# Part VI: Advanced Topics (Bonus Material on CD-ROM)

This part includes additional material that are related to Part IV and Part V; it consists of two sub-parts.

In the first sub-part, three chapters (Chapter 21, Chapter 22, and Chapter 23) cover functions and components of a router in further detail as a continuation of Part IV. First, different approaches to architect the switch fabric of a router are presented in Chapter 21. Second, packet queueing and scheduling approaches are discussed along with their strengths and limitations in Chapter 22. Third, traffic conditioning, an important function of a router, especially to meet service level agreements, is presented in Chapter 23.

In the second sub-part, we include two chapters (Chapter 24 and Chapter 25). Transport network routing is presented first in its general framework, followed by a formal treatment of the transport network route engineering problem over multiple time periods, in Chapter 24. The final chapter (Chapter 25) covers two different dimensions: optical network routing and multi-layer network routing. In optical network routing, we discuss both SONET and WDM in a transport network framework; more importantly, we also point out the circumstances under which a WDM on-demand network differs from a basic transport network paradigm. Furthermore, we discuss routing in multiple layers from the service network to multiple views of the transport networks; this is done by appropriately considering the unit of information on which routing decision is made and the time granularity of making such a decision. We conclude by presenting overlay network routing and its relation to multilayer routing.

# 21

# Switching Packets

*One never notices what has been done; one can only see what remains to be done.*

**Marie Curie**



***Reading Guideline***

The switching fabric of a router must be extremely efficient so that packets are processed quickly. In this chapter, we present a variety of switching architectures used in routers. It is important to note that many concepts originally came from switching architectures for circuit switching; this connection is highlighted. Understanding switching architectures is helpful in gaining an appreciation of modern-day routers. Furthermore, due to commonalities, the material presented here is useful for understanding similar switching architectures employed in networking technologies such as optical networking.

D. Medhi and K. Ramasamy, *Network Routing: Algorithms, Protocols, and Architectures*.

A switch fabric is a core component of any router that provides a physical path between the ingress line card and the egress line card. In routers, the line cards terminate the network interfaces that carry packet traffic from or to another router or host. As the packets arrive from these interfaces, the line cards perform route lookups to determine their destination. The line card then forwards each packet to the switch fabric, which is responsible for transporting the packet from the ingress line card to the egress line card. At the egress line card, the packet is queued and scheduled for transmission on the output network interface.

In this chapter, we study in detail the architecture of different types of switches, but those used in routers belong to a broader class of packet switches. These switches transport packets that contain data as well as the information needed to determine the destination. While the literature on switch fabrics is vast, we restrict most of our discussion to those that are implemented today in commercial routers. In addition, we also discuss some of the recent advances in switch architectures that are capable of carrying terabits of traffic per second. Sometimes, in the literature, a switch fabric is referred to as a *backplane*.

## 21.1 Generic Switch Architecture

A generic switch fabric in a router has five main components as shown in Figure 21.1.

- The *fabric input interface* connects the ingress network processing modules of a line card to the switch fabric. This component performs various functions: it coordinates with the scheduler regarding the presence of packets, segments the variable-length IP packets into fixed-sized cells if needed, and enables their transmission when indicated by the scheduler.

