Fast Switched Backplane for a Gigabit Switched Router

Nick McKeown
Department of Electrical Engineering
Stanford University, Stanford, CA 94305-9030

1 Table of Contents

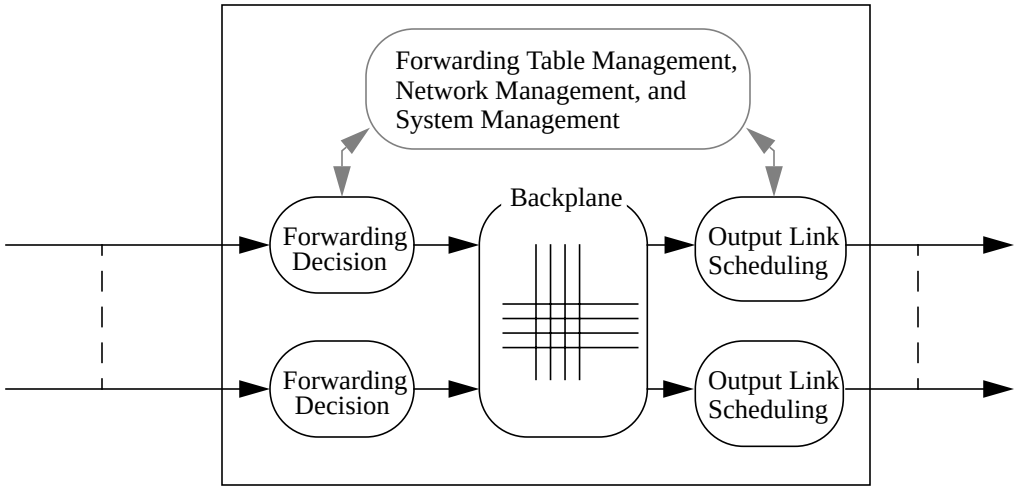


Figure 1: The key common architectural components of a router. In high performance systems, the forwarding decision, backplane and output link scheduling must be performed in hardware, while the less timely management and maintenance functions are performed in software.

2 Abstract

There is a new trend in the architecture of high performance Internet routers: congested, *shared* backplanes are being replaced by much faster *switched* backplanes that allow multiple packets to be transferred simultaneously.

In this paper, we'll see why the demand for router performance means that switched backplanes are needed now, and study the technical problems that must be solved in their design. In particular, we focus on the switched backplane developed for the Cisco 12000 GSR. This router has a high-performance switched backplane architecture capable of switching 16-ports simultaneously, each with a line rate of 2.4Gb/s. The backplane uses a number of new technologies that enable a parallel, compact design providing extremely high throughput for both unicast and multicast traffic. Integrated support for priorities on the backplane allows the router to provide distinguished qualities of service for multimedia applications.

3 The Architecture of Internet Routers

We begin by revisiting the general architecture of an IP router, as illustrated in a simplified form in Figure 1. We find it convenient to separate the router's functions into two types:

1. **Datapath Functions:** operations that are performed on every datagram that passes through the router. These are most often implemented in special purpose hardware, and include the forwarding decision, the backplane and output link scheduling.
2. **Control Functions:** operations that are performed relatively infrequently. These are invariably implemented in software, and include, the exchange of routing table information with neighboring routers, as well as system configuration and management.

Therefore, when trying to improve the per-packet performance of a router, we naturally focus on the *datapath* functions. Let's take a closer look at the datapath functions by tracing the typical path of a packet through an IP router:

The Forwarding Decision: When a datagram arrives, its destination address is looked up in a forwarding table. If the address is found, a next-hop MAC address is appended to the front of the datagram, the time-to-live (TTL) field of the IP datagram header is decremented, and a new header checksum is calculated.

The Backplane: The datagram is then forwarded across the backplane to its outgoing port (or ports). While waiting its turn to be transferred across the backplane, the datagram may need to be queued: if insufficient queue space exists, the datagram may need to be dropped, or it may be used to replace other datagrams.

The Output-link Scheduler: When it reaches the outgoing port, the datagram waits for its turn to be transmitted on the output link. In most routers today, the outgoing port maintains a single first-come-first-served queue, and transmits datagrams in the order that they arrive. More advanced routers distinguish datagrams into different flows, or priority classes, and carefully schedule the departure time of each datagram in order to meet specific quality of service guarantees.

Over the years, there have been many different architectures used for routers. Particular architectures have been selected based on a number of factors, including cost, number of ports, required performance and currently available technology. The detailed implementation of individual commercial routers have generally remained proprietary, but in broad terms, all routers have evolved in similar ways. They have evolved similarly over *time*: as time has progressed, they have followed a common trend of development. They have also evolved in *performance*: at any one time, the higher performance routers differ from the lower-performance devices in similar ways. The first trend in their evolution has been the implementation of more and more of the datapath functions in hardware. In recent years, improvements in the integration of CMOS technology has made it possible to implement a larger number of functions in ASIC components, moving functions that were once per-

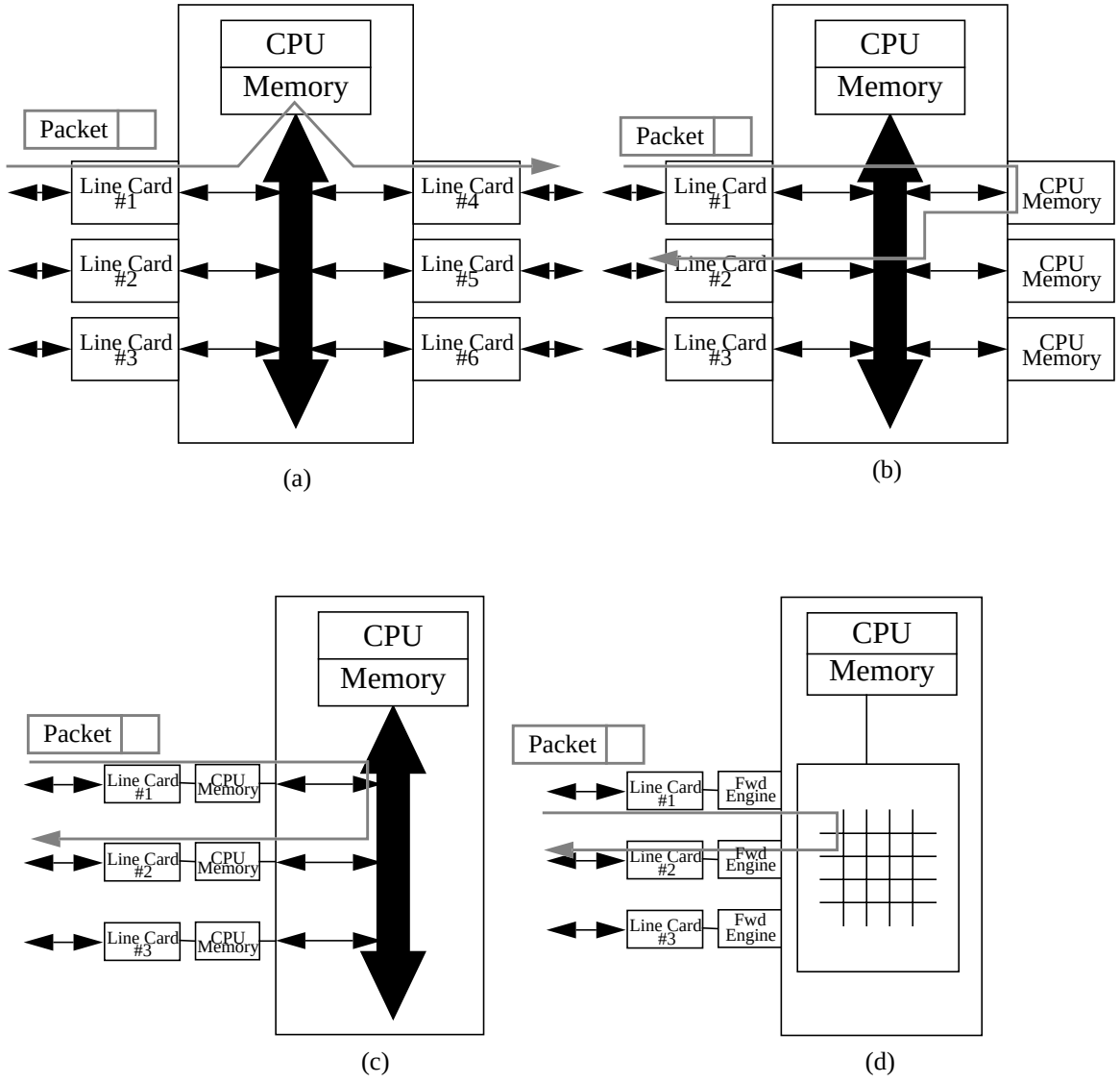


Figure 2: The basic architectures of packet-processors. Over time, or to enhance performance, more hardware is used on the main datapath; and more parallelism is employed.

formed in software to special purpose hardware. Increased integration serves two purposes: on one hand, the same functions can often be performed at similar speed for a lower cost. On the other hand, with sufficient hardware it is often possible to significantly improve the system performance. The second evolutionary trend has been towards parallelism to either achieve higher performance, or to allow the use of lower cost components. Third, and most important to our discussion here, there has been a trend away from the use of shared buses. Buses that are shared between multiple functions often become congested, limiting the performance of the system.

