99

High Speed Switch Scheduling for Local Area Networks

Thomas E. Anderson, Susan S. Owicki, James B. Saxe, and Charles P. Thacker

April 26, 1993

digital

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

Systems Research Center

DEC's business and technology objectives require a strong research program. The Systems Research Center (SRC) and three other research laboratories are committed to filling that need.

SRC began recruiting its first research scientists in 1984—their charter, to advance the state of knowledge in all aspects of computer systems research. Our current work includes exploring high-performance personal computing, distributed computing, programming environments, system modelling techniques, specification technology, and tightly-coupled multiprocessors.

Our approach to both hardware and software research is to create and use real systems so that we can investigate their properties fully. Complex systems cannot be evaluated solely in the abstract. Based on this belief, our strategy is to demonstrate the technical and practical feasibility of our ideas by building prototypes and using them as daily tools. The experience we gain is useful in the short term in enabling us to refine our designs, and invaluable in the long term in helping us to advance the state of knowledge about those systems. Most of the major advances in information systems have come through this strategy, including time-sharing, the ArpaNet, and distributed personal computing.

SRC also performs work of a more mathematical flavor which complements our systems research. Some of this work is in established fields of theoretical computer science, such as the analysis of algorithms, computational geometry, and logics of programming. The rest of this work explores new ground motivated by problems that arise in our systems research.

DEC has a strong commitment to communicating the results and experience gained through pursuing these activities. The Company values the improved understanding that comes with exposing and testing our ideas within the research community. SRC will therefore report results in conferences, in professional journals, and in our research report series. We will seek users for our prototype systems among those with whom we have common research interests, and we will encourage collaboration with university researchers.

Robert W. Taylor, Director

High Speed Switch Scheduling for Local Area Networks

Thomas E. Anderson, Susan S. Owicki, James B. Saxe,
and Charles P. Thacker

April 26, 1993

Thomas E. Anderson's address is: Computer Science Division, University of California, Berkeley, CA 94720

Abstract

Current technology trends make it possible to build communication networks that can support high performance distributed computing. This paper describes issues in the design of a prototype switch for an arbitrary topology point-to-point network with link speeds of up to one gigabit per second. The switch deals in fixed-length ATM-style cells, which it can process at a rate of 37 million cells per second. It provides high bandwidth and low latency for datagram traffic. In addition, it supports real-time traffic by providing bandwidth reservations with guaranteed latency bounds. The key to the switch's operation is a technique called *parallel iterative matching*, which can quickly identify a set of conflict-free cells for transmission in a time slot. Bandwidth reservations are accommodated in the switch by building a fixed schedule for transporting cells from reserved flows across the switch; parallel iterative matching can fill unused slots with datagram traffic. Finally, we note that parallel iterative matching may not allocate bandwidth fairly among flows of datagram traffic. We describe a technique called *statistical matching*, which can be used to ensure fairness at the switch and to support applications with rapidly changing needs for guaranteed bandwidth.

Contents

1 Introduction **1**

2 Background and Related Work **3**

2.1 Switch Size 3

2.2 Internal Interconnect 4

2.3 Fixed-Length Cells vs. Variable-Length Packets 5

2.4 Buffer Organization 5

3 Parallel Iterative Matching **8**

3.1 Overview 8

3.2 Number of Iterations 10

3.3 Implementation Issues 11

3.4 Maximal vs. Maximum Matching 12

3.5 Performance of Iterative Matching 13

4 Real-Time Performance Guarantees **16**

5 Statistical Matching **21**

5.1 Motivation 21

5.2 Algorithm 23

5.3 Discussion 24

6 Summary **25**

Acknowledgements **25**

References **27**

A Parallel Iterative Matching: Number of Iterations **30**

B Latency and Buffer Space Bounds for CBR Traffic **31**

B.1 Bounded Latency 32

B.2 Bounded Buffer Space Requirements 33

C Statistical Matching Throughput **34**

1 Introduction

Over the past few years, several technology trends have converged to provide an opportunity for high performance distributed computing. Advances in laser and fiber optic technology have driven feasible link throughputs above a gigabit per second. Dynamic RAM chips have become cheap enough to be cost-effective at providing the large amounts of buffering needed at these very high link speeds. Moreover, quick routing and switching decisions are possible with current CMOS technology.

In combination, these trends make it possible to construct a practical local area network using multiple switches and gigabit-per-second point-to-point fiber links configured in an arbitrary topology. This kind of a network has several advantages [Schroeder et al. 91]. In contrast to networks like Ethernet [Metcalfe & Boggs 76] that use a broadcast physical medium, or networks like FDDI [Ame 87, Ame 88] based on a token ring, arbitrary topology point-to-point networks offer (i) aggregate network bandwidth that can be much larger than the throughput of a single link, (ii) the ability to add throughput incrementally by adding extra switches and links to match workload requirements, (iii) the potential for achieving lower latency, both by shortening path lengths and by eliminating the need to acquire control over the entire network before transmitting, and (iv) a more flexible approach to high availability using multiple redundant paths between hosts.

This paper studies the architectural issues in building high performance switches for arbitrary topology local area networks.

High performance networks have the potential to change the nature of distributed computing. Low latency and high throughput communication allow a much closer coupling of distributed systems than has been feasible in the past: with previous generation networks, the high cost of sending messages led programmers to carefully minimize the amount of network communication [Schroeder & Burrows 90]. Further, when combined with today's faster processors, faster networks can enable a new set of applications, such as desktop multimedia and the use of a network of workstations as a supercomputer.

A primary barrier to building high performance networks is the difficulty of high speed switching – of taking data arriving on an input link of a switch and quickly sending it out on the appropriate output link. The switching task is simplified if the data can be processed in fixed-length cells, as discussed in Section 2.3. Given fixed-length cells, switching involves at least two separate tasks:

- *scheduling* – choosing which cell to send during each time slot, when more than one cell is destined for the same output, and
- *data forwarding* – delivering the cell to the output once it has been scheduled.

Many high speed switch architectures use the same hardware for both scheduling and data forwarding; Starlite [Huang & Knauer 84], Knockout [Yeh et al. 87], and

Sunshine [Giacopelli et al. 91] are just a few of the switches that take this approach. If the input and output links of a switch are connected internally by a multi-stage interconnection network, the internal network can detect and resolve conflicts between cells as they work their way through the switch.

We take a different approach. We argue that for high speed switching, both now and in the future, switch scheduling can profitably be separated from data forwarding. By doing this, the hardware for each function can be specialized to the task. Because switch cost is dominated by the optical components needed to drive the fiber links, the added cost of separate hardware to do scheduling is justified, particularly if link utilization is improved as a result.

We observe that switch scheduling is simply an application of bipartite graph matching – each output must be paired with at most one input that has a cell destined for that output. Unfortunately, existing algorithms for bipartite matching are either too slow to be used in a high speed switch or do not maximally schedule the switch, sacrificing throughput.

A primary contribution of this paper is a randomized parallel algorithm, called *parallel iterative matching*, for finding a maximal bipartite match at high speed. (In practice, we run the algorithm for a fixed short time; in most cases it finds a maximal match.) Parallel iterative matching can be efficiently implemented in hardware for switches of moderate scale. Our work is motivated by the needs of AN2, an arbitrary topology network under development at Digital's Systems Research Center; we expect to begin using the network in mid-1993. Using only off-the-shelf field-programmable gate array technology [Xil 91], the AN2 switch using parallel iterative matching will be able to schedule a standard 53-byte ATM cell out each link of a 16 by 16 crossbar switch in the time for one cell to arrive at a link speed of one gigabit per second. This requires scheduling over 37 million cells per second. Cell latency across the switch is about 2.2 microseconds in the absence of contention. The switch does not drop cells, and it preserves the order of cells sent between a pair of hosts. If implemented in custom CMOS, we expect our algorithm to scale to larger switches and faster links.

Supporting the demands of new distributed applications requires more from a network than simply high throughput or low latency. The ability to provide guaranteed throughput and bounded latency is crucial to multimedia applications [Ferrari & Verma 90]. Even for applications that do not need guarantees, predictable and fair performance is often important to higher layers of protocol software [Jain 90, Zhang 91].

Parallel iterative matching does not by itself provide either fairness or guaranteed throughput. We present enhancements to our algorithm to provide these features. These enhancements pull from the bag of tricks of network and distributed system design – local decisions are more efficient if they can be made independently of global information, purely static scheduling can simplify performance analysis, and finally, randomness can de-synchronize decisions made by a large number of agents.

The remainder of the paper discusses these issues in more detail. Section 2 puts our work in context by describing related work. Section 3 presents the basic parallel scheduling algorithm. Section 4 explains how we provide guaranteed bandwidth and latency using the AN2 switch. Section 5 describes a technique called *statistical matching*, which uses additional randomness in the switching algorithm to support dynamic allocation of bandwidth. Section 6 provides a summary of our work.

2 Background and Related Work

Our goal is to build a local area network that supports high performance distributed computing; for this, a network must have high throughput, low latency, graceful degradation under heavy workloads, the ability to provide guaranteed performance to real-time applications, and performance that is both fair and predictable. The network should be capable of connecting anywhere from tens to thousands of workstations.

The network we envision consists of a collection of switches, links, and host network controllers. Data is injected into the network by the controller in a sending host; after traversing a sequence of links and switches, the data is delivered to the controller at the receiving host. Each link is point-to-point, connecting a single switch port to either a controller or to the port of another switch. Switches can be connected to each other and to controllers in any topology.

Routing in the network is based on *flows*, where a flow is a stream of cells between a pair of hosts. (Our network also supports multicast flows, but we will not discuss that here.) There may be multiple flows between a given pair of hosts, for example, with different performance guarantees. Each cell is tagged with an identifier for its flow. A routing table in each switch, built during network configuration, determines the output port for each flow. All cells from a flow take the same path through the network.

This paper focuses on the algorithms to be used for switch scheduling. But we must first provide context for our work by discussing other aspects of the AN2 switch design, including switch size, the configuration of the switch's internal interconnect, fixed-length cells vs. variable-length packets, and buffer organization.

2.1 Switch Size

A key parameter in designing a point-to-point network is the size of each switch. Part of the host-to-host interconnect is provided by the fiber optic links between switches and part by the silicon implementing the internal interconnect within each switch. In designing a network, we need to find an appropriate balance between using a large number of small switches or a small number of large switches.

At one extreme, very small switches are not cost-effective. The largest component in the cost of a local area fiber optic network comes from the optoelectronic devices in each switch that drive the fiber links. These devices account for almost half the cost of the 16 by 16 AN2 switch; we discuss the component costs of the AN2 switch in more detail in Section 3.3. A smaller switch size requires the network to have a larger number of fiber connections and thus a larger number of optoelectronic devices.

On the other hand, very large switches are often inappropriate for local area networks. While it is feasible to build switches with thousands of ports, such a switch would be unduly costly for sites that have only dozens of workstations. Smaller switches allow capacity to be added incrementally at low cost; smaller switches can also lower the cost of availability by making it less expensive for the network to have fully redundant paths.

For these reasons, our algorithms are designed for switches of moderate scale, in the range of 16 by 16 to 64 by 64. We expect that it will be some time before workstations are able to use a full gigabit-per-second link; for AN2, we are designing a special concentrator card to connect four workstations, each using slower speed link, to a single AN2 switch port. A single 16 by 16 AN2 switch can thus connect up to 64 workstations.