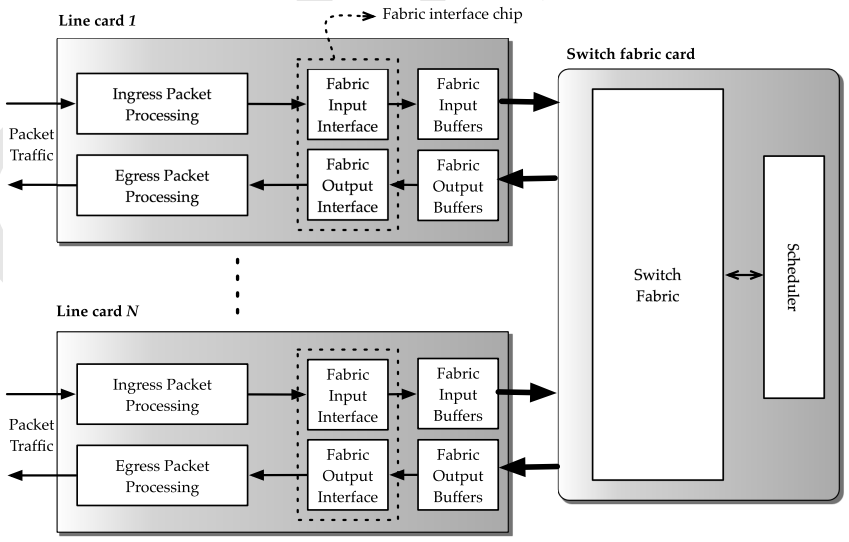


**FIGURE 21.1** A generic switch.

- The *fabric input buffers* provide temporary storage for the packets in case the switch fabric is not able to schedule packets immediately upon their arrival. In addition, the presence of these buffers reduces packet loss during bursty conditions.
- The switch fabric transfers data from one line card to another line card. It dynamically connects multiple ingress line cards to egress line cards to ensure paths are available in the fabric to transfer data simultaneously. The scheduler is the heart of the switch fabric, which identifies the paths through the fabric from ingress line cards to egress line cards.
- The *fabric output interface* receives traffic from the switch fabric and forwards it to the egress network processing modules of the line card. It coordinates with the fabric scheduler to receive packets from the switch fabric. If the packet is originally segmented, it assembles the cells into whole packets. Furthermore, it coordinates with the output scheduler to transmit the packets depending on priority and quality-of-service (QoS) requirements.
- The *fabric output buffers* store the packets as they are awaiting their turn to be transmitted.

In most routers, the fabric input and output interfaces are implemented in a single chip that resides in the line card. The line cards also contain the fabric input and output buffers. However, the switch fabric along with the scheduler is implemented in a separate card called the *switch fabric card*. The switch fabrics can be dichotomized into *shared backplanes* and *switched backplanes*.

## 21.2 Requirements and Metrics

Before delving into individual switch architectures, we need to understand the requirements of a switch fabric when used inside a router. The primary requirement is to maximize the amount of data transferred across the fabric. This means that the fabric should transfer traffic from multiple line cards simultaneously.

A network switch fabric should provide fair bandwidth allocation for all the line cards. This implies that even during a momentary overload, excess traffic destined for line card $A$ should not steal bandwidth from traffic destined for line card $B$ even though the traffic destined for line cards $A$ and $B$ shares the resources of the fabric.

Another desirable requirement is that the switch fabric should not reorder packets. Since higher-layer protocols (like TCP) implement buffering for sequencing out-of-order packets, the natural question is why the switch fabric should not reorder packets. With current routers carrying voice and multimedia traffic, the reordering of packets increases the end-to-end delay, affecting user experience. Such delay-sensitive traffic imposes another requirement: the traffic needs to be prioritized and higher-priority traffic must be transferred across the fabric before the lower-priority traffic.

Since the switch fabric is a central critical component of a router, its failure implies that the router will be unable to forward any packets. Hence, an important requirement is that the switch fabric must provide sufficient redundancy for a router to continue to operate when a fabric failure occurs.

The performance of a switch fabric depends on several factors, such as its internal architecture and the nature of the traffic passing through it. The three primary performance metrics of interest are *throughput*, *latency*, and *path diversity* [163].

- *Throughput:* The throughput of a fabric determines how much data it can transfer in a unit of time. It is measured in bits per second. For instance, consider a router with 16 line cards with each line card capable of sending 40 Gbps of traffic into the fabric. If all the line cards wish to transfer packets simultaneously, then the aggregate throughput of the fabric needs to be 640 Gbps ($= 16 \times 40$ Gbps). Since Internet traffic is growing at a high rate, routers need to forward more packets and thus places more demands on the switch throughput.
- *Latency:* Another metric is the latency experienced by a packet as it travels to the switch fabric. This is significant as IP packets carry multimedia traffic that requires delay guarantees. Formally, the latency in a switch fabric is defined as the time it takes to transfer a packet through the switch fabric from an input port to an output port. To a certain degree, the latency experienced by a packet in a router depends on the latency introduced by the switch fabric.
- *Path Diversity:* This refers to the number of available paths within the switch fabric for every pair of input and output ports. When more than one path is available, the switch fabric is said to be more robust. The traffic can be load-balanced across these paths, which allows the switch to tolerate any component failures.

In the following sections, we study shared and switched backplanes and their representative architectures in detail.

## 21.3 Shared Backplanes

This type of backplane uses a shared medium for transferring packets from one line card to another. The switches using a shared bus or ring topology fall under this category. While this type of backplane is more economical, it is often limited in throughput. Hence, such backplanes are used in low-bandwidth enterprise routers. In the next section, we will discuss the simplest shared backplane, the *shared buses*, in detail.