The evolution in the architecture of routers is illustrated in Figure 2(a)-(d). The original routers were built around a conventional computer architecture, as shown in Figure 2(a): a shared central

bus, with a central CPU, memory and peripheral Line Cards. Each Line Card performs the MAC layer function, connecting the system to each of the external links. Packets arriving from a link are transferred across the shared bus to the CPU, where a forwarding decision is made. The packet is then transferred across the bus again to its outgoing Line Card, and onto the external link.

The main limitation of the architecture in Figure 2(a) is that the central CPU must process every packet, ultimately limiting the throughput of the system. This limitation prompted the architecture in Figure 2(b), in which multiple CPUs process packets in parallel. Incoming packets are forwarded to a CPU as before, but this time there is a choice. For example, the packet could be sent to the first available CPU, or packets with the same detination address could always be sent to the same CPU. The advantage is clear: parallelism can increase the system throughput, or allow the use of lower cost CPUs.

The architecture in Figure 2(c) takes parallelism one stage further, placing a separate CPU at each interface. A local forwarding decision is made in a dedicated CPU, and the packet is immediately forwarded to its outgoing interface. This has the additional benefit that each packet need only traverse the bus once, thus increasing the system throughput. The central CPU is needed to maintain the forwarding tables in each of the other CPUs, and for centralized system management functions.

The performance of the architecture in Figure 2(c) is ultimately limited by two factors. First, forwarding decisions are made in software, and so are limited by the speed of a general purpose CPU. But general purpose CPUs are not well suited to applications in which the data (packets) flow through the system; CPUs are better suited to applications in which data is examined multiple times, thus allowing the efficient use of a cache. Carefully designed, special purpose ASICs can readily outperform a CPU when making forwarding decisions, managing queues, and arbitrating access to the bus. Hence, CPUs are being replaced increasingly by specialized ASICs. The second factor that limits the performance is the use of a shared bus — only one packet may traverse the bus at a time between two Line Cards. Performance can be increased if multiple packets can be transferred across the bus simultaneously. This is the reason that a crossbar switch is increasingly used in place of a shared bus. In a crossbar switch, multiple Line Cards can communicate with each other simultaneously greatly increasing the system throughput.

So, by introducing a hardware forwarding engine and replacing the bus with a crossbar switch, we reach the architecture shown in Figure 2(d). Today, the very highest performance Routers are designed acccording to this architecture.

Our work is motivated by the following configuration, and so we will use it as an example throughput this paper: a 16-port IP router, with each port operating at 2.4Gb/s. Let’s consider how

fast each of the datapath functions must operate in our example router, starting with the *forwarding decision*. A new forwarding decision must be made for every arriving datagram, and so with an average datagram length today of 250bytes, each port of our router must make a forwarding decision every 800ns. This corresponds to approximately 1.2million decisions per second. Although performing a router lookup in such a short time may sound like a difficult goal to achieve, in practice, it is quite stratighforward. In fact, it is a common misconception that routers today — operating at much lower data rates than our example — are limited by their ability to forward packets. This is not so. Techniques for making over a million forwarding decsions per second have been known for sometime. And more recently, a variety of techniques operating at 5-20 million forwarding decisions per second have been described [2][1][18][19][20].

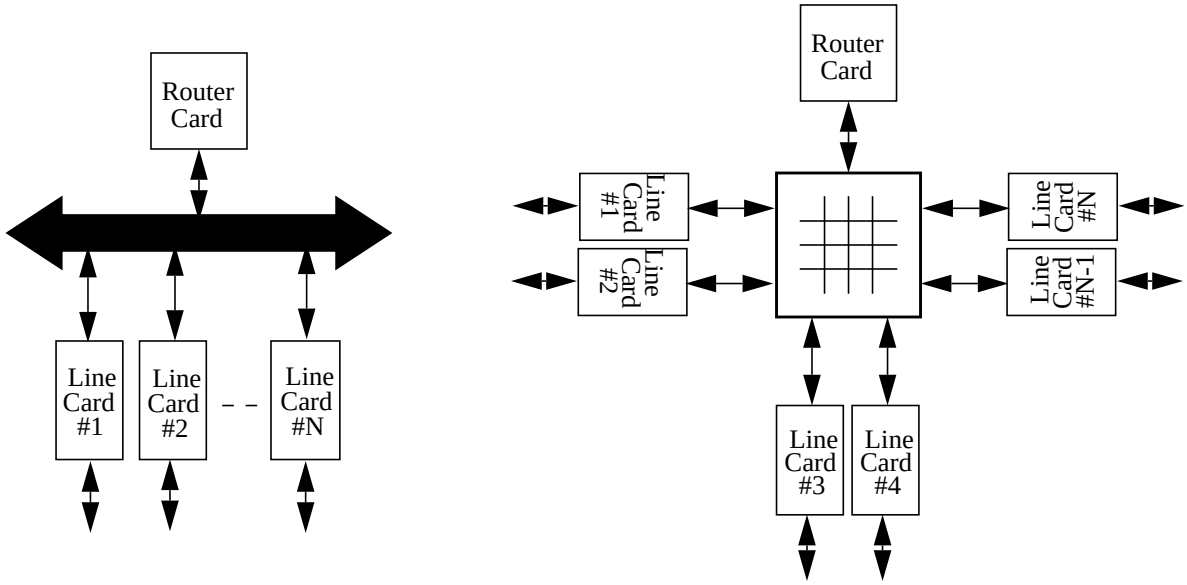
Now let's take a look at the performance we need for our router's backplane. In the worst case, all of the line cards may wish to transfer datagrams simultaneously, requiring a backplane with an aggregate bandwidth of $16 \times 2.4\text{Gb/s} = 38.4\text{Gb/s}$. If the backplane cannot keep up, the shared bus becomes congested and buffers overflow. As we will see, it is impractical today to build shared backplanes operating at 38.4Gb/s. But fortunately, switched backplanes provide a simple, and efficient alternative, and can easily operate at these rates.

The Cisco 12000 router uses the architecture in Figure 2(d) allowing multiple packets to be transferred simultaneously.

4 Why we need fast backplanes

If backplanes are congested, why don't we just make them faster? After all, the computer industry has introduced faster and faster buses over the past few years: from ISA, to EISA, and now PCI. Unfortunately, while the need for routing bandwidth is growing exponentially, bus bandwidth is growing much more slowly.

When the highest performance is required, shared backplanes are impractical for two reasons. First, is congestion: the bandwidth of the bus is shared among all the ports, contention leads to additional delay for packets. If the arrival rate of packets exceeds the bus bandwidth for a sustained period, buffers will overflow and data will be lost. Second, high speed shared busses are difficult to design. The electrical loading caused by multiple ports on a shared bus, the number of connectors that a signal encounters, and reflections from the end of unterminated lines leads to limitations on



(a) Conventional shared backplane.

(b) Switched backplane.

Figure 3: A comparison of conventional and switched backplane designs for a high performance router.

the transfer capacity of a bus. State of the art shared buses today can achieve a maximum capacity today of about 20Gb/s [3]. This is more than adequate for bridges and routers with a few 100Mb/s Ethernet ports. But for our router with line-rates of 2.4Gb/s, we need an aggreagte bandwidth of 40Gb/s, and so a shared backplane cannot keep up.

5 Switched Backplanes: An Overview

5.1 Crossbar Switch

The limitations of a shared backplane are being increasingly overcome by use of a switched backplane. Figure 3 makes a simplified comparison between a conventional shared backplane and a switched backplane comprised of multiple line cards arranged around a central *crossbar* switch. Crossbar switches are widely used for switched backplanes because of their simplicity.

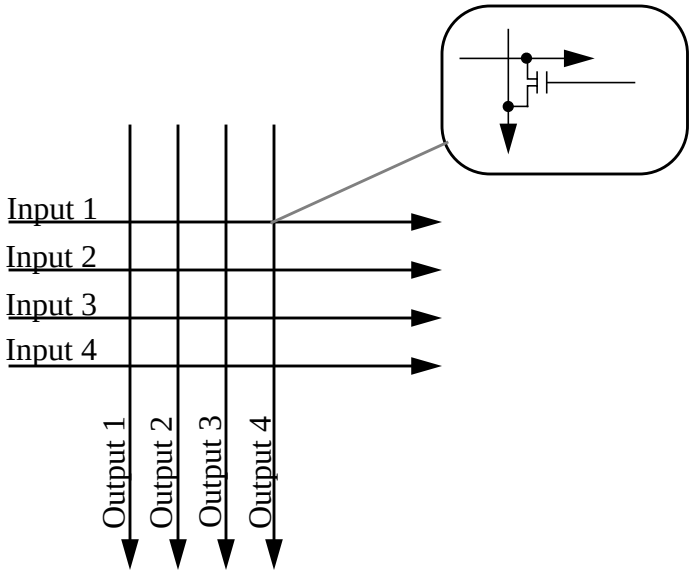


Figure 4: A 4-input crossbar interconnection fabric.

The crossbar switch enables high performance for two reasons: first, connections from Line Cards to the central switch are now simple point-to-point links, which means that they can operate at very high speed. In the last couple of years, semiconductor companies have developed chip-to-chip serial links operating at over 1Gb/s using a conventional CMOS process. These serial links are now being deployed in a number of commercial switches and routers. And laboratory results suggest that links operating at 4-10Gb/s will be available in the next few years. With only one transmitter on each wire, reflections can be controlled allowing signals to settle in a shorter time. Short point-to-point links also help control clock skew, signal integrity, and reduce electromagnetic interference.

The second reason that a crossbar switch can provide higher performance is because it can support *multiple* bus transactions simultaneously. This greatly increases the aggregate bandwidth of the system. A crossbar switch is shown in Figure 4; by closing several crosspoints at the same time, the switch can transfer packets between multiple ports simultaneously. In fact, we say that a crossbar is internally *non-blocking* because it allows *all* inputs and outputs to transfer packets simultaneously. Note that *all* crossbar switches are internally non-blocking.