2.2 Internal Interconnect

Once the switch size has been decided, there are several approaches to designing the internal data path needed to transport cells from the inputs to the outputs of the switch. Probably the simplest approach to transporting data across a switch is to use shared memory or a shared bus. We do not pursue these techniques here, because they do not seem feasible for even moderate-sized switches with gigabit-per-second links, much less for the faster link speeds of the future.

Another uncomplicated approach is to connect inputs to outputs via a crossbar, using some external logic to control the crossbar, i.e., to decide which cells are forwarded over the crossbar during each time slot and to set up the crossbar for those cells. In the absence of a fast algorithm, however, scheduling the crossbar quickly becomes a performance bottleneck for all but the smallest switches.

Many switch architectures call for the switch's internal interconnection to be *self-routing* [Ahmadi & Denzel 89]. The switch is organized internally as a multi-stage network of smaller switches arranged in a butterfly, or more generally, in a banyan [Patel 79]. Cells placed into a banyan network are automatically routed and delivered to the correct output based solely on the information in the cell header.

Unlike a crossbar, however, banyan networks suffer from *internal blocking*. A cell destined for one output can be delayed (or even dropped) because of contention at the internal switches with cells destined for other outputs. This makes it difficult to provide guaranteed performance.

Internal blocking can be avoided by observing that banyan networks are internally non-blocking if cells are sorted according to output destination and then shuffled before being placed into the network [Huang & Knauer 84]. Thus, a common switch design is to put a Batcher sorting network [Batcher 68] and a shuffle exchange network in front of a normal banyan network. As with a crossbar, a cell may be sent from any input to any output provided no two cells are destined for the same output.

Our scheduling algorithm assumes that data can be forwarded through the switch with no internal blocking; this can be implemented using either a crossbar or a batcher-banyan network. Our prototype uses a crossbar because it is simpler and has lower latency. Even though the hardware for a crossbar for an N by N switch grows as $O(N^2)$, for moderate scale switches the cost of a crossbar is small relative to the rest of cost of the switch. In the AN2 prototype switch, for example, the crossbar accounts for less than 5% of the overall cost of the switch.

2.3 Fixed-Length Cells vs. Variable-Length Packets

Within our network, data is transmitted in fixed-length cells rather than variable-length packets. We support standard 53-byte ATM cells with 5-byte cell headers, although a 128-byte cell size with 8-byte cell headers would have simplified our implementation. Applications may still deal in variable-length packets. It is the responsibility of the network controller at the sending host to divide packets into cells, each containing the flow identifier for routing; the receiving controller re-assembles the cells into packets.

Using fixed-length cells has a number of advantages for switch design, despite the disadvantages that the switch must make more frequent scheduling decisions and that a greater proportion of the link bandwidth is consumed by the overhead of cell headers and internal fragmentation. The chief gain of using cells is that performance guarantees are easier to provide when the entire crossbar is re-configured after every cell time slot. In addition, fixed-length cells simplify random access buffer management (discussed in the next sub-section). Using cells can also improve packet latency for both short and long packets. Short packets do better because they can be interleaved over a link with long packets; a long packet cannot monopolize a connection for its entire duration. For long packets, cells simulate the performance of cut-through [Kermani & Kleinrock 79] while permitting a simpler store-and-forward implementation.

2.4 Buffer Organization

Even with an internally non-blocking switch, when several cells destined for the same output arrive in a time slot, at most one can actually leave the switch; the others must be buffered. There are many options for organizing the buffer pools. For example, buffers may be placed at the switch inputs or outputs; when placed at the inputs they may be strictly FIFO or allow random access. There has been considerable research on the impact of these alternatives. In this sub-section we review the work that is most relevant to our switch design.

The simplest approach is to maintain a FIFO queue of cells at each input; only the first cell in the queue is eligible for being transmitted during the next time slot. The difficulty with FIFO queueing is that when the cell at the head of an input queue is blocked, all cells behind it in the queue are prevented from being transmitted, even when the output link they need is idle. This is called *head-of-line (HOL) blocking*. Karol et al. [1987] have shown that head-of-line blocking limits switch throughput to 58% of each link, when the destinations of incoming cells are uniformly distributed among all outputs.

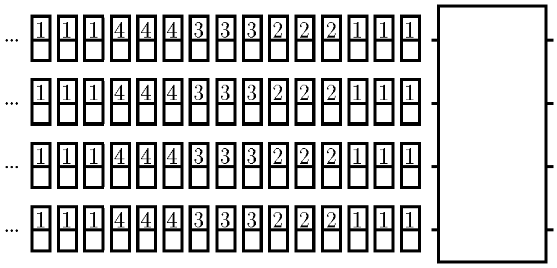

Figure 1: Performance Degradation Due to FIFO Queueing

Unfortunately, FIFO queueing can have even worse performance under certain traffic patterns. For example, if several input ports each receive a burst of cells for the same output, cells that arrive later for other outputs will be delayed while the burst cells are forwarded sequentially through the bottleneck link. If incoming traffic is periodic, Li [1988] shows that the aggregate switch throughput can be as small as the throughput of a single link, even for very large switches; this is called *stationary blocking*. Figure 1 illustrates this effect.[1] The worst case in Figure 1 occurs when scheduling priority rotates among inputs so that the first cell from each input is scheduled in turn. The example assumes for simplicity that cells can be sent out the same link they came in on; even if this is not the case, aggregate switch throughput can still be limited to twice the throughput of a single link. Note that without the restriction of FIFO queueing – that is, if any queued cell is eligible for forwarding – all of the switch's links could be fully utilized in steady state.

Various approaches have been proposed for avoiding the performance problems of FIFO input buffers. One is to expand the internal switch bandwidth so that it can transmit k cells to an output in a single time slot. This can be done by replicating the crossbar or, more typically, in a batcher-banyan switch by replicating the banyan part of the switch k times [Huang & Knauer 84]. Since only one cell can depart from an output during each slot, buffers are required at the outputs with this technique. In the limit, with enough internal bandwidth in an N by N switch to transmit N

[1] In this and other figures in this paper, input ports on the left and output ports on the right are shown as distinct entities. However, in an AN2 switch, the ith input and the ith output actually connect to the same full-duplex fiber optic link. The small boxes represent cells queued at each input; the number in each box corresponds to the output destination of that cell.

cells to the same output, there is no need for input buffers, since any pattern of arriving cells can be transmitted to the outputs. We will refer to this as *perfect* output queueing.

Perfect output queueing yields the best performance possible in a switch, because cells are only delayed due to contention for limited output link bandwidth, never due to contention internal to the switch. Unfortunately, the hardware cost of perfect output queueing is prohibitive for all but the smallest switches; the internal interconnect plus the buffers at each output must accomodate N times the link bandwidth. Thus it is more common for switches to be built with some small k chosen as the replication factor. If more than k cells arrive during a slot for a given output, not all of them can be forwarded immediately. Typically, the excess cells are simply dropped. While studies have shown that few cells are dropped with a uniform workload [Giacopelli et al. 91], unfortunately local area network traffic is rarely uniform. Instead, a common pattern is client-server communication, where a large fraction of incoming cells tend to be destined for the same output port, as described by Owicki and Karlin [1992]. Unlike previous generation networks, fiber links have very low error rates; the links we are using in AN2, for example, have a bit error rate of less than 10^{-12}. Thus, loss induced by the switch architecture will be more noticeable.

Another technique, often combined with the previous one [Giacopelli et al. 91], is to shunt blocked cells into a *re-circulating queue* that feeds back into extra ports in the batcher-banyan network. The re-circulated cells are then sorted, along with incoming cells, during the next time slot. Once again, if there is too much contention for outputs, some cells will be dropped.

Our switch takes the alternative approach of using random access input buffers. Cells that cannot be forwarded in a slot are retained at the input, and the first cell of any queued flow can be selected for transmission across the switch. This avoids the cell loss problem in the schemes above, but requires a more sophisticated algorithm for scheduling the cells to be transmitted in a slot.

While there have been several proposals for switches that use random access input buffers [Karol et al. 87, Tamir & Frazier 88, Obara & Yasushi 89, Karol et al. 92], the difficulty is in devising an algorithm that is both fast enough to schedule cells at high link speeds and effective enough to deliver high link throughput. For example, Hui and Arthurs [1987] use the batcher network to schedule the batcher-banyan. At first, only the header for the first queued cell at each input port is sent through the batcher network; an acknowledgement is returned indicating whether the cell is blocked or can be forwarded during this time slot. Karol et al. [1987] suggest that iteration can be used to increase switch throughput. In this approach, an input that loses the first round of the competition sends the header for the second cell in its queue on the second round, and so on. After some number of iterations k, the winning cells, header plus data, are sent through the batcher-banyan to the outputs. Note that this reduces the impact of head-of-line blocking but does not eliminate it, since only the first k cells in each queue are eligible for transmission.

3 Parallel Iterative Matching

In this section, we describe our algorithm for switch scheduling, first giving an overview, and then discussing its execution time and hardware cost. The section concludes with simulations of its performance relative to FIFO and output queueing.

3.1 Overview

The goal of our scheduling algorithm is to quickly find a conflict-free pairing of inputs to outputs, considering only those pairs with a queued cell to transmit between them. This pairing determines which inputs transmit cells over the crossbar to which outputs in a given time slot. With random access buffers, an input may transmit to any one of the outputs for which it has a queued cell, but the constraint is that each input can be matched to at most one output and each output to at most one input.

Our algorithm, *parallel iterative matching*, uses parallelism, randomness, and iteration to accomplish this goal efficiently. We iterate the following three steps (initially, all inputs and outputs are unmatched):

1. Each unmatched input sends a request to *every* output for which it has a buffered cell. This notifies an output of all its potential partners.

2. If an unmatched output receives any requests, it chooses one *randomly* to grant. The output notifies each input whether its request was granted.

3. If an input receives any grants, it chooses one to accept and notifies that output.

Each of these steps occurs independently and in parallel at each input/output port; there is no centralized scheduler. Yet at the end of one iteration of the protocol, we have a legal matching of inputs to outputs. More than one input can request the same output; the grant phase chooses among them, ensuring that each output is paired with at most one input. More than one output can grant to the same input (if the input made more than one request); the accept phase chooses among them, ensuring that each input is paired with at most one output.

While we have a legal matching after one iteration, there may remain unmatched inputs with queued cells for unmatched outputs. An output whose grant is not accepted may be able to be paired with an input, none of whose requests were granted. To address this, we repeat the request, grant, accept protocol, retaining the matches made in previous iterations. We iterate to "fill in the gaps" in the match left by previous iterations. However, there can be no head-of-line blocking in our approach, since we consider all potential connections at each iteration.

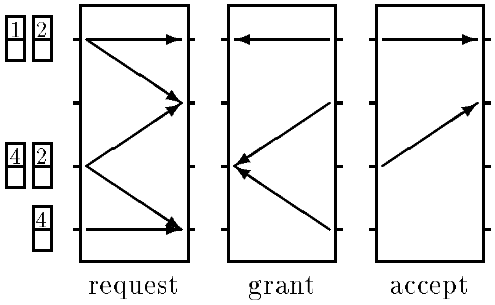

Figure 2: Parallel Iterative Matching: One Iteration

Figure 2 illustrates one iteration of parallel iterative matching. Five requests are made, three are granted, and two are accepted. Further, at the end of the first iteration, one request (from the bottom input to output 4) remains from an unmatched input to an unmatched output. This request is made, granted, and accepted during the second iteration; at this point, no further pairings can be added.

