bursty. Many ways of modeling bursts in network traffic have been described [10, 14, 3, 18]. Recently, Leland et al. have demonstrated that measured network traffic is bursty at every level and that, contrary to popular belief, burstiness in Ethernet networks typically *intensifies* as the number of active traffic sources increases [18]. So it is very important to understand the performance of switches in the presence of bursty traffic, whether it is a small private switch with a small number of active users, or a large public switch with many thousands of aggregated flows.

We illustrate the effect of burstiness on PIM, *i*SLIP and *i*LRU using a simple packet train model similar to the one introduced in [14]. The source

alternately produces a burst (train) of full cells (all with the same destination) followed by one or more empty cells. The bursts contain a geometrically distributed number of cells. The model contains two parameters: the average burst length and the overall average utilization of the input link.

Figure 12 illustrates the performance of the different algorithms for bursty traffic. In this example, we use an average burst length of 32 cells and then vary the offered load. Although the performance of all the algorithms will degrade as the average burst length is increased, we found that they all degrade in a similar manner. An average burst length of 32 cells is representative

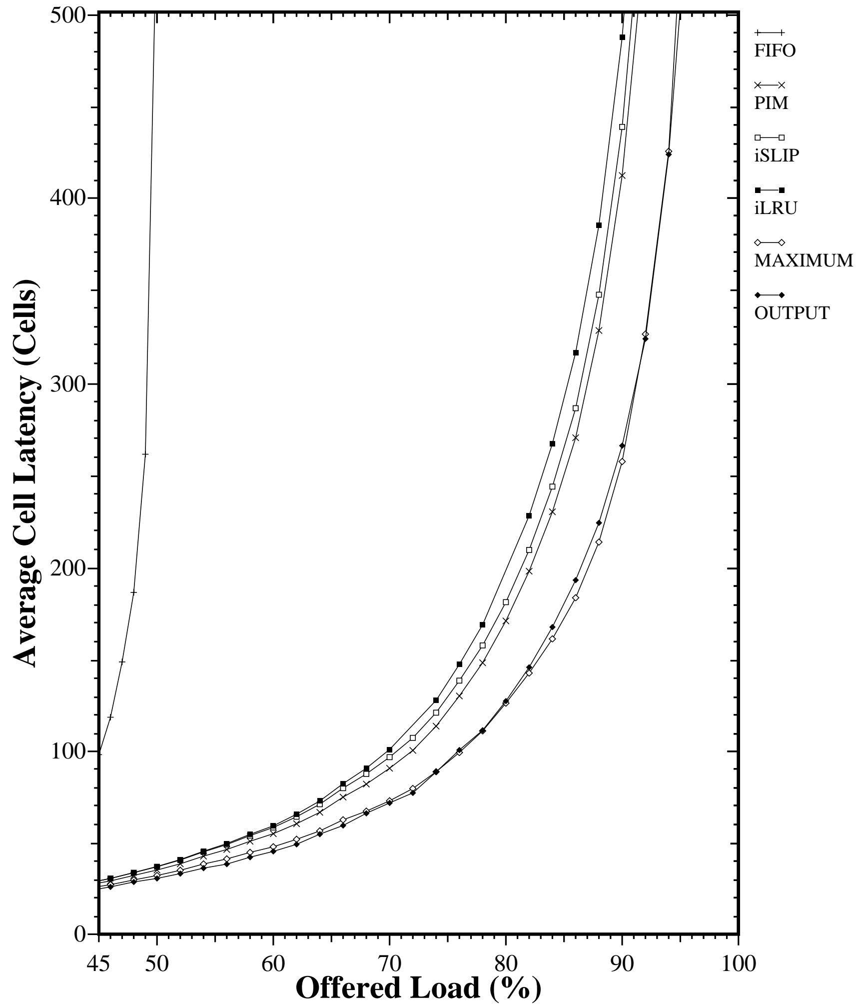

Figure 12: Delay vs. offered load for four iterations of PIM, *i*SLIP, *i*LRU, FIFO, maximum size matching and output queueing with uniform workload and bursty traffic.

of our findings.

PIM, *i*SLIP and *i*LRU are shown for four iterations and are once again compared to FIFO, maximum size matching and output queueing. The three most important characteristics of this graph are: (i) PIM, *i*SLIP and *i*LRU are again indistinguishable, as they were for non-bursty traffic, (ii) maximum size matching performs similarly to output queueing, but (iii) all PIM, *i*SLIP and *i*LRU are all noticeably worse than maximum size matching.

Once again, the three iterative algorithms achieve a very similar number of matches in four iterations. It appears that even for bursty traffic, flows not made in the first iteration of each algorithm are effectively and similarly filled in by subsequent iterations.