### 21.3.1 Shared Bus

A shared bus is the simplest and most commonly used form of switching. A bus connects a number of ports with a shared channel that serves as a broadcast medium. Within a router, each port houses a line card. A typical implementation of a bus uses a set of signal lines or a single line connected to all the ports. When a packet is transmitted over the bus, every port receives it. Depending on whether the packet is destined for it, a port chooses to accept the packet or ignore it. A bus protocol determines which port has permission to transmit at any given time. A shared bus with line cards is shown in Figure 21.2.

A shared bus has two key properties. First, it implements broadcast and multicast natively and they are no more expensive than a packet transmitted point-to-point. This is because all the packets transmitted over the bus are broadcast to all the ports. Second, at any given instant, only one port can transmit a packet over the bus and, hence, there is no need for any packet resequencing on the destination line cards.

Now let us turn our attention to how much bus bandwidth will be required. If each port is capable of a data rate of $R$ bps, a bus supporting $N$ line cards needs to operate at a bandwidth

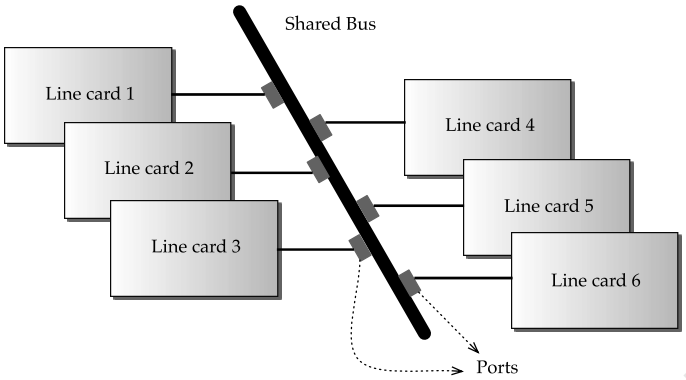


**FIGURE 21.2** A shared bus with 6 line cards connected to ports.

of $RN$ bps. If the bus uses a clock frequency of $r$ Hz, the bus width $w$ must be at least $Rn/r$ bits. Let us go through an example that calculates the bus bandwidth and the bus width.

**Example 21.1** *Shared bus bandwith and width for a router.*

Consider a router with 16 line cards using a shared bus with each line card operating at 100 Mbps. In this scenario, the bus must provide a bandwidth of 1.6 Gbps ($= 100 \times 16$). Assuming the bus uses an internal clock rate of 40 MHz, the bus width should be 40 bits (1.6 Gbps/40 MHz). ▲

With the availability of fast CPUs, in router architectures using a shared bus (discussed in Section 14.6.1), the primary bottleneck is the bus itself. When one line card is sending a packet to another line card, other line cards have to refrain from communicating even though they might have packets to transmit. Clearly, this is not desirable as the need for routing bandwidth is growing exponentially and, hence, high-capacity routers are needed. For instance, consider a high-capacity router that has eight line cards with each operating at 40 Gbps. The required bus bandwidth is, at worst, 320 Gbps ($= 8 \times 40$ Gbps) when all the line cards want to transfer their packets simultaneously. It is not practically feasible to build a shared bus operating at 320 Gbps. At present, a fast off-the-shelf bus commercially available is the PCI Express, which offers speeds of up to 80 Gbps [96], [671].

Another disadvantage with the use of a shared bus is that as the number of ports connected to a bus increases, the electrical loading on the signal lines grows [459], [705]. This reduces the maximum clock frequency that can be achieved (that is, $r$ reduces to $cr$, $0 < c < 1$). Hence, the bus width $w$ should grow more than the number of ports $N$ to maintain sufficient bus bandwidth. For instance, if $c$ is 0.8 due to electrical loading, the bus width for Example 21.1 should be 50 bits ($= 1.6$ Gbps$/(0.8 \times 40$ MHz$)$). While there are techniques to reduce or eliminate the impact of this electrical loading, they are more complex to implement [705].

Because of these limitations, it is necessary to develop backplanes that provide high performance at a reasonable cost. For smaller routers with a few Gbps of throughput, the shared bus is attractive from both a cost and performance perspective.

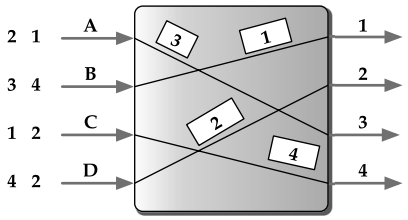


**FIGURE 21.3** A switched backplane showing the transfer of multiple packets simultaneously.

## 21.4 Switched Backplanes

A switched backplane allows packets to be transferred simultaneously between different line cards. Such a parallel transfer of packets increases the aggregate throughput of a backplane. Like a shared backplane, a switched backplane also consists of $N$ ports with each port housing a line card. Since each line card can transmit as well as receive packets simultaneously from the backplane, conceptually it has an input port and an output port. Hence, the switched backplanes are depicted using $N$ input and $N$ output ports.