After a fixed number of iterations (discussed below), we use the result of parallel iterative matching to set up the crossbar for the next time slot. We then transmit cells over the crossbar, and re-run parallel iterative matching from scratch for the following time slot. Any remaining flows with queued cells can be considered for matching, as can any flows that have had cells arrive at the switch in the meantime.

Parallel iterative matching may forward cells through the switch in an order different from the order in which they arrived. However, the switch maintains a FIFO queue for each flow, so cells within a flow are not re-ordered. Only the first queued cell in each flow is eligible to be trasmitted over the crossbar. This use of FIFO queueing does not lead to head-of-line blocking, however: since all cells from a flow are routed to the same output, either none of the cells of a flow are blocked or all are.

Our algorithm can be generalized to handle switches with replicated switching fabrics. For instance, consider a batcher-banyan switch with k copies of the banyan network. With such a switch, up to k cells can be delivered to a single output during one time slot. (Note that this requires buffers at the outputs, since only one cell per slot can leave the output.) In this case, we can modify parallel iterative matching to allow each output to make up to k grants in step 2. In all other ways, the algorithm remains the same. An analogous change can be made for switch fabrics that allow inputs to forward more than one cell during any time slot. For the remainder of the paper, however, we assume that each input must be paired with at most one output and vice versa.

3.2 Number of Iterations

A key performance question is the number of iterations that it takes parallel iterative matching to complete, that is, to reach a point where no unmatched input has cells queued for any unmatched output. In the worst case, this can take N iterations for an N by N switch: if all outputs grant to the same input, only one of the grants can be accepted on each round. If this pattern were repeated, parallel iterative matching would be no faster than a sequential matching algorithm. On the other hand, in the best case, each output grants to a distinct input, in which case the algorithm takes only one iteration to finish.

To avoid the worst case behavior, we make it unlikely that outputs grant to the same input by having each output choose among requests using an independent random number. In Appendix A, we show that by using randomness, the algorithm completes in an average of $O(\log N)$ iterations; this result is independent of the initial pattern of input requests. The key to the proof is that each iteration resolves, either by matching or by removing from future consideration, an average of at least 3/4 of the remaining unresolved requests.

The AN2 prototype switch runs parallel iterative matching for four iterations, rather than iterating until no more matches can be added. There is a fixed amount of time to schedule the switch – the time to receive one cell at link speed. In our current implementation using 53-byte ATM cells, field-programmable gate arrays, and 1.0 gigabit-per-second links, there is slightly more than enough time for four iterations.

Pr{input i has a cell for output j}	Number of Iterations (K)			
	1	2	3	4
.10	87%	99.8%	100%	
.25	75%	97.6%	99.97%	100%
.50	69%	93%	99.6%	99.997%
.75	66%	90%	98.6%	99.97%
1.0	64%	88%	97%	99.9%

Table 1: Percentage of Total Matches Found Within K Iterations: Uniform Workload

To determine how many iterations it would take in practice for parallel iterative matching to complete, we simulated the algorithm on a variety of request patterns. Table 1 shows the results of these tests for a 16 by 16 switch. The first column lists the probability p that there is a cell queued, and thus a request, for a given input-output pair; several hundred thousand patterns were generated for each value of p. The remaining columns show the percentages of matches found within one through four iterations, where 100% represents the number of matches found by running iterative matching to completion. Table 1 shows that additional matches are hardly ever found after four iterations in a 16 by 16 switch; we observed similar results for client-server request patterns.

3.3 Implementation Issues

We next consider issues in implementing parallel iterative matching in hardware.

Functional Unit	Prototype Cost	Production Cost (est.)
Optoelectronics	48%	63%
Crossbar	4%	5%
Buffer RAM/Logic	21%	19%
Scheduling Logic	10%	3%
Routing/Control CPU	17%	10%

Table 2: AN2 Switch Component Costs, as Proportion of Total Switch Cost

First, note that the overall cost of implementing parallel iterative matching is small relative to the rest of the cost of the AN2 switch. In addition to switch scheduling hardware, the AN2 switch has optoelectronics for receiving and transmitting cells over the fiber links, a crossbar for forwarding cells from inputs to outputs, cell buffers at each input port along with logic for managing the buffers, and a control processor for managing routing tables and the pre-computed schedule described in the next section. Table 2 lists the hardware cost of each of these functional units as a percentage of the total cost of a 16 by 16 AN2 switch. Table 2 considers only the cost of the hardware devices needed by each functional unit, not the engineering cost of designing the switch logic. We list both the actual costs for our prototype switch and our estimate of the costs for a production version of the switch. To simplify the design process, we implemented most of the logic in the AN2 prototype with Xilinx field-programmable gate arrays [Xil 91]. A production system would use a more cost-effective technology, such as custom CMOS, reducing cost of the random logic needed to implement parallel iterative matching relative to the rest of the cost of the switch. In either the prototype or the production version, the cost of the optoelectronics dominates the cost of the switch.

Parallel iterative matching requires random access input buffers, so that any input-output pair with a queued cell can be matched during the next time slot. We implement this by organizing the input buffers into lists. Each flow has its own FIFO queue of buffered cells. A flow is *eligible* for scheduling if it has at least one cell queued. A list of eligible flows is kept for each input-output pair. If there is at least one eligible flow for a given input-output pair, the input requests the output during parallel iterative matching. If the request is granted, one of the eligible flows is chosen for scheduling in round-robin fashion. When a cell arrives, it is put on the queue for its flow, and its flow is added to the list of eligible flows if it is not already there. When a cell departs the switch, its flow may need to be removed from the list of eligible flows. Much of this mechanism is also needed for providing guaranteed performance to flows as described in the next section.

We implement the request, grant, accept protocol by running a wire between every input and output. Even though this requires hardware that grows as $O(N^2)$ for an N by N switch, this is not a significant portion of the switch cost, at least for moderate scale switches. The request and grant signals can be encoded by a single bit on the appropriate wire. As a simple optimization, no separate communication is required in step 3 to indicate which grants are accepted. Instead, when an input accepts an output's grant, it simply continues to request that output on subsequent iterations, but drops all other requests. Once an output grants to an input, it continues to grant to the same input on subsequent iterations unless the input drops its request.

The thorniest hardware implementation problem is randomly selecting one among k requesting inputs. The obvious way to do this is to generate a pseudo-random number between 1 and k, but we are examining ways of doing more efficient random selection. For instance, for moderate-scale switches, the selection can be efficiently implemented using tables of precomputed values. Our simulations indicate that the number of iterations needed by parallel iterative matching is relatively insensitive to the technique used to approximate randomness.

3.4 Maximal vs. Maximum Matching

It is reasonable to consider whether a switch scheduling algorithm more sophisticated than parallel iterative matching might achieve better switch throughput, although perhaps with higher hardware cost. Scheduling a switch with random access input buffers is an application of bipartite graph matching [Tarjan 83]. Switch inputs and outputs form the nodes of a bipartite graph; the edges are the connections needed by queued cells.

Bipartite graph matching has been studied extensively. There are two interesting kinds of bipartite matches. A *maximum* match is one that pairs the maximum number of inputs and outputs together; there is no other pairing that matches more inputs and outputs. A *maximal* match is one for which pairings cannot be trivially added; each node is either matched or has no edge to an unmatched node. A maximum match must of course be maximal, but the reverse is not true; it may be possible to improve a maximal match by deleting some pairings and adding others.

We designed parallel iterative matching to find a maximal match, even though link utilization would be better with a maximum match. One reason was the length of time we had to make a scheduling decision; we saw no way using current technology to do maximum matching under the time constraint imposed by 53-byte ATM cells and gigabit-per-second links. Finding a maximum match for an N by N graph with M edges can take $O(N \times (N + M))$ time. Although Karp et al. [1990] give a randomized algorithm that comes close on average to finding a maximum match, even that algorithm can take $O(N + M)$ time. As discussed above, our parallel algorithm finds a maximal match in logarithmic time, on average.

Another disadvantage of maximum matching is that it can lead to starvation.

The example we used to explain parallel iterative matching (Figure 2) also illustrates this possibility. Assuming a sufficient supply of incoming cells, maximum matching would never connect input 1 with output 2. In contrast, parallel iterative matching does not incur starvation. Since every output grants randomly among requests, an input will eventually receive a grant from every output it requests. Provided inputs choose among grants in a round-robin or other fair fashion, every queued cell will eventually be transmitted.

In the worst case, the number of pairings in a maximal match can be as small as 50% of the number of pairings in a maximum match. However, the simulations reported below indicate that even if it were possible to do maximum matching (or some even more sophisticated algorithm) in one ATM cell time slot at gigabit link speeds, there could be only a marginal benefit, since parallel iterative matching comes close to the optimal switch performance of perfect output queueing.

3.5 Performance of Iterative Matching

To evaluate the performance of parallel iterative matching, we compared it to FIFO queueing and output queueing by simulating each under a variety of workloads on a 16 by 16 switch. All simulations were run for long enough to eliminate the effect of any initial transient. As noted in Section 2, FIFO queueing is simple to implement, but can have performance problems. Perfect output queueing is infeasible to implement even for a moderate scale gigabit switch, but provides a good basis for comparison, because it indicates the optimal switch performance given unlimited hardware resources.

Figure 3 shows average queueing delay (in cell time slots) vs. offered load for the three scheduling algorithms. Offered load is the probability that a cell arrives (departs) on a given link in a given time slot. The destinations of arriving cells are uniformly distributed among the outputs.

Figure 3 illustrates several points:

- At low loads, there is little difference in performance between the three algorithms. When there are few queued cells, it does not matter (beyond hardware implementation cost) which switch scheduling algorithm is used.

- At moderately high loads, neither parallel iterative matching nor output queueing is limited, as FIFO queueing is, by head-of-line blocking. Parallel iterative matching does have significantly higher queueing delay than perfect output queuing. This is because, with output queueing, a queued cell is delayed only by other cells at the same output. With parallel iterative matching, a queued cell must compete for the crossbar with both cells queued at the same input *and* cells destined for the same output.