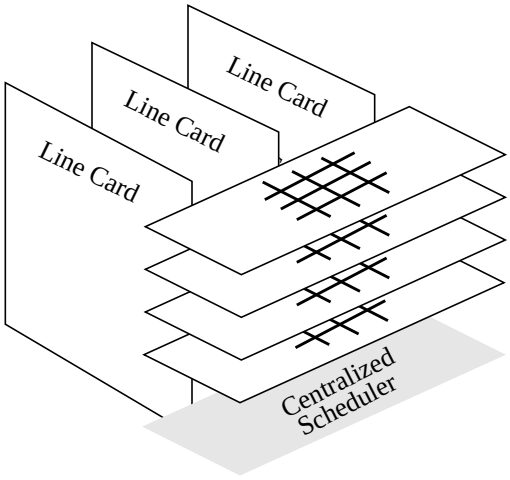


Figure 5: A four-way parallel crossbar switch, interconnecting three line cards. A centralized scheduler connects to each line card, and determines the configuration of the crossbar switch for each time slot.

A centralized scheduler considers all of the packets waiting to be transferred across the switch fabric and selects a configuration of the crossbar ensuring that at any one instance each input is connected to at most one output, and each output is connected to at most one input.

The simple structure of a crossbar switch allows it to be readily implemented in silicon. Furthermore, we can boost performance by using multiple crossbar switches in parallel. Figure 5 shows a system with multiple line cards connected to a four-way parallel crossbar. During each time slot, all of the crossbar "slices" are set to the same configuration, presenting a 4-bit wide bus from each line card to the switch core. If the connection from each Line Card to each slice of the crossbar switch uses one of the fast serial links described above, we can readily build a switched backplane with a very high aggregate bandwidth.

5.2 Fixed vs. Variable Length Packets

Packets may be transferred across the switched backplane in small fixed sized units, or as variable length packets. As we shall see shortly, there are significant disadvantages against using variable length packets. And so the highest-performance routers segment variable length packets into fixed sized units, or "cells"[†], before transferring them across the backplane. The cells are reassem-

bled back into variable length packets at the output before being transmitted on the outgoing line. Let's examine the choice between using fixed, and variable length packets.

Life is simple if we use fixed size cells. We can think of time progressing in fixed size increments, which we call "time slots". All outputs and inputs become free at the same time — at the end of every time slot. At the end of the time slot, the scheduler examines the packets that are waiting to be tranferred across the crossbar switch. It then selects a configuration, deciding which inputs will be connected to which outputs in the next time slot. As we shall see later, it is possible to make this scheduling decision in a way that is fair among connections, never starves an input or output, and maintains highly efficient use of the crossbar switch. Efficiency is important — the crossbar switch is a central resource shared among all of the Line Cards. If we don't use it efficiently, and waste its bandwidth, the performance of the whole system will be degraded. Finally, from the perspective of hardware design, experience shows that processing fixed size cells is simpler and faster than handling variable length packets.

Life is much harder if we use variable length packets; in particular, things are made difficult for the scheduler. Because variable length packets may finish traversing the crossbar switch at *any* time, the scheduler must keep constant track of those outputs that are busy and those that are idle. As soon as an output becomes idle, the scheduler must quickly choose which input is allowed to connect to it next. Unfortunately, the choice is not straightforward as there can be several inputs waiting for the same output. The problem for the scheduler is this: if an idle input has packets for several different outputs, should it make the input wait for a busy output to become free (for a possibly long time), or should it immediately connect to an idle output? If it does not wait for busy outputs, it can perpetually starve packets to that output, placing the switch at the mercy of the arriving traffic. For this reason, switches that use variable length packet generally have lower throughput, and are able to use only about half the system bandwidth. The rest of the bandwidth is wasted by the scheduling mechanism.

Because of their superior performance, we choose to use switched backplanes with *fixed* size cells. We will make this assumption throughout the rest of this paper.

---

†. The name is obviously borrowed from ATM. From now on, we will refer to all fixed length packets as cells. Note that they can be of any length, or format. In particular, they will not necessarily have the same length or format as ATM cells.

5.3 Input Queueing and Virtual Output Queueing

Even though a crossbar switch is always internally non-blocking, there are three other types of blocking that can limit its performance. Understanding these blocking mechanisms is important — our goal is to build a very high performance switched backplane using a crossbar switch and so we must take steps to avoid blocking mechanisms wherever possible. As we will see, we adopt different mechanisms to eliminate or mitigate each type of blocking in a systematic way.

The first type of blocking is called *head-of-line (HOL) blocking;* the second and third are very closely related to each other, and we call them *input-blocking* and *output-blocking*, respectively. As we shall see shortly, *HOL-blocking* can significantly reduce the performance of a crossbar switch, wasting nearly half of the switch's bandwidth. Most routers today that use crossbar switches suffer from *HOL-blocking*. Fortunately, there is a solution for this problem called *virtual output queueing*, that eliminates *HOL-blocking* entirely and makes 100% of the switch bandwidth available for transferring packets. Our switch will use virtual output queueing, and so we will examine this method in the next section. The other types of blocking, *input-* and *output-blocking*, are present in all crossbar switches. They arises because of contention for access to the crossbar: each input line and each output line of a crossbar can only transport one cell at a time. If multiple cells wish to access a line simultaneously, only one will gain access while the others will be queued. Fortunately, as we shall see later, *input-* and *output-blocking* do not reduce the *throughput* of a crossbar switch. Instead, they increase the *delay* of individual packets through the system, and perhaps more importantly make the delay random and unpredictable. Fortunately, we can reduce this problem with two techniques: the first is to use a *prioritization* mechanism in the crossbar switch, and the second is to use a technique called *speedup.* Our switch uses both methods to control packet delay, and so we will examine these methods in detail in Section 7.

Overcoming HOL-blocking: In the very simplest crossbar switches, all of the cells waiting at each input are stored in a single FIFO queue. When a cell reaches the head of its FIFO queue, it is considered by the centralized scheduler. The cell contends for its output with cells destined to the same output, but currently at the HOL of other inputs. It is the job of the centralized scheduler to decide which cell will go next. Eventually, each cell will be selected and will be delivered to its output across the crossbar switch. Although FIFO queueing is widely used, it has a problem: cells can be held up by cells ahead of them that are destined to a different output. This phenomenon is called

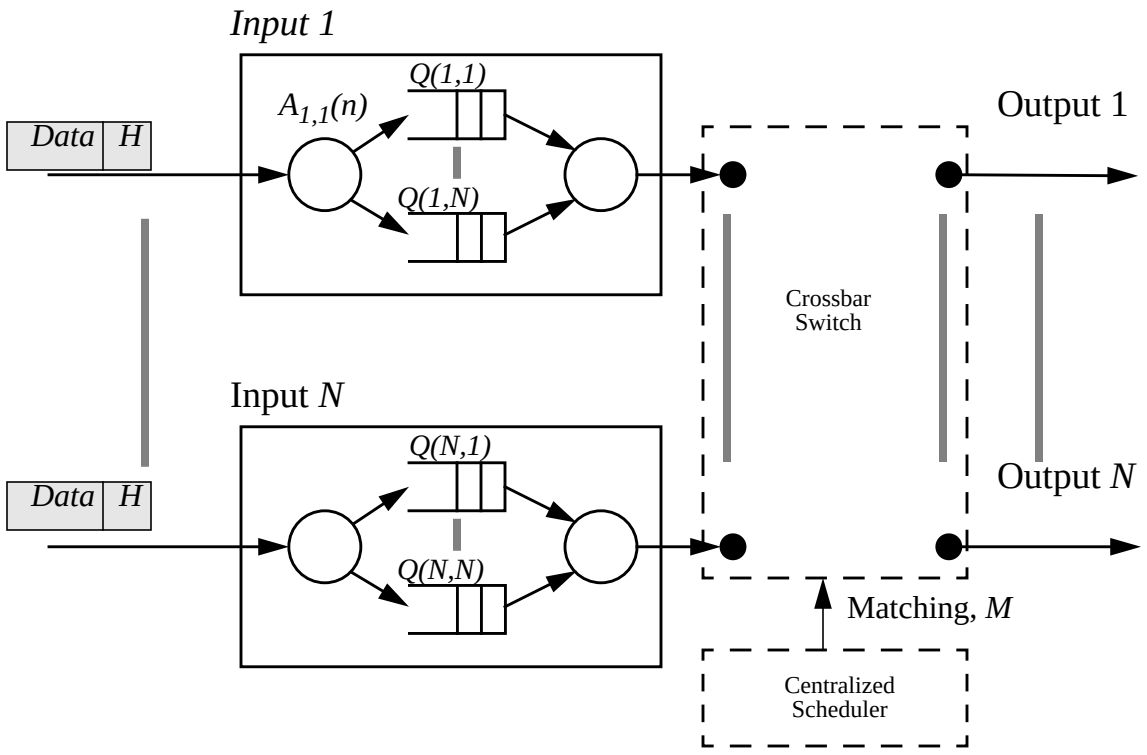


Figure 6: A model of an *N*-port input-queued switch with virtual output queueing (VOQ). Cells arrive at input *i*, and are placed into the appropriate VOQ. At the beginning of each time slot, the centralized scheduler selects a configuration for the crossbar, by matching inputs to outputs. Head of line blocking is eliminated by using a separate queue for each output at each input.