The similarity of maximum size matching to output queueing indicates that the performance for bursty traffic is not heavily influenced by the queueing policy. Recall that for maximum size matching, conflicts occur at both inputs *and* outputs, whereas for output queueing they occur

only at the outputs. Burstiness tends to concentrate the conflicts on outputs rather than inputs: each burst contains cells destined for the same output and each input will be dominated by a single burst at a time. As a result, the performance is limited by output contention.

6 Conclusions

We have described and compared three realizable iterative algorithms, PIM, iSLIP and iLRU, for scheduling flows in an input-queued switch. iSLIP and iLRU differ from PIM by allocating flows according to priorities which vary according to the traffic pattern.

Simulation results indicate that iSLIP will significantly outperform both PIM and iLRU for a single iteration with a uniform workload and Bernoulli arrivals. But as the number of iterations increases, the difference between the algorithms diminishes. With four iterations, for a 16×16 switch, all three algorithms do almost as well as maximum size matching for input-queued switches, and their performance becomes indistinguishable. Furthermore, we show that for multiple iterations, the algorithms perform almost identically for bursty traffic.

The algorithms differ considerably in the presence of asymmetric workloads. Workloads exist for which PIM will allocate bandwidth unfairly between flows, for any number of iterations. This appears not be the case for iSLIP with a single iteration. On the other hand, we can construct workloads for which iSLIP with a single iteration will only achieve a low throughput.

Finally, and perhaps most importantly, we estimate that iSLIP is an order of magnitude less complex to implement than PIM; iLRU appears to be more complex to implement than iSLIP with no performance gain. We believe that it is feasible to implement the scheduler for a 32×32 switch on a single chip using the iSLIP algorithm. The consequence is that iSLIP should be more scalable than PIM as link speeds increase in the future.

References

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

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

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

[4] ANSI. "Fiber distributed data interface (FDDI). Token ring media access control(MAC)," *ANSI Standard X3.139*, American National Standards Institute, Inc., New York.

[5] The ATM Forum. "The ATM Forum user-network interface specification, version 3.0, " *Prentice Hall Intl.,* New Jersey, 1993.

[6] Brayton, R.; Rudell, R.;Sangiovanni-Vincentelli, A.; and Wang, A. "MIS: A Multiple-Level Logic Optimization System", *IEEE Trans on CAD*, CAD-6 (6), pp.1062-1081, November 1987.

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

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

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

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

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

[12] Huang, A., Knauer, S. "Starlite: A wideband digital switch," *Proc. GLOBECOM '84* (1984), pp.121-125.

[13] Hui, J., Arthurs, E. "A broadband packet switch for integrated transport," *IEEE J. Selected Areas Commun.*, 5, 8, Oct 1987, pp 1264-1273.

[14] Jain, R., Routhier, S. A., "Packet Trains: measurements and a new model for computer network traffic," *IEEE J. Selected Areas in Commun.*, 4, 1986, pp.986-995.

[15] Karol, M., Hluchyj, M., and Morgan, S. "Input versus output queueing on a space division switch," *IEEE Trans. Comm*, 35(12) (1987) pp.1347-1356.

[16] Karol, M., Eng, K., Obara, H. "Improving the performance of input-queued ATM packet switches," *INFOCOM '92*, pp.110-115.

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

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

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

[20] McKeown, N., Varaiya, P., Walrand, J., "Scheduling Cells in an Input-Queued Switch," *Electronics Letters*, Vol. 29, No.25 pp.2174-2175, 9 Dec. 1993.

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

[22] McKeown, N., Anantharam, V., Walrand, J., "Achieving 100% Throughput in an Input-Queued Switch," *Proc. INFOCOM '96*, To appear.

[23] McKeown, N., Izzard, M., Mekkittikul, A., Ellersick, W., and Horowitz, M.; "The Tiny Tera: A Small High-Bandwidth Packet Switch Core," *IEEE Micro Magaine,* Vol. 17, No. 1, pp. 26 - 33, January-February 1997.

[24] Metcalfe, R., Boggs, D. "Ethernet: Distributed packet switching for local computer networks," *Communic. ACM*, 19, 7, July 1976, pp.395-404.

[25] Obara, H. "Optimum architecture for input queueing ATM switches," *Elect. Letters*, 28th March 1991, pp.555-557.

[26] Obara, H., Okamoto, S., and Hamazumi, Y. "Input and output queueing ATM switch architecture with spatial and temporal slot reservation control" *Elect. Letters*, 2nd Jan 1992, pp.22-24.

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

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

[29] Thacker, C., *Private communication.*

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

[31] Xilinx, Inc. "Xilinx: The programmable gate array data book" *Xilinx, Inc.*, 1991.