- The peak switch throughput of parallel iterative matching comes quite close to that of perfect output queueing. Even at very high loads, the queueing delay

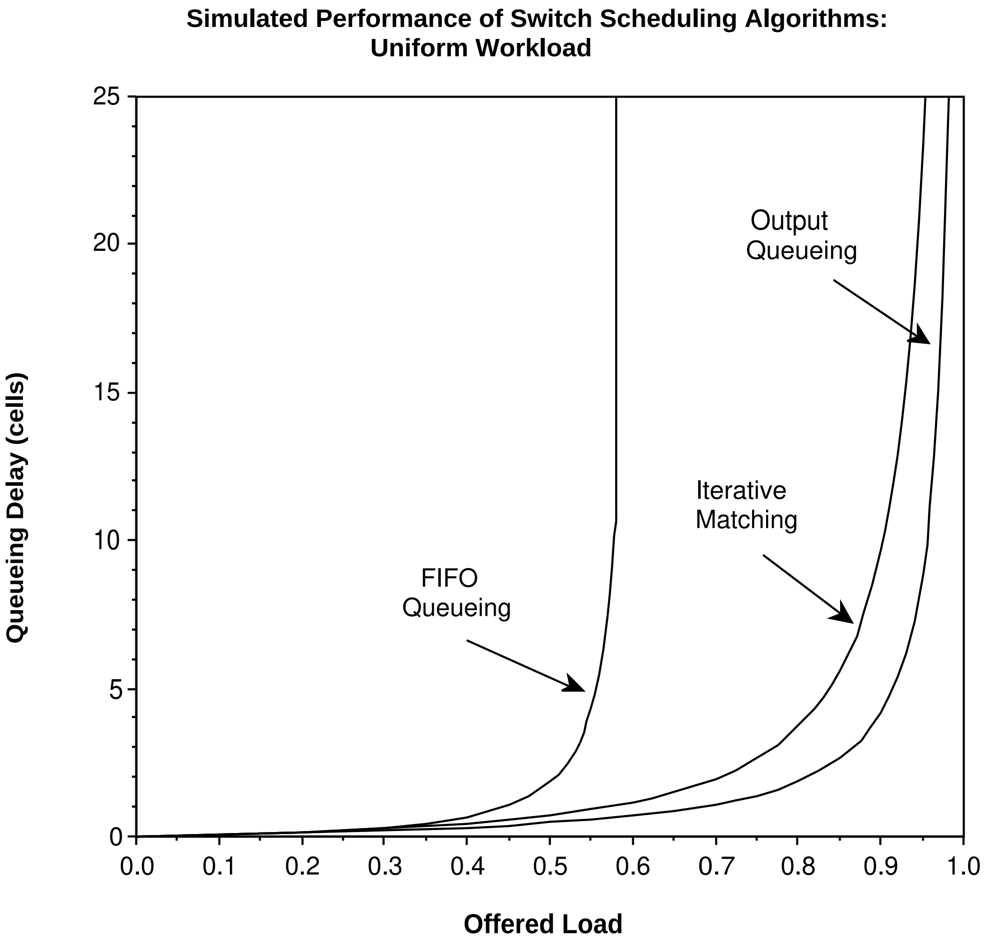

Figure 3: Simulated Performance of Switch Scheduling Algorithms: Uniform Workload

for parallel iterative matching is quite reasonable. For instance, our switch, with 53-byte ATM cells and gigabit-per-second links, will forward an arriving cell in an average of less than 13 μsec. when the links are being used at 95% of capacity.

Figure 4, shows average queueing delay vs. offered load under a non-uniform client-server workload. In defining the workload, four of the sixteen ports were assumed to connect to servers, the remainder to clients. Destinations for arriving cells were randomly chosen in such a way that client-client connections carried only 5% of the traffic of client-server or server-server connections. Here offered load refers to the load on a server link.

The results in Figure 4 are qualitatively similar to those of Figure 3. FIFO queueing still suffers from head-of-line blocking, limiting its maximum possible through-

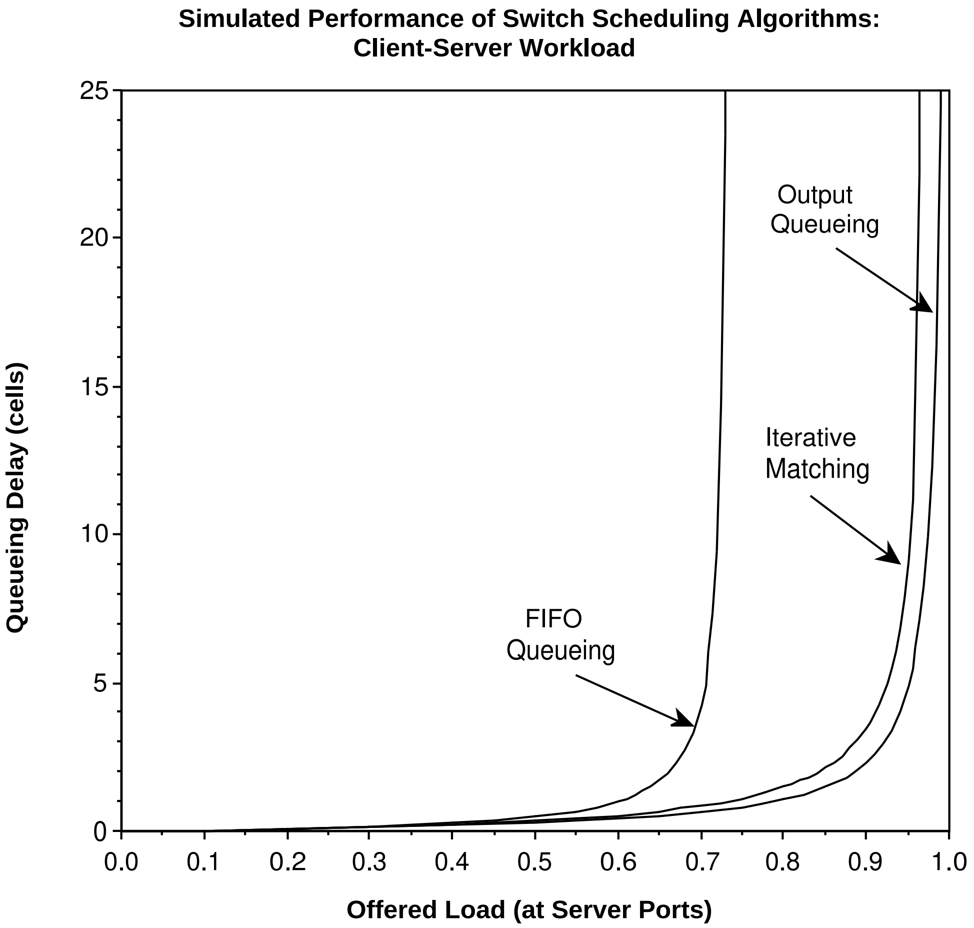

Figure 4: Simulated Performance of Switch Scheduling Algorithms: Client-Server Workload

put. Parallel iterative matching performs well on this workload, coming even closer to optimal than in the uniform case. The results were similar for other client/server traffic ratios and for different numbers of servers.

Finally, Figure 5 shows the impact of the number of iterations on the performance of parallel iterative matching. Here the number of iterations was varied, using the uniform workload of Figure 3. The result confirms that for a 16 by 16 switch, there is no significant benefit to running parallel iterative matching for more than four iterations; the queueing delay with four iterations is everywhere within 0.5% of the delay assuming parallel iterative matching is run to completion. Note that even with one iteration, parallel iterative matching does better than FIFO queueing.

To summarize, parallel iterative matching makes it possible for the switch to achieve a nearly ideal match in a short time. Moreover, the hardware requirements

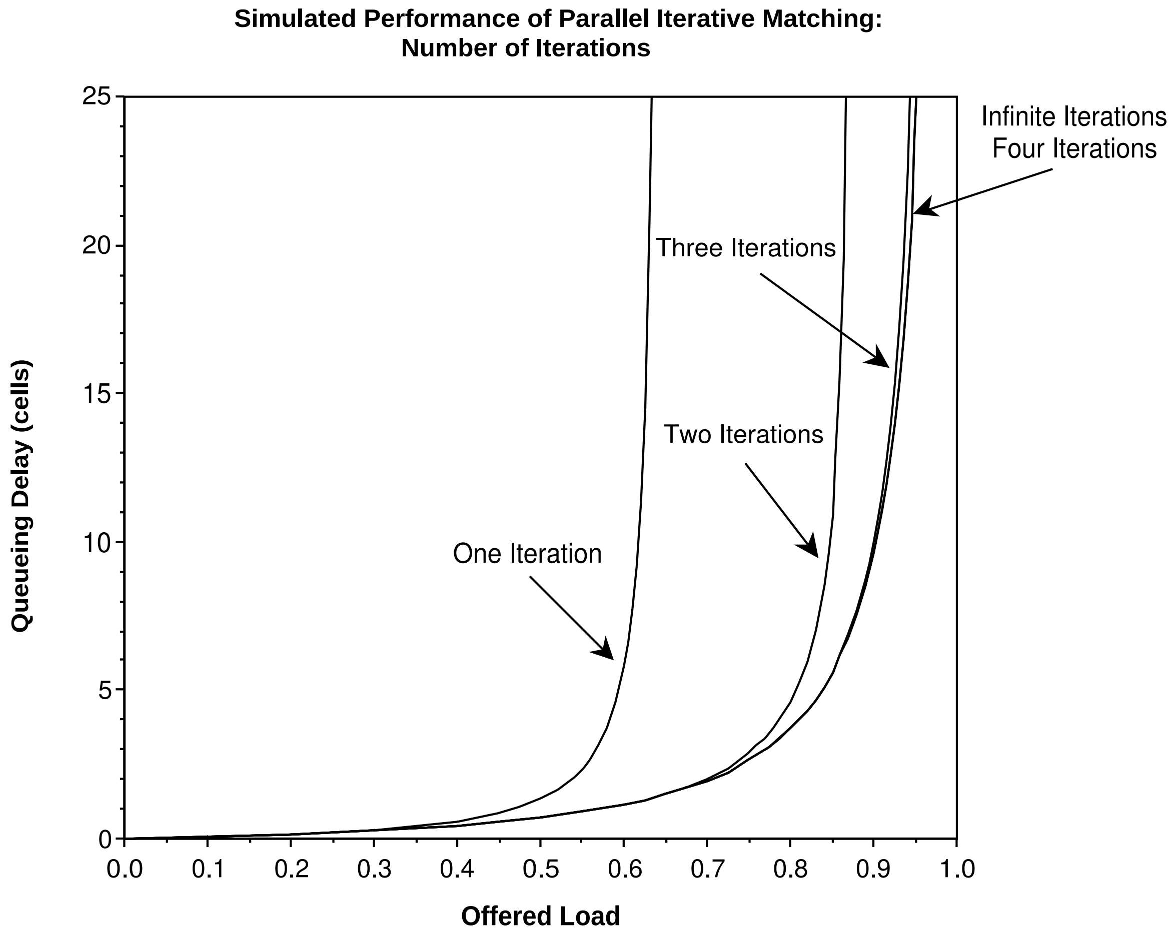

Figure 5: Simulated Performance of Parallel Iterative Matching: Uniform Workload

are modest enough to make parallel iterative matching practical for high speed switching.

4 Real-Time Performance Guarantees

As network and processor speeds increase, new types of high performance distributed applications become feasible. Supporting the demands of these applications requires more from a network than just high throughput or low latency. Parallel iterative matching, while fast and effective at keeping links utilized, cannot by itself provide all of the needed services. The remainder of this paper discusses these issues and suggests ways of augmenting the basic algorithm to address them.

One important class of applications are those that depend on real-time performance guarantees. For example, multimedia applications must display video frames at fixed intervals. They require that the network provide a certain minimum bandwidth and a bounded latency for cell delivery. Following the conventions of the ATM community, we will refer to traffic with reserved bandwidth requirements as *constant bit rate (CBR)*, and refer to other traffic as *variable bit rate (VBR)*. (VBR traffic is often called *datagram* traffic). Switches distinguish VBR and CBR cells based on the flow identifier in the cell header.

To ensure guaranteed performance, an application issues a request to the network to reserve a certain bandwidth and latency bound for a CBR flow [Ferrari & Verma 90]. If the request can be met without violating any existing guarantees, the network grants it and reserves the required resources on a fixed path between source and destination. The application can then transmit cells at a rate up to its requested bandwidth, and the network ensures that they are delivered on time. By contrast, applications can transmit VBR cells with no prior arrangement. If the network becomes heavily loaded, VBR cells may suffer arbitrary delays. But CBR performance guarantees are met no matter how high the load of VBR traffic.

With CBR traffic, since we know the offered load in advance, we can afford to spend time to pre-compute a schedule at each switch to accomodate the reservations. By contrast, parallel iterative matching was devised to rapidly schedule the switch in response to whatever VBR traffic arrives at the switch.