HOL-blocking. The centralized scheduler only "sees" the cell at the head of the FIFO queue, and so the HOL cell blocks packets behind it that need to be delivered to different outputs. A good analogy is a traffic-light in which traffic turning to the right is allowed to proceed even when the light is red. If there is only a single lane and you wish to turn right, but the car ahead of you wants to go straight-ahead, you are blocked: with a single lane you can't pass the car in front to reach your "free" output. Even under benign traffic patterns, HOL blocking limits the throughput to just 60% of the aggregate bandwidth for fixed [4] or variable length packets. When the traffic is bursty, or favors some output ports the throughput can be much worse.

Fortunately, there is a simple fix to this problem known as virtual output queueing (VOQ), first proposed in [5]. At each input, a separate FIFO queue is maintained for each output, as shown in Figure 6. After a forwarding decision has been made, an arriving cell is placed in the queue corre-

sponding to its outgoing port. At the beginning of each time slot, a centralized scheduling algorithm examines the contents of all the input queues, and finds a conflict-free match between inputs and outputs. In theory, a scheduling algorithm can increase the throughput of a crossbar switch from 60% with FIFO queueing to a full 100% if VOQs are used [6]. This is because HOL blocking has been eliminated entirely: no cell is held up by a cell in front of it that is destined to a different output. In our traffic-light example, this is the equivalent to having a dedicated right turn lane: the car in front of you no longer blocks you, and you are free to proceed to your free output.

Our switched backplane will use virtual output queueing to boost performance.

5.4 Crossbar Scheduling Algorithms

In recent years, there has been a significant amount of research work on scheduling algorithms for crossbar switches that use virtual output queueing. Scheduling algorithms have been developed by a number of researchers [23][24][25][26], for research prototypes [2][10], and commercial products [7].

What makes a good crossbar scheduling algorithm for the backplane of a router? We desire algorithms with the following properties:

- *High Throughput* — An algorithm that keeps the backlog low in the VOQs. Ideally, the algorithm will sustain an offered load up to 100% on each input and output.
- *Starvation Free* — The algorithm should not allow any VOQ to be unserved indefinitely.
- *Fast* — To achieve the highest bandwidth switch, it is important that the scheduling algorithm does not become the performance bottleneck. The algorithm should therefore select a crossbar configuration as quickly as possible.
- *Simple to implement* — If the algorithm is to be fast in practice, it must be implemented in special-purpose hardware; preferably within a single chip.

5.5 The *i*SLIP Algorithm

The *i*SLIP algorithm is designed to meet our goals. *i*SLIP is an iterative algorithm — during each time slot, multiple iterations are performed to select a crossbar configuration, matching inputs to outputs. The *i*SLIP algorithm uses rotating priority (“round-robin”) arbitration to schedule each active input and output in turn. The main characteristic of *i*SLIP is its simplicity; it is readily imple-

mented in hardware and can operate at high speed. A prototype version of *i*SLIP is currently being implemented that makes a new scheduling decision every 40ns [10].

*i*SLIP attempts to quickly converge on a conflict-free match in multiple iterations, where each iteration consists of three steps. All inputs and outputs are initially unmatched and only those inputs and outputs not matched at the end of one iteration are eligible for matching in the next. The three steps of each iteration operate in parallel on each output and input and are shown in Figure 5. The steps of each iteration are:

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

Step 2. *Grant*. If an output receives any requests, it chooses the one that appears next in a fixed, round-robin schedule starting from the highest priority element. The output notifies each input whether or not its request was granted.

Step 3. *Accept*. If an input receives a grant, it accepts the one that appears next in a fixed, round-robin schedule starting from the highest priority element. The pointer a_i to the highest priority element of the round-robin schedule is incremented (modulo *N*) to one location beyond the accepted output. The pointer g_i to the highest priority element at the corresponding output is incremented (modulo N) to one location beyond the granted input. The pointers are only updated after the first iteration.

By considering only unmatched inputs and outputs, each iteration matches inputs and outputs that were not matched during earlier iterations.

Despite its simplicity, the performance of *i*SLIP is surprisingly good. A detailed study of its performance and implementation can be found in [8], and more briefly in [9]. In brief, its main properties are:

Property 1. *High Throughput* — For uniform, and uncorrelated arrivals, the algorithm enables 100% of the switch capacity to be used.

Property 2. *Starvation Free* — No connection is starved. Because pointers are not updated after the first iteration, an output will continue to grant to the highest priority requesting input until it is successful. Furthermore *i*SLIP is in some sense fair: with one iteration and under heavy load, all queues with a common output have identical throughput.

Property 3. *Fast* — The algorithm is guaranteed to complete in at most *N* iterations. However, in practice the algorithm almost always completes in fewer than $\log_2(N)$ iterations. i.e. for a switch with 16 ports, four iterations will suffice.

Property 4. *Simple to implement* — An *i*SLIP scheduler consists of 2*N* programmable priority encoders. A scheduler for a 16-port switch is readily implemented on a single chip[†].

To benefit from these properties, our switched backplane will use VOQs and the *i*SLIP scheduling algorithm.

6 Unicast Traffic: Performance Comparison

It is worth pausing to consider our choices so far, and compare the performance with some alternative approaches. The graph in Figure 7 plots the average delay seen by cells waiting to traverse backplane against the amount of traffic arriving to the system. In each case, there is an offered load at which the system saturates, and the delay grows very rapidly. It is at this operating point that the backplane becomes so severely congested that it must start dropping packets. We can easily infer from this graph how much of the system bandwidth is available for transferring useful packets. For example, let's first consider the bandwidth available for the shared bus. We see that even though the bus is running at four times the speed of each Line Card, it saturates when the offered load at each 2.4Gb/s input line is just 600 Mb/s. If we move to a switched backplane with VOQs and the *i*SLIP scheduling algorithm, we see that the maximum achievable throughput jumps all the way to 100%. In other words, the switched backplane can support all of its lines even when they are fully loaded. The situation is not nearly so good if we don't use VOQs, or if we use variable length packets instead of fixed length packets. The graph shows that in this case, only about 60% of the system bandwidth is available. Put another way, about 15Gb/s is wasted due to HOL blocking. We summarise these results in Table 1.

TABLE 1.

Architecture	**Shared Bus**	**Switched Backplane**		
	9.6Gb/s bus	**FIFO Queues**	**Variable Length Packets**	**VOQs & *i*SLIP**
Maximum Throughput	9.6 Gb/s (25%)	23 Gb/s (60%)	23 Gb/s (60%)	38 Gb/s (100%)

7 Controlling Packet Delay

Up until now, we have concentrated on how to overcome HOL blocking, and increase the system throughput from 60% to 100% of the available switching bandwidth. While this is a major improvement over existing switched backplanes, there is another problem faced by crossbar

†. In [10], a 32-port *i*SLIP scheduler is implemented on a single chip.