A typical switched backplane with four ports is shown in Figure 21.3, which shows that multiple packet transfers are occurring simultaneously from input to output ports. For instance, input port $A$ is transferring packet 3 to output port 3 while port $B$ is transmitting packet 1 to output port 1, and so on. Meanwhile, other packets are waiting at the input ports for their turn.

An important component of any switched backplane is the scheduler. The scheduler determines which input ports will transmit their packets to which output ports. Since IP packets are of variable length, the design of the scheduler becomes complex and leads to starvation and reduction in throughput (see Exercise). Hence, variable-length packets are segmented into fixed-sized cells and these cells are scheduled so that their transfers can occur within a fixed time called a *timeslot*. At the end of each timeslot, the scheduling algorithm examines the cells at the input ports waiting to be transferred across the backplane and decides which inputs will be connected to which outputs (for the next timeslot). Then the cells are physically transferred during the next timeslot. Such segmentation of packets to cells efficiently uses backplane and simplifies the hardware design. Unless otherwise specified, for the rest of the chapter we will assume that an IP packet is segmented into cells before traversing the switch fabric. In the next few sections, we will study switched backplanes in detail starting with shared memory, and followed by crossbar, Clos networks, Beneš networks, and torus networks.

## 21.5 Shared Memory

Perhaps the simplest implementation of a switched backplane is based on a centralized memory shared between input and output ports. When packets arrive at the input ports, they are written to this centralized shared memory. When the packets are scheduled for transmission, they are read from shared memory and transmitted on the output ports. Figure 21.4 shows a shared memory switch. As shown, the memory is partitioned into multiple queues, one for

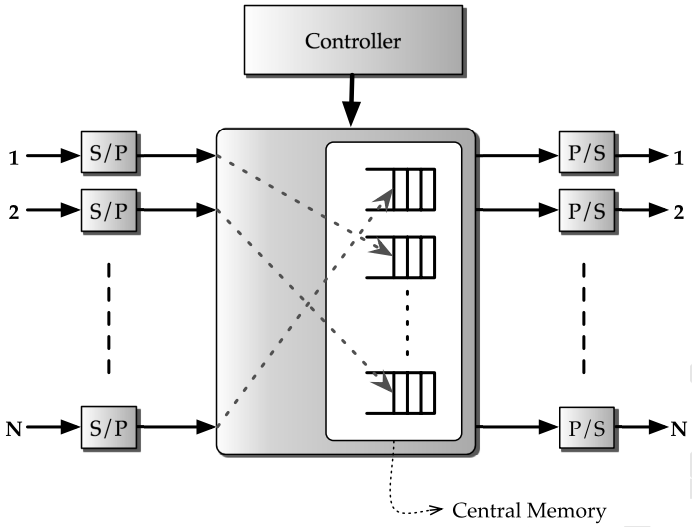


**FIGURE 21.4** A shared memory switch where the memory is partitioned into multiple queues.

each output port, and an incoming packet is appended to the appropriate queue (the queue associated with the output port on which the packet needs to be transmitted). The incoming bits of the packet are accumulated in an input shift register. Once enough bits equal to the width of the memory word are accumulated in the shift register, it is stored in memory. During output, the packet is read out from the output shift register and transmitted bit by bit in the outgoing link.

A related issue with each output port being associated with a queue is how the memory should be partitioned across these queues. One possibility is to partition the memory into fixed-sized regions, one per queue. While this is simple, the problem with this approach is that when a few output ports are oversubscribed, their queues can fill up and eventually start dropping packets. An alternative approach is to allow the size of each partition to be flexible. In other words, there is no boundary on the size of each queue as long as the sum of all queue sizes does not exceed total memory. Such flexible-sized partitions require more sophisticated hardware to manage; however, they improve the packet loss rate [699]. The rationale is that a queue does not suffer from overflow until no free memory remains; since outputs idle at a given time they can "lend" some memory to other outputs that happen to be heavily used at the moment.