Our contribution is showing how to implement performance guarantees in a network of input-buffered switches with unsynchronized clocks. The rest of this section describes our approach to CBR traffic. We first describe the form of a bandwidth request and the criterion used to determine whether it can be accepted. We next show how a switch can be scheduled to meet bandwidth guarantees. Finally, we show that buffers for CBR traffic can be statically allocated and the latency of CBR cells can be bounded, even when network switch clock rates are unsynchronized. Our approach smoothly integrates both CBR and VBR traffic; VBR cells can consume all of the network bandwidth unused by CBR cells.

Bandwidth allocations are made on the basis of *frames* which consist of a fixed number of *slots*, where a slot is the time required to transmit one cell [Golestani 90]. An application's bandwidth request is expressed as a certain number of cells per frame; if the request is granted, each switch in the path schedules the flow into that number of frame slots, and repeats the frame schedule to deliver the promised throughput. Frame boundaries are internal to the switch; they are not encoded on the link.

Frame size is a parameter of the network. A larger frame size allows for finer granularity in bandwidth allocation; we will see later that smaller frames yield lower latency. The frame size in our prototype switch is 1000 slots; a frame takes less than half a millisecond to transmit. This leads to latency bounds that seem acceptable for multimedia applications, the most likely use for CBR guarantees.

When a request is issued, network management software must determine whether it can be granted. In our approach, this is possible if there is a path from source to destination on which each link's uncommitted capacity can accomodate the requested bandwidth. If network software finds such a path, it grants the request, and notifies the involved switches of the additional reservation. The application can then send up to the reserved number of cells each frame. The host controller or the first switch on the flow's path can meter the rate at which cells enter the network; if the application exceeds its reservation, the excess cells may be dropped. Alternatively, excess cells may be allowed into the network, and any switch may drop cells for a flow that exceeds its allocation of buffers.

Note that this allocation criterion allows 100% of the link bandwidth to be reserved (although we shall see later that a small amount of bandwidth is lost in dealing with clock drift). Meeting this throughput level is straightforward with perfect output queueing [Golestani 90, Kalmanek et al. 90], but this assumes the switch has enough internal bandwidth that it never needs to drop cells under any pattern of arriving CBR cells. With input buffering, parallel iterative matching is not capable of guaranteeing this throughput level.

Instead, in AN2, CBR traffic is handled by having each switch build an explicit schedule of input-output pairings for each slot in a frame; the frame schedule is constructed to accomodate the guaranteed traffic through the switch. The Slepian-Duguid theorem [Hui 90] implies that such a schedule can be found for any traffic pattern, so long as the number of cells per frame from any input or to any output is no more than the number of slots in a frame, in other words, so long as the link bandwidth is not over-committed. When a new reservation is made, it may be necessary to rearrange the connections in the schedule. We are free to rearrange the schedule, since our guarantees depend only on delivering the reserved number of cells per frame for each flow, not on which slot in the frame is assigned to each flow. The slot assignment can be changed dynamically without disrupting guaranteed performance.

An algorithm for computing the frame schedule is as follows [Hui 90]. Suppose a reservation is to be added for k cells per frame from input P to output Q; P and Q have k free slots per frame, or else the reservation cannot be accomodated. We add the reservation to the schedule one cell at a time. First, if there is a slot in the schedule where both P and Q are unreserved, the connection can be added to that slot. Otherwise, we must find a slot where P is unreserved, and a different slot where Q is unreserved. These slots must exist if P and Q are not over-committed. The algorithm swaps pairings between these two slots, starting by adding the connection from P to Q to either of the two slots. This will cause a conflict with an existing connection (for instance, from R to Q); this connection is removed and added to the other slot. In turn, this can cause a conflict with an existing connection (from R to S), which is removed and added to the first slot. The process is repeated until no conflict remains. It can be shown that this algorithm always terminates.

Figure 6 provides an example of reservations and a schedule for a frame size of 3 slots; Figure 7 illustrates the modification to the schedule needed to accomodate

Reservations (cells per frame)

Input	Output 1	2	3	4
1		1	1	1
2	2			
3		2		1
4	1		1	

Schedule

Slot 1	$1 \rightarrow 3$	$2 \rightarrow 1$	$3 \rightarrow 2$	
Slot 2	$1 \rightarrow 4$	$2 \rightarrow 1$	$3 \rightarrow 2$	$4 \rightarrow 3$
Slot 3	$1 \rightarrow 2$		$3 \rightarrow 4$	$4 \rightarrow 1$

Figure 6: CBR Traffic: Reservations and Schedule

Reservations (cells per frame)

Input	Output 1	2	3	4
1		1	1	1
2	2			1
3		2		1
4	1		1	

Schedule

Slot 1	$1 \rightarrow 2$	$2 \rightarrow 1$	$3 \rightarrow 4$	
Slot 2	$1 \rightarrow 4$	$2 \rightarrow 1$	$3 \rightarrow 2$	$4 \rightarrow 3$
Slot 3	$1 \rightarrow 3$	$2 \rightarrow 4$	$3 \rightarrow 2$	$4 \rightarrow 1$

Figure 7: CBR Traffic with Added Reservation

an additional reservation of one cell per frame from input 2 to output 4. Because there is no slot in which both input 2 and output 4 are free, the existing schedule must be shuffled in order to accommodate the new flow. In the example, we added the connection to slot 3, and swapped several connections between slots 1 and 3.

Computing a new schedule may require a number of steps proportional to the size of the reservation (in cells/frame) $\times$ N, for an N by N switch. However, the test for whether a switch can accommodate a new flow is much simpler; it is possible so long as the input and output link each have adequate unreserved capacity. Once a feasible path is found, the selected switches can compute their new schedules in parallel.

CBR cells are routed across the switch during scheduled slots. VBR cells are transmitted during slots not used by CBR cells. For example, in Figure 6, a VBR cell can be routed from input 2 to output 3 during the third slot. In addition, VBR cells can use an allocated slot if no cell from the scheduled flow is present at the switch.

Pre-scheduling the switch ensures that there is adequate bandwidth at each switch and link for CBR traffic. It is also necessary to have enough buffer space at each switch to hold cells until they can be transmitted; otherwise, some cells would be lost. The AN2 switch statically allocates enough buffer space for CBR traffic. VBR cells use a different set of buffers, which are subject to flow control.

In a network where switch clock rates are synchronized, as in the telephone network, a switch needs enough buffer space at each input link for two frames worth of cells [Golestani 90, Zhang & Keshav 91]. Note that one frame of buffering is not enough, because the frame boundaries may not be the same at both switches, and because the switches can rearrange their schedules from one frame to the next.

The situation becomes more complicated when each switch or controller's clock can run at a slightly different rate. The time to transmit a frame of cells is determined by the local clock rate at the switch or controller. Thus, an upstream switch or controller with a fast clock rate can overrun the buffer space for a slow downstream switch, by sending cells at a faster rate than the downstream switch can forward cells. More deviously, a switch may run more slowly for a time, building up a backlog of cells, then run faster, dumping the backlog onto the downstream switch.

Our solution assumes the clock rates on all switches and controllers are within some tolerance of the same rate. We then constrain the network controllers to insert cells at a slower rate than that of the slowest possible downstream switch. We do this by adding extra empty slots to the end of each controller (but not switch) frame, so that even if the controller has a fast clock and a switch has a slow clock, the controller's frame will still take longer than the switch's frame. Because the rate at which controllers insert cells is constrained, a fast switch can only temporarily overrun a slower downstream switch; we need to allocate enough buffer space to accomodate these temporary bursts. Over the long run, cells can arrive at a switch only at the rate at which they are inserted by the network controller.

We derive the exact bound on the buffer space required in Appendix B as a function of network parameters: the switch and controller frame sizes, the network diameter, and the clock error limits. Four or five frames of buffers are sufficient for values of these parameters that are reasonable for local area networks.

Now let us consider latency guarantees. If switch clocks are synchronized, a cell can be delayed at most two frame times at each switch on its path [Golestani 90, Zhang & Keshav 91]. Let p be the number of hops in the cell's path, f the time to transmit a frame, and l an upper bound on link latency plus switch overhead for processing a cell. Then the total latency for a cell is less than $p(2f + l)$. When switches are not synchronized, the delay experienced by a cell at a particular switch may be larger than $(2f + l)$, but the *end-to-end delay* is still bounded by $p(2f + l)$. Again, the derivation is presented in Appendix B. This yields latency bounds in AN2 that are adequate for most multimedia applications. A smaller frame size would provide lower CBR latency, but as already mentioned it would entail a larger granularity in bandwidth reservations. We are considering schemes in which a large frame is subdivided into smaller frames. This would allow each application to trade off a guarantee of lower latency against a smaller granularity of allocation.

To summarize, bandwidth and latency guarantees are provided through the following mechanisms:

- Applications request bandwidth reservations in terms of slots/frame.

- The network grants a request if it can find a path on which each link has the required capacity.
- Each switch, when notified of a new reservation, builds a schedule for transmitting cells across the switch.
- Enough buffers are permanently reserved for CBR traffic to ensure that arriving cells will always find an empty buffer.
- Latency is bounded by a simple function of link latency, path length, and frame size.

5 Statistical Matching

The AN2 switch combines the methods described in the previous two sections to provide low latency and high throughput for VBR traffic and guaranteed performance for CBR traffic. In this section, we present a generalization of parallel iterative matching, called *statistical matching*, that can efficiently support frequent changes of bandwidth allocation. In contrast, the Slepian-Duguid technique for bandwidth allocation works well so long as allocations are not changed too frequently, since changes require computing a new schedule at each switch. One motivation for dynamic bandwidth allocation is to provide fair sharing of network resources among competing flows of VBR traffic. Another is to support applications that require guaranteed performance and have bandwidth requirements that vary over time, as can be the case with compressed video.

Statistical matching works by systematically using randomness in choosing which request to grant and which grant to accept. We might say that parallel iterative matching uses fair dice in making random decisions; with statistical matching, the dice are weighted to divide bandwidth between competing flows according to their allocations. About 72% of the bandwidth can be reserved using our scheme; the remaining bandwidth can be filled in by normal parallel iterative matching. The first implementation of the AN2 switch does not implement statistical matching.

In this section, we first motivate statistical matching by briefly discussing network fairness, then we describe the statistical matching algorithm.

5.1 Motivation

Ramakrishnan et al. [1990] provide a formal definition of fairness in the allocation of network resources. To be fair, every user should receive an equal share of every network resource that does not have enough capacity to satisfy all user requests. If a user needs less than its equal share, the remainder should be split among the other users. One result of a fair network, then, is that users typically see graceful degradation in performance under increased load. Adding an additional user to an already crowded system will result in a relatively small decrease in everyone else's resource allocation.

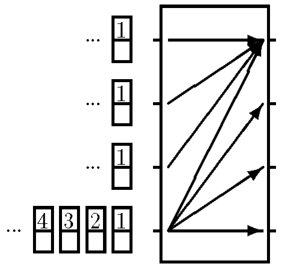

Figure 8: Unfairness with Parallel Iterative Matching

Unfortunately, an arbitrary topology network built out of switches using parallel iterative matching may not be fair, for two reasons. First, to be scheduled, a queued cell needs to receive a grant from its output and to have its input accept the grant. Both the input and output ports are sources of contention; parallel iterative matching will tend to give higher throughput to input-output connections that have fewer contending connections. In Figure 8, for instance, if input 4 chooses randomly which grant to accept, the connection from input 4 to output 1 will receive only one sixteenth of the link throughput; all other connections receive five times this bandwidth.