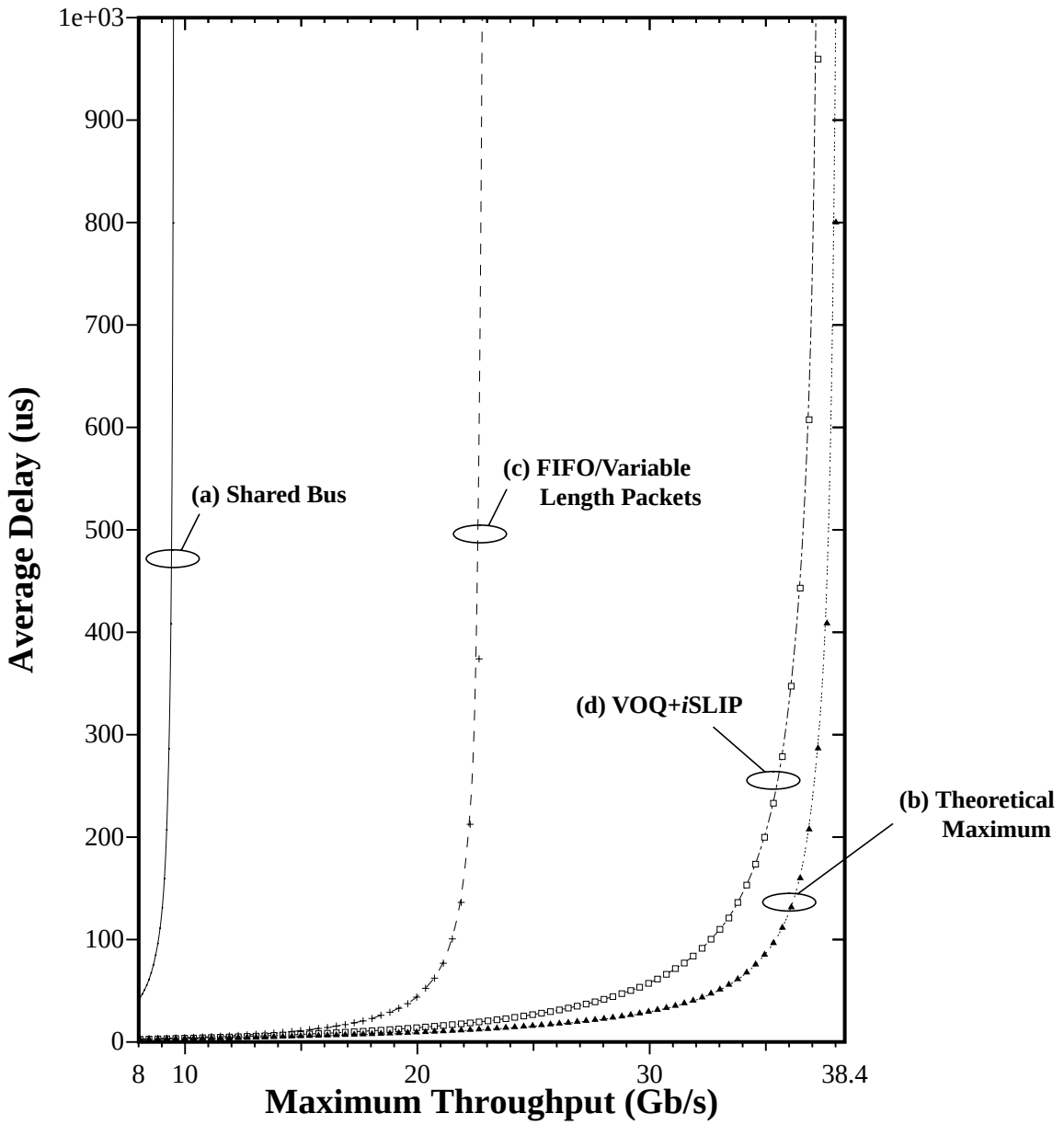


Figure 7: Graph comparing the performance of a variety of switched backplanes for a 16×16 router, with each port operating at 2.4Gb/s. The traffic is benign: random, but uniform traffic is applied to each input of the router. The graph plots aggregate traffic load at all the ports against packet delay, (in μs). In each case, there is a traffic load which saturates the router, and the packet delay grows without bound. Lines (a) and (d) show the two extremes: line (a) is for a conventional router with a shared bus operating at four times the line rate. This router can only support a total aggregate traffic load of 9Gb/s. Line (b) on the other hand represents the maximum theoretical throughput achievable by *any* router. Inbetween these two extremes lie two alterantive crossbar designs. Line (c) shows what happens if we use a crossbar switch, but with FIFO queues. The performance is limited to about 22Gb/s, and is well short of the theoretical maximum. But if we use VOQs and the iSLIP scheduling algorithm, line (d) shows that the throughput comes close to the maximum achievable: 38.4Gb/s.

switches: it is difficult to control the delay of packets across the backplane. This can be a great dis-

advantage for routers that attempt to control packet delay as part of their support for multimedia traffic.

Before considering this problem, let's revisit the way in which packet delay is controlled in a router. Arriving packets are delivered across the switched backplane to their outgoing port where an *output-link scheduler* determines the exact time that a packet will be transmitted on the outgoing line. If, however, the time that it takes for the packet to reach the outgoing port is unpredictable and uncontrollable, then the output link scheduler cannot precisely schedule the packet's departure time. As we shall see, both *input-* and *output-blocking* make a packet's delay unpredictable. But fortunately, two methods can help us overcome this problem: *prioritization*, and *speedup*.

Input-Blocking: To understand how input-blocking occurs, consider again the VOQ switch illustrated in Figure 6. At the end of each cell time, the *i*SLIP scheduler selects at most one VOQ to be served at each input. Input-blocking arises because VOQs at one input can be blocked by other VOQs at the same input that receive preferential service. Consider now what happens when a VOQ at one input is selected to be serviced by the scheduler: other non-empty VOQs at the same input must wait until a later cell-time to receive service. In fact, it is very difficult to predict exactly *when* a non-empty VOQ will receive service. This is because it depends on the occupancy of other VOQs at the same input. The VOQ must contend for access to the crossbar switch with other VOQs that may *block* it for an unpredictable number of cell times.

Output-Blocking: Output-blocking occurs because each output line from the crossbar switch can only transfer one cell at a time. Consider two cells at different inputs that are both waiting to be transferred to the same output. Only one cell can be transferred at a time, while the other cell will be *blocked* until a later cell time. As with input-blocking, *output-blocking* makes it very difficult to predict when a cell will be delivered to its output.

In our switched backplane, we use two techniques to control the delay of packets. The first technique is to separate packets into different *priority classes* according to their urgency; packets with a higher priority are given preferential access to the crossbar switch. Although this does not reduce the total amount of *input-* or *output-blocking*, it prevents low priority packets from affecting the delay of high priority packets. In essence, high priority cells see a less-congested switch — a high priority cell can only be blocked by another high-priority cell. This simple prioritization technique

is found to be very effective, as can be easily seen from Figure 7. We use a reservation protocol such as RSVP to limit the amount of high priority traffic entering the switch. Figure 7 clearly shows that if the amount of traffic in the highest priority class is kept relatively small, the queueing delay is close to zero. High priority cells will therefore be transferred to the output port with almost constant (and negligible) delay. Once they reach the output port, their departure time can be strictly controlled to meet the quality of service requirements.

Prioritization works well when the amount of high-priority traffic is kept small. However, with the fraction of multimedia traffic growing in the Internet, we would like to offer multiple priority classes, and be able to control the delay in each class. This is why we use our second technique: *speedup*.

Speedup: A common way to reduce *input-* and *output-blocking* is to run the crossbar switch faster than the external line rate. For example, if we operate the crossbar switch twice as fast as the external line, we can transfer *two* cells from each input port, and *two* cells to each output port during each cell time. In this case, we say that the switch has a speedup of two. The advantage of speedup is obvious: delivering more cells per cell time reduces the delay of each cell through the switch. In fact, with sufficient speedup, we can guarantee that every cell is *immediately* transferred to the output port, where its departure time can be precisely scheduled.

So why not determine how much speedup is needed, and always run the backplane at that speed?

Unfortunately, if we want to ensure that no packets are queued at the input, it is generally believed to require a speedup of *N*, where *N* is the number of ports. This is very unfortunate: it means that the queue memories and links connecting the memories to the crossbar must run *N* times faster than for a conventional system with a speedup of just one. This is totally impractical for high-performance routers: their performance is invariably limited by the availability of fast memories. Significant speedup is either uneconomical, or simply impossible. For example, if the Cisco 12000 were to have a speedup of *N*, the memory bandwidth at each output port would increase from 2.4Gb/s to 38.4Gb/s. Even if the memories were arranged to be one cell wide, the memory devices would have to perform a read or write operation every 5ns.

However, research work in recent years has suggested that a crossbar switch with a speedup of two behaves *almost identically* to a crossbar switch with a speedup of N; independently of N. This has been confirmed quite recently with the following results: (i) If a crossbar switch maintains only a single FIFO queue, then a speedup of N is required for the cells to experience predictable delay. (ii) If instead VOQs are used, then a speedup of just *four* will suffice. In fact, results suggest that it will soon be possible to precisely control delay through a switch with VOQs and a speedup of just *two*.

We summarize the effect of speedup in Figure 8,where we see that if VOQs are used, then a speedup of just 2 is sufficient to achieve minimum delay. This result clearly demonstrates the benefit of using VOQs rather than FIFO queues.

8 Supporting Multicast Traffic

So far we have focussed on how a switched backplane can efficiently transfer unicast packets. But it is becoming increasingly important for a router to efficiently support multicast traffic. In this section, we consider how our switched backplane can be modified to support multicast traffic at very high performance. As we will see, we can achieve very high performance using the same crossbar switch, and a modified version of the *i*SLIP scheduling algorithm, called ESLIP.

A simple way to service the input queues is to replicate the input cell over multiple cell times, generating one output cell per time slot. However, this approach has two disadvantages. First, each input must be copied multiple times, increasing the required memory bandwidth. Second, input cells contend for access to the switch multiple times, reducing the bandwidth available to other traffic at the same input. Higher throughput can be attained if we take advantage of the natural multicast properties of a crossbar switch. So instead, we will design our system to copy an input cell to any number of idle outputs in a single cell time.

In our combined switch, we maintain two types of queues: unicast cells are stored in VOQs, and multicast cells are stored in a separate multicast queue. By closing multiple crosspoints simulta-