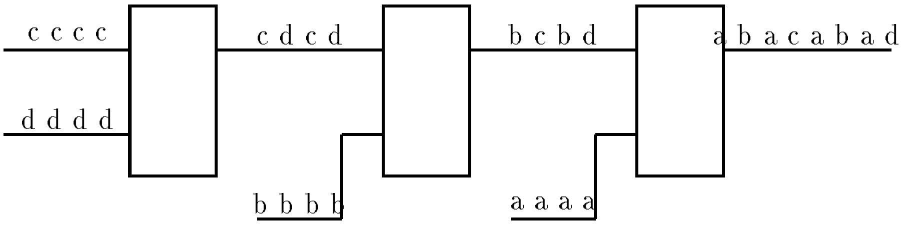

Figure 9: Unfairness with Arbitrary Topology Networks

Second, even if switches allocate output bandwidth equally among all requesting input ports, arbitrary topology networks using these switches may not share bandwidth fairly among users or flows [Demers et al. 89].[2] Depending on the workload and the topology of the network, each switch input may have a different number of flows. A flow reaching a bottleneck link at the end of a long chain of switches may receive an arbitrarily small portion of the link throughput, while another flow merging closer to the bottleneck receives a much larger portion. Unfortunately, this pattern is quite likely when one host is a highly-used server. Figure 9 illustrates

[2] A "network user" may, of course, be sending more than one flow of cells through a switch, for example, to different hosts. For simplicity, though, the remainder of our discussion will assume that our target is fairness among flows as an approximation to fairness among users.

what happens when four flows share a bottleneck link. Each letter represents a cell; switches are assumed to select input ports round robin. In a fair allocation, each flow would receive the same throughput on the rightmost link, but flows 'c' and 'd' receive much less throughput than does flow 'a'.

A number of approaches to fairness in arbitary topology networks have been proposed. One class of techniques involves using some measure of network load to determine a fair allocation of bandwidth among competing flows. Once such an allocation has been determined, the problem remains of dividing network resources according to the allocation. For example, Zhang [1991] suggests a *virtual clock* algorithm. Host network software assigns each flow a share of the network bandwidth and notifies each switch along the flow's path of the rate to be delivered to the flow. When a cell arrives at a switch, it is assigned a timestamp based on when it would be scheduled if the network were operating fairly; the switch gives priority to cells with earlier timestamps.

The virtual clock algorithm requires that each output link can select arbitrarily among any of the cells queued for it. This is the case in a switch with perfect output queueing. In our input-buffered switch, however, only one cell from each input can be forwarded at a time. Section 4 gave one way of supporting bandwidth allocation in an input-buffered switch. Statistical matching is another approach; one which is more suited to the rapid changes in allocation needed to provide fairness.

5.2 Algorithm

Statistical matching, like using a pre-computed frame schedule, delivers to each flow a specified portion of the link throughput. With statistical matching, up to $(1 - \frac{1}{e})(1 + \frac{1}{e^2})$, or 72%, of each link's throughput can be reserved; the throughput allocation can be in any pattern, provided the sum of the throughputs at any input or output is less than 72%. Any slot not used by statistical matching can be filled with other traffic by parallel iterative matching. However, adjusting throughput rates is more efficient with statistical matching than with a pre-computed schedule, because only the input and output ports used by a flow need be informed of a change in its rate.

Statistical matching is based on parallel iterative matching, but it makes more systematic use of randomness in making and accepting grants. The pairing of inputs to outputs is chosen independently for each time slot, but on average, each flow is scheduled according to its specified throughput rate. The algorithm mirrors parallel iterative matching except that there is no request phase.

We divide the allocatable bandwidth per link into X discrete units; $X_{i,j}$ denotes the number of units allocated to traffic from input i to output j. The key is that we arrange the random weighting factors at the inputs and outputs so that each input receives up to X *virtual grants*, each made independently with probability $\frac{1}{X}$. $X_{i,j}$ of the potential virtual grants to input i are associated with output j. If input i then chooses randomly among the virtual grants it receives, it will connect to each output with probability proportional to its reservation.

We outline the steps of the algorithm here, using the simplifying assumption that switch bandwidth is completely allocated. Appendix C presents a precise definition of the algorithm without this assumption, and shows that it delivers up to 72% of the link throughput.

1. Each output randomly chooses one input to grant; output j chooses input i with probability $\frac{X_{i,j}}{X}$ proportional to the bandwidth reservation.

2. If an input receives any grants, it chooses at most one grant to accept (it may accept none) in a two-step process:

   (a) The input reinterprets the grant as zero or more *virtual grants*, so that the resulting probability that input i receives k virtual grants from output j is just the binomial distribution – the likelihood that exactly k of X independent events occur, given that each occurs with probability $\frac{1}{X}$.

   (b) If an input receives any virtual grants, it chooses one randomly to accept; the output corresponding to the accepted virtual grant is then matched to the input.

Since each virtual grant is made with probability $\frac{1}{X}$, the likelihood that an input receives no virtual grants (and thus is not matched) by the above algorithm is $(\frac{X-1}{X})^X$. As X grows large, this approaches $\frac{1}{e}$ from below. Since each virtual grant is equally likely to be accepted, the probability of a connection between an input i and an output j is $\frac{X_{i,j}}{X}(1-\frac{1}{e})$, or about 63% of $\frac{X_{i,j}}{X}$.

Better throughput can be achieved by running a second iteration of statistical matching. The grant/accept steps are carried out independently of the results of the first iteration, but a match made by the second iteration is added only if both the input and output were left unmatched by the first iteration. Conflicting matches are discarded. We show in Appendix C that a match is added by the second iteration with probability (for large X) $\frac{1}{e^2}(1-\frac{1}{e})\frac{X_{i,j}}{X}$, yielding the ability to reserve a total 72% of the link bandwidth. Additional iterations yield insignificant throughput improvements.

Statistical matching requires more hardware to implement than does parallel iterative matching, although the cost is not prohibitive. Steps 1 and 2a can both be implemented as table lookups. The table is initialized with the number of entries for each outcome proportional to its probability; a random index into the table selects the outcome. Step 2b is a generalization of the random choice among requests needed by parallel iterative matching; similar implementation techniques apply.

5.3 Discussion

We motivated statistical matching by suggesting that it could be used to schedule the switch fairly among competing flows. Statistical matching meets many of the goals that motivated Zhang's virtual clock approach. With either approach, the switch can be set to assign equal throughput to every competing flow through a bottleneck link. Statistical matching can provide roughly equal throughput without the need for tagging individual cells with timestamps and prioritizing flows based on those timestamps, although some unfairness may be added when parallel iterative matching fills in gaps left by statistical matching. With statistical matching, as with the virtual clock approach, a flow can temporarily send cells faster or slower than its specified rate, provided the throughput is not exceeded over the long term. Queues in the network increase if the flow sends at a faster rate; queues empty as the flow sends at a slower rate. The virtual clock approach also provides a way of monitoring whether a flow is exceeding its specified rate over the long term; there is no analogue with statistical matching.

6 Summary

We have described the design of the AN2 switch, which can support high performance distributed computing. Key to the switch's operation is a technique called *parallel iterative matching*, a fast algorithm for choosing a conflict-free set of cells to forward across the switch during each time slot. Our prototype switch combines this with a mechanism to support real-time traffic even in the presence of clock drift. The switch will be used as the basic component of an arbitrary topology point-to-point local area network, providing

1. high bandwidth,
2. low latency for datagram traffic, so long as the network is not overloaded, and
3. bandwidth and latency guarantees for real-time traffic.

In addition, the switch's scheduling algorithm can be extended to allocate resources fairly when some part of the network is overloaded.

We believe that the availability of high performance networks with these characteristics will enable a new class of distributed applications. Networks are no longer slow, serial, highly error-prone bottlenecks where message traffic must be carefully minimized in order to get good performance. This enables distributed systems to be more closely coupled than has been possible in the past.

Acknowledgements

We would like to thank Mike Burrows, Hans Eberle, Mike Goguen, Domenico Ferrari, Butler Lampson, Tony Lauck, Hal Murray, Roger Needham, John Ousterhout, Tom Rodeheffer, Ed Satterthwaite, and Mike Schroeder for their helpful comments.

References

[Ahmadi & Denzel 89] Ahmadi, H. and Denzel, W. A Survey of Modern High-Performance Switching Techniques. *IEEE Journal on Selected Areas in Communications*, 7(7):1091–1103, September 1989.

[Ame 87] American National Standards Institute, Inc. *Fiber distributed data interface (FDDI). Token ring media access control (MAC). ANSI Standard X3.139*, 1987.

[Ame 88] American National Standards Institute, Inc. *Fiber distributed data interface (FDDI). Token ring physical layer protocol (PHY). ANSI Standard X3.148*, 1988.

[Batcher 68] Batcher, K. Sorting Networks and their Applications. In *AFIPS Conference Proc.*, pages 307–314, 1968.

[Demers et al. 89] Demers, A., Keshav, S., and Shenker, S. Analysis and Simulation of a Fair Queueing Algorithm. In *Proc. ACM SIGCOMM '89 Conference on Communications Architectures and Protocols*, pages 1–12, September 1989.

[Ferrari & Verma 90] Ferrari, D. and Verma, D. A Scheme for Real-Time Channel Establishment in Wide-Area Networks. *IEEE Journal on Selected Areas in Communications*, 8(3):361–379, April 1990.

[Giacopelli et al. 91] Giacopelli, J., Hickey, J., Marcus, W., Sincoskie, W., and Littlewood, M. Sunshine: A High-Performance Self-Routing Broadband Packet Switch Architecture. *IEEE Journal on Selected Areas in Communications*, 9(8):1289–1298, October 1991.

[Golestani 90] Golestani, S. Congestion-Free Transmission of Real-Time Traffic in Packet Networks. In *Proc. INFOCOM '90*, pages 527–542, June 1990.

[Huang & Knauer 84] Huang, A. and Knauer, S. Starlite: A Wideband Digital Switch. In *Proc. GLOBECOM '84*, pages 121–125, December 1984.

[Hui & Arthurs 87] Hui, J. and Arthurs, E. A Broadband Packet Switch for Integrated Transport. *IEEE Journal on Selected Areas in Communications*, 5(8):1264–1273, October 1987.

[Hui 90] Hui, J. *Switching and Traffic Theory for Integrated Broadband Networks*. Kluwer Academic Press, 1990.

[Jain 90] Jain, R. Congestion Control in Computer Networks: Issues and Trends. *IEEE Network Magazine*, pages 24–30, May 1990.

[Kalmanek et al. 90] Kalmanek, C., Kanakia, H., and Keshav, S. Rate Controlled Servers for Very High-Speed Networks. In *Proc. IEEE Global Telecommunications Conference*, pages 300.3.1–300.3.9, December 1990.

[Karol et al. 87] Karol, M., Hluchyj, M., and Morgan, S. Input Versus Output Queueing on a Space-Division Packet Switch. *IEEE Transactions on Communications*, 35(12):1347–1356, December 1987.

[Karol et al. 92] Karol, M., Eng, K., and Obara, H. Improving the Performance of Input-Queued ATM Packet Switches. In *Proc. INFOCOM '92*, pages 110–115, May 1992.

[Karp et al. 90] Karp, R., Vazirani, U., and Vazirani, V. An Optimal Algorithm for On-line Bipartite Matching. In *Proc. 22nd Annual ACM Symposium on Theory of Computing*, pages 352–358, May 1990.

[Kermani & Kleinrock 79] Kermani, P. and Kleinrock, L. Virtual Cut-through: A New Computer Communication Switching Technique . *Computer Networks*, 3:267–286, September 1979.

[Li 88] Li, S.-Y. Theory of Periodic Contention and Its Application to Packet Switching. In *Proc. INFOCOM '88*, pages 320–325, March 1988.

[Metcalfe & Boggs 76] Metcalfe, R. and Boggs, D. Ethernet: Distributed Packet Switching for Local Computer Networks. *Communications of the ACM*, 19(7):395–404, July 1976.

[Obara & Yasushi 89] Obara, H. and Yasushi, T. An Efficient Contention Resolution Algorithm for Input Queueing ATM Cross-Connect Switches. *International Journal of Digital and Analog Cabled Systems*, 2(4):261–267, October 1989.

[Owicki & Karlin 92] Owicki, S. and Karlin, A. Factors in the Performance of the AN1 Computer Network. In *Proc. 1992 ACM SIGMETRICS and PERFORMANCE '92 Conference on Measurement and Modeling of Computer Systems*, pages 167–180, June 1992.

[Patel 79] Patel, J. Processor-Memory Interconnections for Multiprocessors. In *Proc. 6th Annual Symposium on Computer Architecture*, pages 168–177, April 1979.

[Ramakrishnan & Jain 90] Ramakrishnan, K. and Jain, R. A Binary Feedback Scheme for Congestion Avoidance in Computer Networks. *ACM Transactions on Computer Systems*, 8(2):158–181, May 1990.

[Schroeder & Burrows 90] Schroeder, M. and Burrows, M. Performance of Firefly RPC. *ACM Transactions on Computer Systems*, 8(1):1–17, February 1990.

[Schroeder et al. 91] Schroeder, M., Birrell, A., Burrows, M., Murray, H., Needham, R., Rodeheffer, T., Satterthwaite, E., and Thacker, C. Autonet: A High-Speed Self-Configuring Local Area Network Using Point-to-Point Links. *IEEE Journal on Selected Areas in Communications*, 9(8):1318–1335, October 1991.

[Tamir & Frazier 88] Tamir, Y. and Frazier, G. High-Performance Multi-Queue Buffers for VLSI Communication Switches. In *Proc. 15th Annual Symposium on Computer Architecture*, pages 343–354, June 1988.

[Tarjan 83] Tarjan, R. *Data Structures and Network Algorithms.* SIAM, 1983.

[Xil 91] Xilinx, Inc. *Xilinx: The Programmable Gate Array Data Book*, 1991.

[Yeh et al. 87] Yeh, Y., Hluchyj, M., and Acampora, A. The Knockout Switch: A Simple Modular Architecture for High-Performance Switching. *IEEE Journal on Selected Areas in Communications*, 5(8):1274–1283, October 1987.

[Zhang & Keshav 91] Zhang, H. and Keshav, S. Comparison of Rate-Based Service Disciplines. In *Proc. ACM SIGCOMM '91 Conference on Communications Architectures and Protocols*, pages 113–122, September 1991.

[Zhang 91] Zhang, L. Virtual Clock: A New Traffic Control Algorithm for Packet Switching Networks. *ACM Transactions on Computer Systems*, 9(2):101–124, May 1991.

A Parallel Iterative Matching: Number of Iterations

In this appendix, we show that the parallel iterative matching algorithm described in Section 3 reaches a maximal match in an average of $O(\log N)$ iterations for an N by N switch. This bound is independent of the pattern of requests. The key to the proof is to observe that if an unmatched output receives a request, one iteration of parallel iterative matching will usually either (i) match the output to one of its requesting inputs or (ii) match most of the inputs requesting that output to other outputs. The result is that each iteration reduces the number of unresolved requests by an average of at least 3/4. A request is unresolved if both its input and its output port remain unmatched.

Consider the requests to each output separately. Suppose an output Q receives requests from n inputs during some iteration. Of these n inputs, some fraction will request and receive a grant from some output besides Q, and the rest will receive no grants from other outputs. Let k be the number of inputs requesting Q that receive no other grants.

Q randomly chooses one of its n requests to grant. Since Q chooses among the requesting inputs with equal probability, and since Q's choice is independent of the choices made by other outputs, the probability that Q will grant to an input that has a no competing grant from another output is k/n. In this case, Q's grant will be accepted, and as a result, *all* of the n requests to Q will be resolved – one will be accepted, while the rest will never be accepted.

On the other hand, with probability $1-(k/n)$, Q will grant to an input that also receives a grant from some other output. If the input picks Q's grant to accept, all of the requests to Q will be resolved. But even if Q's grant is not accepted, all of the $n-k$ inputs that received a grant will be matched (to some other output) during this iteration; none of their $n-k$ requests to Q will remain on the next iteration.

Thus, with probability k/n all requests to Q are resolved, and with probability $1-(k/n)$ at most k remain unresolved. As a result, the average number of unresolved requests to Q is at most $(1-(k/n)) \times k$, which is no greater than $n/4$ for any k. Since we start with at most N^2 requests, this implies that the expected number of unresolved requests after i iterations is at most $N^2/4^i$.

It remains to be shown that the algorithm reaches a maximal match in an average of $O(\log N)$ steps. Let C be the step on which the last request is resolved. Then the expected value of C is:

$$E[C] = \sum_{i=1}^{\infty} i \ \Pr\{C = i\}$$

Re-writing the sum yields:

$$E[C] = \sum_{i=0}^{\infty} \Pr\{C > i\} \tag{1}$$

The likelihood that $C > i$ is just the likelihood that at least one match remains unresolved at the end of i iterations:

$$\Pr\{C > i\} = \sum_{j=1}^{\infty} \Pr\{j \text{ requests remain after } i \text{ iterations}\}$$

Replacing the sum by an obviously larger one:

$$\Pr\{C > i\} \leq \sum_{j=1}^{\infty} j \Pr\{j \text{ requests remain after } i \text{ iterations}\} \leq \frac{N^2}{4^i}$$

Here the final inequality comes from the previously derived bound on the average number of unresolved matches after i iterations.

Since a probability can never be greater than 1,

$$\Pr\{C > i\} \leq \min(1, N^2/4^i)$$

. Substituting into Formula 1 yields:

$$E[C] \leq \sum_{i=0}^{\infty} \min(1, \frac{N^2}{4^i})$$

Since $N^2 = 4^i$ when $\log_2 N = i$, the sum has no more than $\log_2 N$ terms with the value 1, and the remainder of the sum is a power series bounded by $4/3$. Thus:

$$E[C] \leq \log_2 N + \frac{4}{3}$$

B Latency and Buffer Space Bounds for CBR Traffic

In this appendix, we show that we can provide end-to-end guaranteed performance for constant bit rate flows, even if the clocks in the network switches and controllers are known only to run at approximately the same rate, within some tolerance. As discussed in Section 4, a frame schedule is pre-computed at each switch, assigning a flow's cells to a fixed number of frame slots. Because the frame rate depends on the clock rate in each switch, the bandwidth delivered to a flow varies slightly (and unpredictably) at each switch in the flow's path. We address this by adding extra empty slots to each controller (but not switch) frame, to constrain the controller frame rate to be slower than the frame rate of the slowest possible downstream switch. Using this constraint and some natural ground rules for controller and switch operation, we can demonstrate bounds on both a flow's end-to-end latency and its buffer space requirements. A flow's end-to-end throughput is bounded by its rate on the slowest possible controller. Because each flow has its own reserved buffer space and bandwidth, the behavior of each flow is independent of the behavior of other flows; our discussion focuses on a single flow at a time.

Table 3 summarizes a number of terms used in our proof. The minimum and maximum frame times are in terms of real "wall-clock" time; that is, the nominal time for one slot $\times$ the number of slots per frame $\times$ the maximum or minimum possible clock rate. Note that $F_{c-min} > F_{s-max}$: the frame of the fastest controller is constrained to be longer than the frame of the slowest switch. The link latency l is the maximum wall clock time from when a cell departs one switch to when it is first eligible to be forwarded at the next switch, including any processing overhead at the switch. Finally, the adjusted latency, $L(c_i, s_n)$, is the wall clock time from the end of the frame in which cell c_i departs the controller s_0 to the end of the frame in which the cell departs switch s_n. We use the adjusted latency instead of the true latency because it is independent of which slots within a frame are allocated to a particular flow.

We temporarily make the simplifying assumption that each flow reserves only a single cell per frame; we remove this assumption later in this section. Each controller and switch obeys the reservation – each forwards at most one cell per frame for each flow. Further, switches forward cells in FIFO order, with no needless delays – if a cell has arrived at a switch and is eligible for forwarding at the beginning of a frame, then either that cell or an earlier (queued) cell from the same flow is forwarded during the frame.

Symbol	Definition
F_{s-min}, F_{s-max}	minimum, maximum time for a switch frame
F_{c-min}, F_{c-max}	minimum, maximum time for a controller frame
l	maximum link latency and switch overhead
p	a flow's path length (number of hops)
c_i	the i'th cell transmitted in a flow
s_n	n'th switch in a flow's path, $0 <= n <= p$
$T(c_i, s_n)$	time at the end of the frame in which cell c_i departs switch s_n
$L(c_i, s_n)$	adjusted latency, $T(c_i, s_n) - T(c_i, s_0)$

Table 3: Symbol Definitions

B.1 Bounded Latency

The key observation to bounding the end-to-end latency is that if two cells, c_i and c_{i+1}, depart a switch s_n in consecutive frames, then the adjusted latency of c_{i+1} is less than that of c_i. This is because c_i and c_{i+1} must depart the controller in separate frames, and frames take longer at the controller s_0 than at any switch s_n.

$$T(c_{i+1}, s_n) - T(c_i, s_n) <= F_{s-max} < F_{c-min} <= T(c_{i+1}, s_0) - T(c_i, s_0)$$

$$T(c_{i+1}, s_n) - T(c_{i+1}, s_0) < T(c_i, s_n) - T(c_i, s_0)$$

$$L(c_{i+1}, s_n) < L(c_i, s_n) \qquad (2)$$

Note that the queueing delay that cell c_{i+1} experiences at switch s_n may well be longer than that of the previous cell c_i, but c_{i+1}'s *end-to-end* adjusted latency will be shorter than c_i's.

We define an *active* frame to be one in which a cell is forwarded to the next switch. Because switch frames occur more frequently than controller frames, at each switch there will be sequences of active frames interspersed with inactive frames (when there is no cell available to be forwarded). The consequence of Formula 2 is that the worst case adjusted latency at a switch s_n is experienced by some cell c_i that is sent in the *first* in a sequence of active frames – that is, the cell must be sent in a frame immediately after a frame when the switch had nothing to forward. Because we assume the switch does not needlessly delay cells, c_i must have arrived at switch s_n after the previous (inactive) frame started, in other words, no more than two frames before c_i departed switch s_n. The cell must have departed the upstream switch s_{n-1} no earlier than $T(c_i, s_n) - (2F_{s-max} + l)$. Since c_i's adjusted latency in departing from the upstream switch s_{n-1} is likewise bounded by some first cell in a sequence of active frames, by induction we have:

$$L(c_i, s_p) <= 2p(F_{s-max} + l) \qquad (3)$$

B.2 Bounded Buffer Space Requirements

We next derive bounds for the buffer space required at each switch. Clearly, a bound must exist because end-to-end latency is bounded; in this sub-section, we develop a precise formula for the bound.