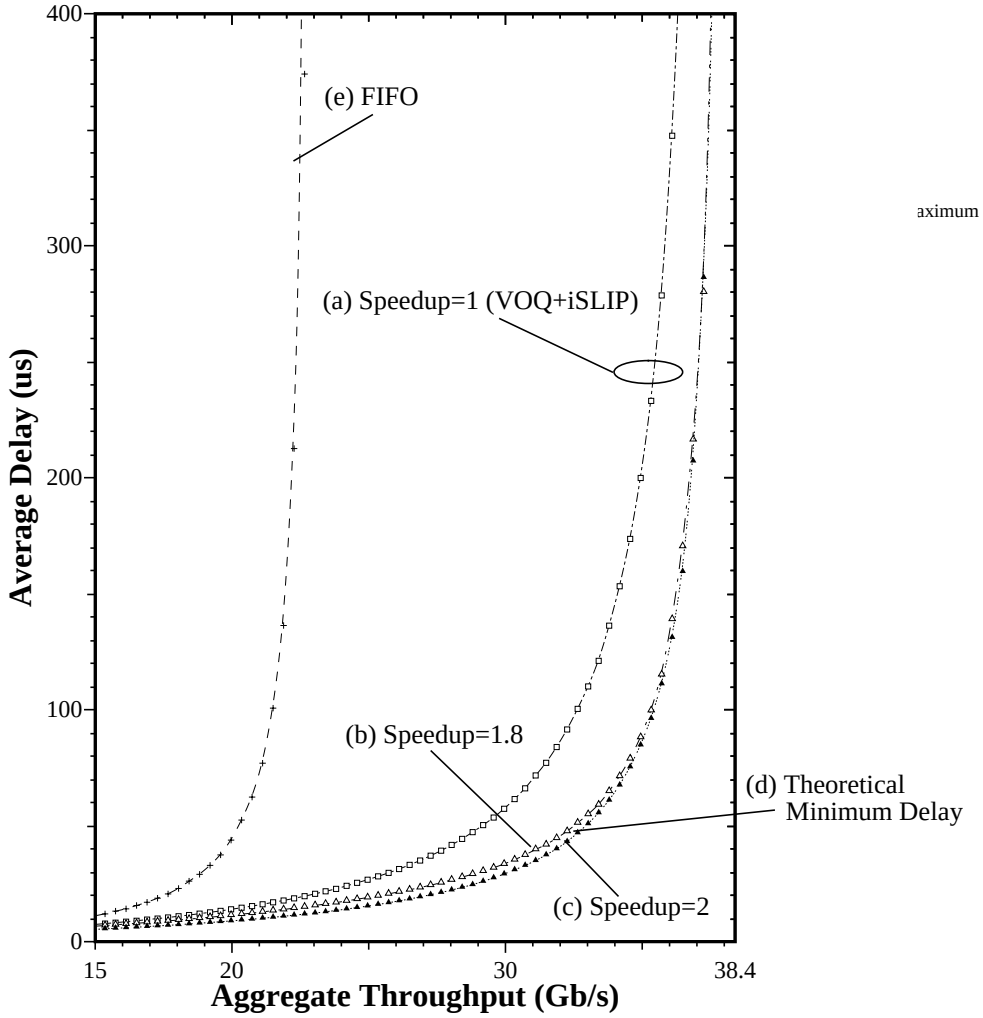


Figure 8: The effect of speedup on the delay of packets through a switched backplane. (a) If VOQs are used along with the iSLIP scheduling algorithm, the throughput can achieve 100%, but the delay is larger than the theoretical minimum delay. (b) With a speedup of just 1.8, the delay is reduced to close to the minimum. (c) When the speedup equals 2, the performance is indistinguishable from the minimum achievable delay.

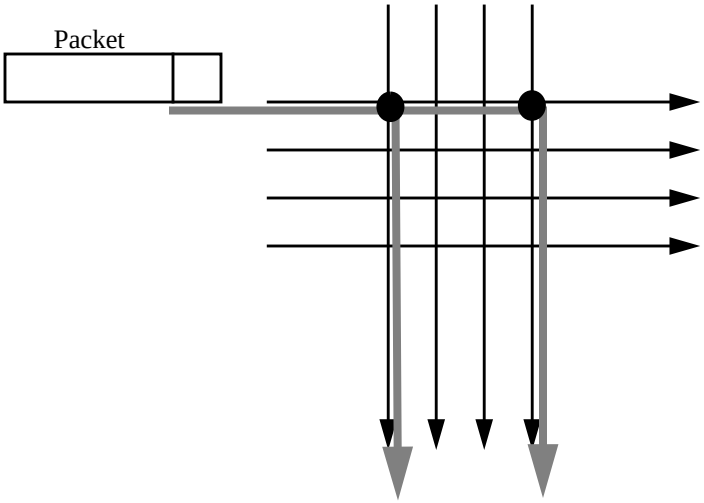


Figure 9: In a crossbar fabric, multicast packets can be sent to multiple outputs simulatneously by "closing" multiple crosspoints at the same time. This example shows a 4-input switch forwarding a multicast packet to two destinations.

neously, the crossbar switch is used to perform cell replication within the fabric, as shown in Figure 8. As before, at the beginning of each time slot, a centralized scheduling algorithm decides which crosspoints to close.

As a simple example of how a crossbar switch can support multicast, consider the 2 input and *N* output switch shown in Figure 9. Queue Q_A has an input cell destined for outputs {1,2,3,4} and queue Q_B has an input cell destined for outputs {3,4,5,6}. The set of outputs to which an input cell wishes to be copied is called the *fanout* of that input cell. For example, in Figure 9, the input cell at the head of each queue is said to have a fanout of four.

For clarity, we distinguish an arriving *input* cell from its corresponding *output* cells. In the figure, the single input cell at the head of queue Q_A will generate four output cells. For practical reasons, an input cell waits in line until all of the cells ahead of it have departed.

8.1 Scheduling Multicast Traffic

There are two different service disciplines that we can use. The first is called *no fanout-splitting* in which all of the copies of a cell must be sent in the same cell time. If any of the output cells loses contention for an output port, none of the output cells are transmitted and the cell must try again in the next cell time. The second discipline is called *fanout-splitting*, in which output cells may be

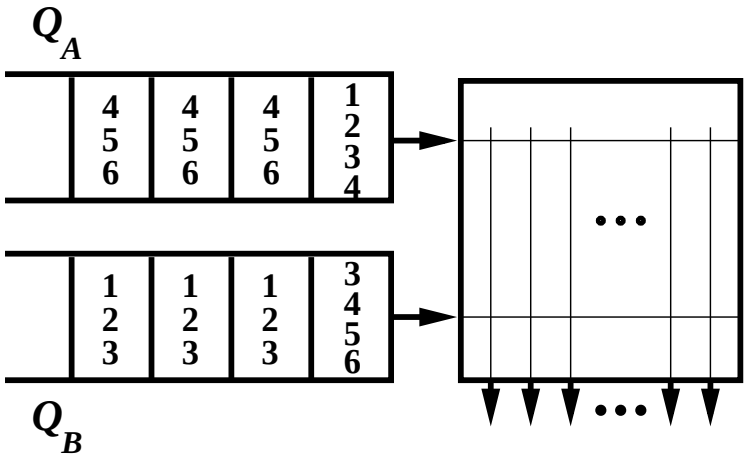


Figure 10: A $2 \times N$ crossbar switch that supports multicast. Each input maintains a special FIFO queue for multicast cells.

delivered to output ports over any number of cell times. Only those output cells that are unsuccessful in one cell time continue to contend for output ports in the next cell time.† Research has shown that fanout-splitting enables a higher switch throughput for little increase in implementation complexity. For example, Figure 10 compares the average cell latency (via simulations) with and without fanout-splitting of a random scheduling algorithm. The figure demonstrates that fanout-splitting can lead to approximately 40% higher throughput. Because of its simplicity, and performance benefits, we use fanout-splitting in our multicast scheduling algorithms.‡

8.2 ESLIP: Combining Unicast and Multicast

To support multicast, we need a new scheduling algorithm — one that can examine the contents of the multicast queues at each input, and select which output cells will be delivered in each time

†. It might appear that *fanout-splitting* is difficult to implement. However this is not the case. In order to support *fanout-splitting*, we need one extra signal from the scheduler to inform each input port when a cell at its HOL is completely served.

‡. For an in-depth comparison of a variety of scheduling algorithms with and without fanout-splitting, the reader is referred to [12][29][30]

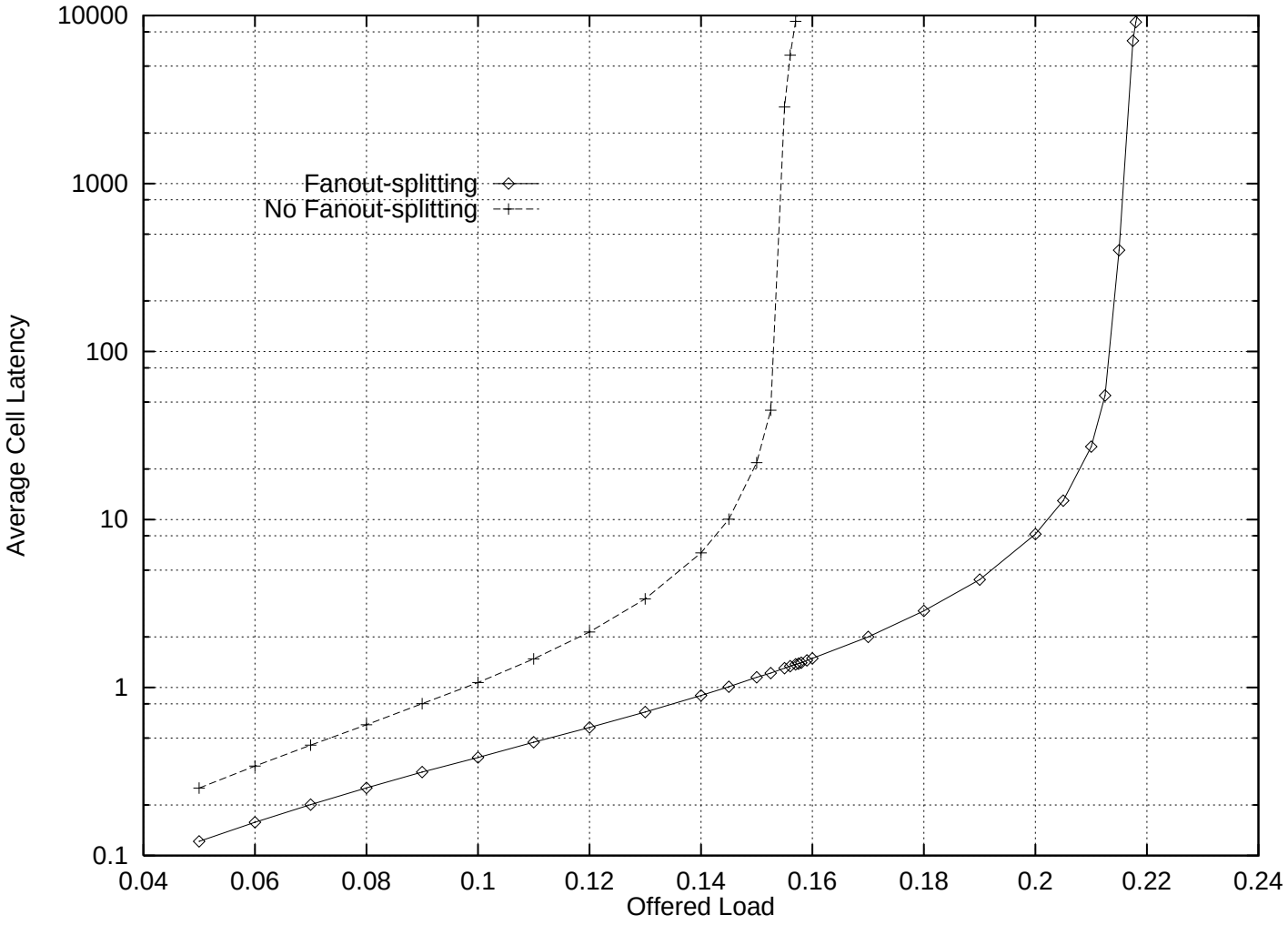


Figure 11: Graph of average cell latency (in number of cell times) as a function of offered load for a 8-port switch (with uniform input traffic and average fanout of four). The graph compares a random scheduling policy with and without *fanout-splitting*.

slot. Furthermore, the algorithm needs to choose between competing unicast and multicast cells, each with its own priority.

For this purpose, we have devised a novel algorithm, ESLIP, that efficiently schedules unicast and multicast traffic simultaneously, and is able to provide preferential service to cells with higher priority. ESLIP is an enhanced version of the unicast *i*SLIP algorithm.

The queues maintained by a 16-port switch supporting four priority levels are illustrated in Figure 11. At the beginning of each time slot, the ESLIP scheduler examines the contents of all the

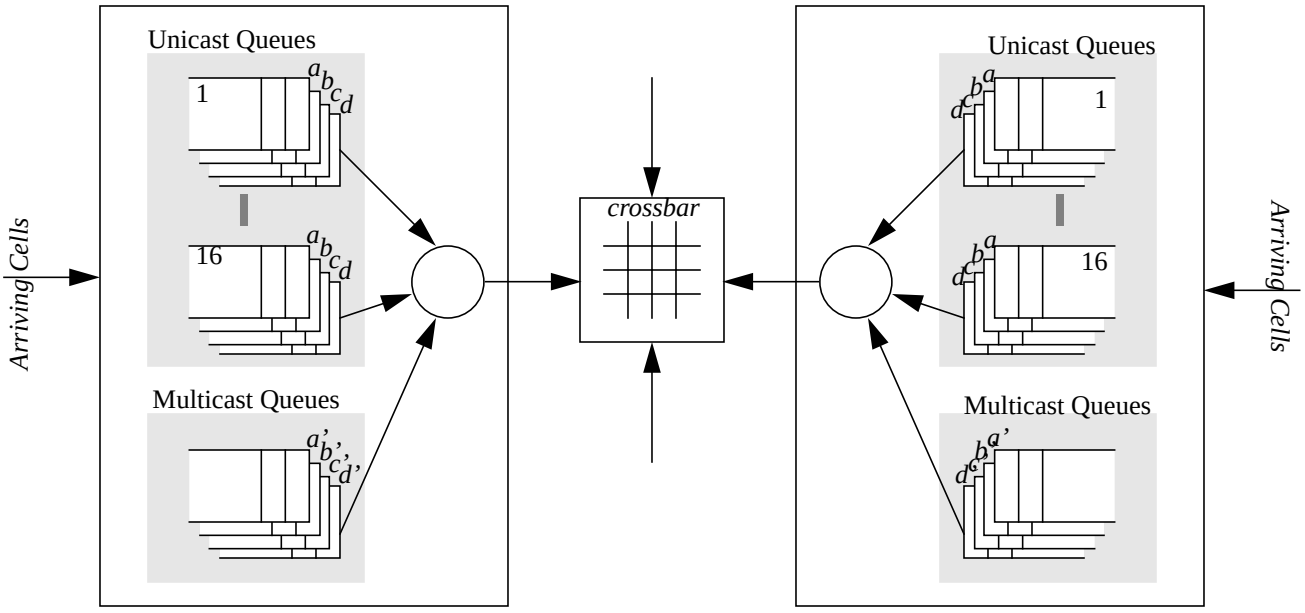


Figure 12: The queues maintained by each input for a 16-port combined unicast-multicast switch, with four priority levels {a,b,c,d}. Each input maintains 16 VOQs for unicast traffic and 1 for multicast traffic, for each priority level. Hence, there are 68 queues per input.

queues. Like *i*SLIP, it is an iterative algorithm: all inputs and outputs are initially unmatched, with each successive iteration adding additional connections. To keep track of which connection to favor next, each output maintains a separate grant pointer for each priority level. For example, output j keeps pointers g_j^a through g_j^d. In addition, all the outputs share common pointers for multicast: $g_M^{a'}$ through $g_M^{d'}$. Likewise, each input maintains a separate accept pointer for each priority level, and share common pointers $a_M^{a'}$ through $a_M^{d'}$ for multicast traffic. Each iteration consists of three steps:

Step 1. *Request*. Each input sends a request to every output for which it has a queued unicast cell. The request if prioritized; the input sends a priority equal to the highest priority cell that it has for each output. Likewise, each input sends a prioritized request to every output for which it has a queued multicast cell.

Step 2. *Grant*. If an output receives any requests, it must choose just one. First, it considers all the unicast and multicast requests, keeping those equal to the highest priority, and discarding the rest. The remaining requests are either all unicast, or all multicast. If they are unicast requests at priority *k*, it chooses the input that appears next in a fixed, round-robin schedule starting from its unicast pointer value g_i^k. This is the same as the grant procedure for *i*SLIP. If they are multicast requests, it chooses the input that appears next in a fixed, round-robin schedule starting from the global multicast pointer g_M^k. The output notifies each input whether or not its request was granted.

Step 3. *Accept*. If an input receives any grants, it accepts just one. First, it considers all the unicast and multicast grants, keeping those equal to the highest priority, and discarding the rest. If they are unicast grants at priority level *k*, it chooses the output that appears next in a fixed, round-robin schedule starting from its unicast pointer value a_i^k. The pointers a_i^k and g_i^k are both incremented as for *i*SLIP. If they are multicast grants, it chooses the output that appears next in a fixed, round-robin schedule starting from the global multicast pointer a_M^k. If this connection completes the fanout, the global multicast pointers a_M^k and g_M^k are updated. Finally, to avoid starvation, the pointers are only updated after the first iteration.

Although the ESLIP algorithm is more complex than the *i*SLIP algorithm, it is surprisingly simple to implement in hardware. A schematic implementation is shown in Figure 12. The scheduler consists of two sets of arbiters — grant arbiters that implement Step 2, and accept arbiters that implement Step 3. An enlarged drawing of an arbiter shows how the arbiter selects between unicast and multiple traffic, and determines the priority level of the selected request.

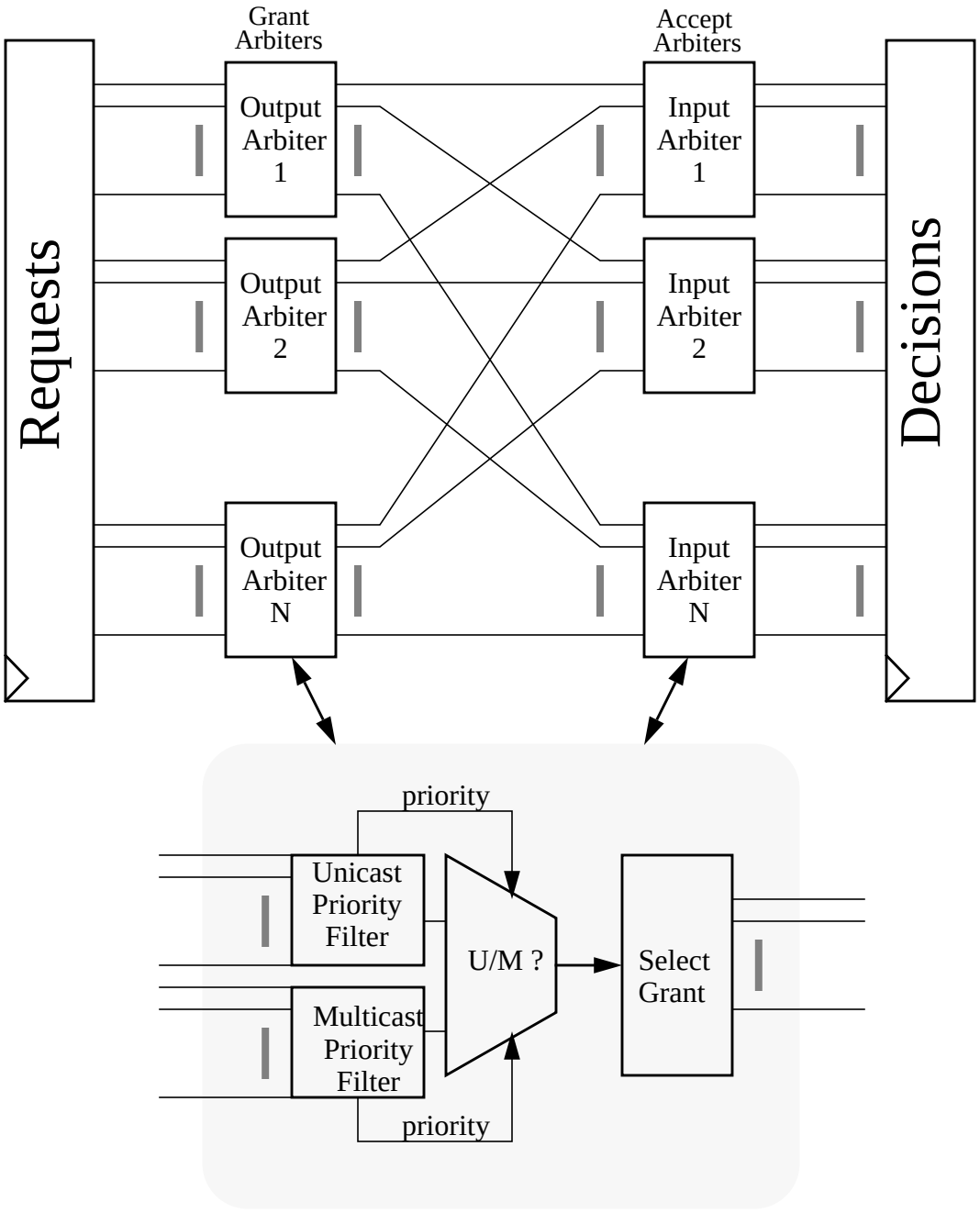


Figure 13: A schematic implementation of the ESLIP algorithm.