First, observe that there is a bound on the maximum number of consecutive active frames. Formula 2 implies that with each successive active frame, adjusted latency decreases by at least $F_{c-min} - F_{s-max}$. But Formula 3 implies that there is also a maximum adjusted latency. The minimum adjusted latency is $-F_{c-max}$; this is negative because of the definition of adjusted latency – a cell can depart the first switch s_1 in a frame that finishes before the controller s_0's frame does. Thus, the maximum sequence of active frames is:

$$1 + \left\lfloor \frac{(2F_{s-max} + l)p + F_{c-max}}{F_{c-min} - F_{s-max}} \right\rfloor$$

Since the frames immediately before and after this sequence are inactive, there could not be a cell queued at the beginning of the frame before the first active frame, nor at the end of the last active frame in the sequence[3]. This means that the

[3] We consider only the buffer space needed by queued cells that are eligible for forwarding; additional (implementation-dependent) buffer space may be needed by cells that are in the process of arriving at the switch.

maximum length of time that a switch can continuously have a cell queued is:

$$F_{s-max}\left(2+\left\lfloor\frac{(2F_{s-max}+l)p+F_{c-max}}{F_{c-min}-F_{s-max}}\right\rfloor\right) \tag{4}$$

During any period of time t, the maximum number of cells that could arrive at a switch is $2+\lfloor\frac{t}{F_{s-min}}\rfloor$. Two cells can depart the upstream switch, one at the end of a frame and the other at the beginning of the next frame, both arriving at the beginning of the time period. From then on, the arrival rate is limited by the fastest possible switch frame rate. Analagously, the minimum number of cells that must depart the switch during an interval t in which there are queued cells is $\lfloor\frac{t}{F_{s-max}}\rfloor-1$.

The buffer space needed at a switch can be bounded by the difference in the maximum arrival rate and the minimum departure rate, over the maximum interval for which queued cells can be present. Substituting that interval (Formula 4) for t in the arrival and departure rates derived above, the buffer space required is no more than:

$$4+\frac{F_{s-max}-F_{s-min}}{F_{s-min}}\left(2+\frac{(2F_{s-max}+l)p+F_{c-max}}{F_{c-min}-F_{s-max}}\right) \tag{5}$$

The above results were derived assuming that each flow reserved only a single cell per frame. For a flow of k cells per frame, we must change the rules on switch and controller operation in the obvious way – no switch or controller forwards more than k cells of the flow in the same frame, cells are forwarded in FIFO order, and if a cell arrives at a switch before the beginning of a frame, it is either forwarded in the frame, or k previous cells of the flow are forwarded. If we consider (for purposes of analysis) a flow of k cells per frame to be partitioned into k classes, with cell c_i assigned to class $(i \bmod k)$, the cells of a single class will be treated (under these rules of operation) as if they belonged to a flow with one cell per frame.

Thus the buffer space required for a CBR flow is a constant factor times the number of reserved cells per frame, and buffer space required for all flows is $(4+c)$ times the frame size, where c is governed by Formula 5. The value of c is determined by network parameters: clock skew, link and switch delay, network diameter, and the difference between controller and switch frame size. For many common local area network configurations, c is small; it can be made arbitrarily small by increasing controller frame size, at some cost in reduced throughput.

C Statistical Matching Throughput

In this appendix, we describe the statistical matching algorithm more completely and show that it allows up to $(1-\frac{1}{e})(1+\frac{1}{e^2})\approx 0.72$ of the switch throughput to be allocated in any arbitrary pattern. Recall from Section 5.2 that we divide the allocatable bandwidth per link into X discrete units; $X_{i,j}$ denotes the number of units allocated to traffic from input i to output j. We assume temporarily that the switch bandwidth is completely allocated; we will remove this assumption shortly.

The algorithm is as follows:

1. Each output j randomly chooses an input i to grant to, with probability proportional to its reservation:

$$\Pr\{j \text{ grants to } i\} = \frac{X_{i,j}}{X}$$

2. Each input chooses at most one grant to accept (it may accept none) in a two-step process:

   (a) Each input i reinterprets each grant it receives as a random number $m_{i,j}$ of *virtual grants*, chosen between 0 and $X_{i,j}$ according to the probability distribution:

$$\Pr\{m_{i,j} = m, 0 < m \leq X_{i,j}\} = \binom{X_{i,j}}{m} \times \left(\frac{1}{X}\right)^m \times \left(\frac{X-1}{X}\right)^{X_{i,j}-m} \times \frac{X}{X_{i,j}}$$

$$\Pr\{m_{i,j} = 0\} = 1 - \Pr\{1 \leq m_{i,j} \leq X_{i,j}\}$$

   When j does not grant to i, $m_{i,j}$ is set to zero.

   (b) If an input receives any virtual grants, the input chooses one randomly to accept. In other words, the input chooses among granting outputs with probability proportional to the number of virtual grants from each output:

$$\Pr\{i \text{ accepts } j\} = \frac{m_{i,j}}{\sum_k m_{i,k}}$$

   If a grant is accepted, the input randomly chooses among the flows for the connection according to their bandwidth reservations.

The key to the algorithm is that each input i receives the same number of virtual grants from an output j that it would receive had each of the virtual grants been made with probability $\frac{1}{X}$ by an independent output. To see this, note that the probability that exactly m of $X_{i,j}$ events occur, given that each occurs with probability $\frac{1}{X}$, has the binomial distribution:

$$\binom{X_{i,j}}{m} \times \left(\frac{1}{X}\right)^m \times \left(\frac{X-1}{X}\right)^{X_{i,j}-m} \tag{6}$$

Of course, an input i can receive a virtual grant from output j only if j sends a physical grant to i in step 1:

$$\begin{aligned}&\Pr\{m_{i,j} = m, m > 0\} = \\ &\quad \Pr\{j \text{ grants to } i\} \times \Pr\{i \text{ chooses } m_{i,j} = m, m > 0 | j \text{ grants to } i\}\end{aligned} \tag{7}$$

Substituting in Formula 7 with the probabilities from steps 1 and 2a in the algorithm, we see that the probability that input i chooses $m_{i,j} = m$ is exactly the binomial distribution from Formula 6, for $m > 0$. Since the probabilities in both cases must sum to one, it follows that the probability that input i chooses $m_{i,j} = 0$ is also as specified by the binomial distribution.

If an input receives any virtual grants, it randomly chooses one among them to accept. By the argument above, the input will recieve no virtual grant from any output with probability $(\frac{X-1}{X})^X$. Otherwise, the input will match some output, and because each virtual grant is made and accepted with equal likelihood, each output is matched with probability proportional to its reservation:

$$\begin{aligned}
\Pr\{i \text{ matches } j\} &= \frac{X_{i,j}}{X} \times \left(1 - \left(\frac{X-1}{X}\right)^X\right) \\
&= \frac{X_{i,j}}{X} \times \Pr\{i \text{ matches}\} \\
&= \frac{X_{i,j}}{X} \times \Pr\{j \text{ matches}\}
\end{aligned}$$

As X becomes large, $(1 - (\frac{X-1}{X})^X)$ approaches $1 - \frac{1}{e} \approx 0.63$ from above.

This result implies a rather surprising fact: the probability that a given output matches is independent of the input to which it grants. For

$$\begin{aligned}
\frac{X_{i,j}}{X} \times \Pr\{j \text{ matches}\} &= \Pr\{i \text{ matches } j\} \\
&= \Pr\{i \text{ matches } j | j \text{ grants to } i\} \times \Pr\{j \text{ grants to } i\} \\
&= \Pr\{i \text{ matches } j | j \text{ grants to } i\} \times \frac{X_{i,j}}{X}
\end{aligned}$$

or

$$\Pr\{j \text{ matches}\} = \Pr\{j \text{ matches} | j \text{ grants to } i\}.$$

This fact is useful in analyzing the effect of running a second iteration of statistical matching. The second iteration is run independently of the first. If input i and output j are matched on the second round, a connection between them is made provided that neither was matched on the first round.

$$\begin{aligned}
&\Pr\{i \text{ matches } j \text{ in two rounds}\} = \\
&\quad \Pr\{i \text{ matches } j \text{ in round } 1\} + \\
&\quad \Pr\{i \text{ matches } j \text{ in round 2 and neither matches in round } 1\}
\end{aligned}$$

Now, matches in the two rounds are independent and equally likely, Moreover, the events "i unmatched on the first round" and "j unmatched on the first round" are either independent or positively correlated. Consider the probabilities of i and/or j being matched conditional on each possible recipient of j's grant. If j grants to

i, then it is impossible for j to be matched while i is unmatched, so "i unmatched" and "j unmatched" cannot have negative correlation. Now suppose j grants to some other input $h \neq i$, and there is no output k such that $X_{h,k}$ and $X_{i,k}$ are both positive. Then the events "i unmatched" and "j unmatched" are independent, because no other choice made in the algorithm effects both events. Finally, suppose j grants to $h \neq i$, and there is an output k such that $X_{h,k}$ and $X_{i,k}$ are both positive. Then the potential matching of h to k conflicts both with the matching of i to k and that of h to j, inducing a positive correlation between the events "i unmatched" and "j unmatched." We have now established that

$$\begin{aligned}&\forall x(\Pr\{i \text{ and } j \text{ unmatched} | j \text{ grants to } x\} \geq \\ &\quad \Pr\{i \text{ unmatched} | j \text{ grants to } x\} \times \Pr\{j \text{ unmatched} | j \text{ grants to } x\})\end{aligned}$$

Using the previous result that the probability of j matching is independent of the input to which it grants, and summing over all inputs x we have

$$\Pr\{i \text{ and } j \text{ unmatched}\} \geq \Pr\{i \text{ unmatched}\} \times \Pr\{j \text{ unmatched}\}$$

Finally, we can conclude

$$\begin{aligned}&\Pr\{i \text{ matches } j \text{ in two rounds }\} \\ &\quad \geq \Pr\{i \text{ matches } j \text{ in round 1}\} + \\ &\qquad \Pr\{i \text{ matches } j \text{ in round 2}\} \times \Pr\{i \text{ unmatched in round 1}\} \times \\ &\qquad \Pr\{j \text{ unmatched in round 1}\} \\ &\quad \geq \frac{X_{i,j}}{X} \times \left(1 - \frac{1}{e}\right) \times \left(1 + \frac{1}{e^2}\right)\end{aligned}$$

The last step in the analysis is to consider what happens when the switch is not fully reserved. On each round, an input i (or output j) with less than a full reservation can simulate being fully reserved by assigning the unreserved bandwidth, denoted by $X_{i,0}$ (resp., $X_{0,j}$) to an imaginary output (input). If output j is less than fully reserved, it simulates granting to its imaginary input (i.e., sends no grant to any real input) with probability $X_{0,j}/X$. Similarly, an input i that is less than fully reserved randomly chooses a number $m_{i,0}$ of virtual grants from its imaginary output, using the probability distribution:

$$\Pr\{m_{i,0} = m, 0 \leq m \leq X_{i,0}\} = \binom{X_{i,j}}{m} \times \left(\frac{1}{X}\right)^m \times \left(\frac{X-1}{X}\right)^{X_{i,j}-m}$$

The input accepts such grants (by rejecting grants from real outputs) in proportion to their number, just as for grants from real outputs. When a second round match conflicts with a first round match to an imaginary input or output, it is not necessary to discard the second round match. Retaining it can only increase the throughput derived above.