The ESLIP algorithm has the following features:

Property 1. *High Throughput* — For unicast traffic, ESLIP performs as well as iSLIP, and hence

for uniform, and uncorrelated unicast arrivals, the algorithm enables 100% of the switch capacity to be used. Because fanout-splitting is used for multicast traffic, a high throughput is possible. In addition, the global multicast pointer helps clear backlog quickly — all of the outputs tend to favor the same input, reducing the need to split fanout.

Property 2. *Starvation Free* — No connection is starved.

Property 3. *Fast* — The algorithm is guaranteed to converge in at most N iterations. However, in practise the algorithm generally converges in fewer than $\log_2(N)$ iterations. i.e. for a switch with 16 ports, four iterations will suffice. Although the algorithm will not necessarily converge to a maximum sized match, it will find a *maximal* match: the largest size match without removing connections made in earlier iterations.

Property 4. *Simple to implement* — Like *i*SLIP, an ESLIP scheduler consists of 2N programmable priority encoders. A scheduler for a 16-port switch can readily be implemented on a single chip.

The Cisco 12000 router uses the ESLIP algorithm to schedule unicast and multicast cells across its backplane.

9 Concluding Remarks

Using the Cisco 12000 as an example, we have seen that a switched backplane based on a crossbar switch offers significant performance advantages over a conventional router with a centralized shared bus. Furthermore, we can expect this architecture to scale well into the future.

However, care must be taken when designing a switched backplane to achieve high throughput and predictable delay. We can summarize our design choices as follows:

Use Fixed Length Packets: Using fixed length packets (“cells”) allows up to 100% of the switch bandwidth to be used for transferring cells. If variable lentgh packets are used, the system throughput is limited to approximately 60%.

Use Virtual Output Queueing: If VOQs are used, instead of the conventional FIFO queues, then head of line blocking can be eliminated entirely. Again, this raises the system throughput from 60% to 100%.

Use Priority Levels and Speedup: Priority levels can make packet delay more predictable, allowing the router to control delay for multimedia traffic. If VOQs and a moderate speedup are employed, then the delay can be made precisely predictable.

Use Separate Multicast Queues: If multicast traffic is queued separately, then the crossbar may be used to replicate cells, rather than wasting precious memory bandwidth at the input.

Use Fanout-Splitting for Multicast Packets: tIf the crossbar implements *fanout-splitting* for multicast packets, then the system throughput can be increased by 40%.

Use a Combined Unicast and Multicast Scheduler: If a single scheduler is used to schedule both unicast and multicast traffic, then: (a) the scheduling decisions can be made at greater speed, and (b) the relative priority of unciast and multicast traffic can be maintained, preventing either type of traffic from starving the other.

10 References

[1] W. Doeringer; G. Karjoth; M. Nassehi, “Routing on Longest-Prefix Matches,” IEEE Transactions on Networking, Vol.4 No.1, pp. 86-97, Feb 1996.

[2] C. Partridge; P. Carvey et al. “A Fifty-Gigabit per Second IP Router,” Submitted to IEEE/ACM Transactions on Networking, 1996.

[3] A. Singhal et al., “Gigaplane (TM): A High Performance Bus for Large SMPs,” Proc. of Hot Interconnects ‘96, Stanford, CA..

[4] M. Karol; M. Hluchyj; S. Morgan, “Input versus output queueing on a space-division switch,” IEEE Transactions on Communications, vol. 35, pp. 1347-1356, Dec 1987.

[5] Y. Tamir; G. Frazier, “High Performance Multiqueue Buffers for VLSI Communication Switches,“ Proc. of 15th Annual Symp. on Computer Arch, June 1988.

[6] N. McKeown; V. Anantharam; J. Walrand, “Achieving 100% Throughput in an input-queued switch,” Proceedings of IEEE Infocom ‘96.

[7] T. Anderson et al. “High Speed Switch Scheduling for Local Area Networks,” ACM Trans. on Computer Systems, Nov 1993, pp. 319-352.

[8] N. McKeown, “Scheduling Cells in an input-queued switch,” PhD Thesis, University of California at Berkeley, May 1995.

[9] N. McKeown, “*i*SLIP: A Scheduling Algorithm for Input-Queued Switches,” Submitted to IEEE Transactions on Networking.

[10] N. McKeown et al. “The Tiny Tera: A small high-bandwidth packet switch core,” IEEE Micro, Jan-Feb 1997.

[11] S. Deering, “Multicast Routing in a Datagram Internetwork,” Ph.D. Thesis, Stanford University, CA, 1991.

[12] B. Prabhakar et al. “Multicast Scheduling for Input-Queued Switches,” To appear in IEEE JSAC, June 1997.

[13] H. Ahmadi; W. Denzel, “A Survey of Modern High-Performance Switching Techniques,” IEEE JSAC, vol. 7, no. 7, pp1091-1103, Sept 1989.

[14] A. Demers, S. Keshav; S. Shenker, “Analysis and Simulation of a Fair Queueing Algorithm,” J. of Internetworking : Research and Experience, pp.3-26, 1990.

[15] L. Zhang, “Virtual Clock: A New Traffic Control Algorithm for Packet Switching Networks,” ACM Transactions on Computer Systems, vol.9 no.2, pp.101-124, 1990.

[16] D. Clark; S.Shenker; L. Zhang, “Supporting Real-Time Applications in an Integrated Services Packet Network:Architectures and Mechanisms,” Proc. SIGCOMM ‘92, August 1992.

[17] H. Zhang, “Service Disciplines for Guaranteed Performance Service in Packet-Switching Networks,” Proc. of the IEEE, vol. 83, no. 10, pp.1374-1396.

[18] A. Brodnik , S. Carlsson, M. Degermark , and S. Pink, “Small Forwarding Tables for Fast Routing Lookups,” Proceedings of Sigcomm ‘97, France, September 1997.

[19] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, “Scalable High Speed IP Routing Lookups,” Proceedings of Sigcomm ‘97, France, September 1997.

[20] S. C. Lin, P. Gupta, N. McKeown, “Fast Routing Lookups in Hardware,” submitted to IEEE Infocom ‘98. Preprint available on request.

[21] S. C. Lin, and N. McKeown, “A Simulation Study of IP Switching,” Proceedings of Sigcomm ‘97, France, September 1997.

[22] Tamir, Y.; and Chi, H-C.;“Symmetric Crossbar Arbiters for VLSI Communication Switches,” *IEEE Trans on Parallel and Dist. Systems*, Vol. 4, No.1, pp.13-27, 1993.

[23] LaMaire, R.O.; Serpanos, D.N.; “Two-dimensional round-robin schedulers for packet switches with multiple input queues,” *IEEE/ACM Transactions on Networking*, Oct. 1994, vol.2, (no.5):471-82.

[24] Lund, C.; Phillips, S.; Reingold, N.; “Fair prioritized scheduling in an input-buffered switch,” *Proceedings of the IFIP-IEEE Conf. on Broadband Communications ‘96*, Montreal, April 1996. p. 358-69.

[25] Hui, J.; Arthurs, E. “A broadband packet switch for integrated transport,” *IEEE J. Selected Areas Communications, 5*, 8, Oct 1987, pp 1264-1273.

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

[27] ATM Forum, “ATM User-Network Interface (UNI) Signalling Specification,” Version 4.0, April 1996.

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

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

[30] J.Y. Hui, and T. Renner, “Queueing Analysis for Multicast Packet Switching’’, *IEEE Transactions on Communications*, vol.42, no.2/3/4}, pp.723-731, Feb 1994.

Nick McKeown is a Professor of Electrical Engineering and Computer Science at Stanford University. He received his Phd from the University of California at Berkeley in 1995. From 1986-1989 he worked for Hewlett-Packard Labs, in their network and communications research group in Bristol, England.

During the Spring of 1995, he worked for Cisco Systems as an architect of the Cisco 12000 GSR router.

Nick serves as an Editor for the IEEE Transactions on Communications. He is the Robert Noyce Faculty Fellow at Stanford, and recipient of a fellowship from the Alfred P. Sloan Foundation.

Nick researches techniques for high speed networks, including high speed Internet routing and architectures for high speed switches. More recently, he has worked on the analysis and design of cell scheduling algorithms, memory architectures, and the economics of the Internet.

His research group is currently building the Tiny Tera; an all-CMOS Terabit per second network switch.