Despite its simplicity, it is difficult to scale the capacity of shared memory switches to the aggregate capacity needed today. Let us examine why. First, a significant issue is the memory bandwidth. When the line rate $R$ per port increases, the memory bandwidth should be sufficiently large to accommodate all input and output traffic simultaneously. A switch with $N$ ports that buffers packets in memory requires a memory bandwidth of $2NR$ as $N$ input ports and $N$ output ports can write and read simultaneously. Hence, the memory bandwidth needs to scale linearly with the line rate.

Second, the access times of memory available are much higher than required. It is typical in most implementations to segment the packets into fixed-sized cells as memory can be

utilized more efficiently when all buffers are the same size [332]. If the cell size is $C$, the shared memory will be accessed every $C/(2NR)$ sec. For a switch with $N = 32$ ports, a cell size of $C = 40$ bytes, and a data rate of $R = 40$ Gbps, the access time required will be 0.125 nanosec. This is an order of magnitude smaller than the fast-memory SRAM, the access time of which is 5–10 nanosec at present.

Third, as the line rate $R$ increases, a larger amount of memory will be required. As indicated in Chapters 7 and 22, the routers need buffers to hold packets during times of congestion to reduce packet loss. The standard rule of thumb is to use buffers of size $RTT \times R$ for each link, where $RTT$ is the average roundtrip time of a flow passing through the link. For example, a port capable of 10 Gbps needs approximately 2.5 Gbits (= 250 millisec $\times$ 10 Gbps). If there are 32 ports in a router, the shared memory required is $32 \times 2.5$ Gbits = 80 Gbits, which would be impractical.

Finally, the time required to determine where to enqueue the incoming packets and issue the appropriate control signals for that purpose should be sufficiently small to keep up with the flow of incoming packets. In other words, the central controller must be capable of issuing control signals for simultaneous processing of $N$ incoming packets and $N$ outgoing packets.

Despite these disadvantages, some of the early implementations of switches used shared memory. These include the datapath switch [346], the PRELUDE switch from CNET [154], [175], and the SBMS switching element from Hitachi [199]. Commercially, some of the routers such as the Juniper M40 [629] use shared memory switches. Before closing the discussion on shared memory, let us examine a few techniques for increasing memory bandwidth.

### 21.5.1 Scaling Memory Bandwidth

With increasing link data rate, the memory bandwidth of a shared memory switch, as shown in the previous section, needs to proportionally increase. However, currently available memory technologies like SRAM and DRAM are not very well suited for use in large shared memory switches. While SRAM has access times that can keep up with the line rates, it does not have large enough storage because of its low density. On the other hand, DRAM is too slow, with access times on the order of 50 nanosec (which has increased very little in recent years).

In such scenarios, the standard tricks to increase memory bandwidth [293] are to use a wider memory word or use multiple banks and interleave the access. For a line rate of 40 Gbps, a minimum-sized packet of 40 bytes will arrive every 8 nanosec, which will require two accesses to memory; one to store the packet in memory when it arrives at the input port and the other to read from memory for transmission through the output port. If we were to use a DRAM with an access time of 50 nanosec, the width of the memory should be approximately 500 bytes (= $2 \times 50$ nanosec/8 nanosec $\times$ 40 bytes), in which 2 is for read and write. This means more than one minimum-sized packet needs to be stored in a single memory word. However, it is not possible to guarantee that these packets will be read out at the same time for output. This is because the packets could belong to different flows and QoS requirements might require that these packets depart at different times.

Alternatively, the memory can be organized as multiple DRAM banks so that multiple words can be read or written at a time rather than a single word. This type of organization is sometimes referred to as *interleaved* memory. In this case, for a line rate of 40 Gbps, we would need 13 (= $\lceil 50$ nanosec/8 nanosec $\times 2 \rceil$) DRAM banks with each bank required to be

40 bytes wide. When a stream of packets arrives, the first packet is sent to bank 1, the second packet to bank 2, and so on. The idea is that by the time packet 14 arrives, bank 1 would have completed writing packet 1. Assuming minimum-sized packets, if packet 1 arrives at time $t = 0$, then packet 14 will arrive at $t = 104$ nanosec ($t = 13$ packets $\times$ 40 bytes/packet $\times$ 8 bits/byte/40 Gbps). By this time, bank 1 would have finished writing packet 1 and would be ready to write packet 14. Actually, bank 1 would be ready at $t = 50$ nanosec. If so, then why a gap of 54 nanosec? It is because another 50 nanosec is needed for an opportunity to read a packet from bank 1 for transmission to an output port.

However, the problem with this approach is that it is not clear in what order the packets must be read. To satisfy QoS requirements, the packets might have to be read in a different order. This could lead to something called the "hot bank" syndrome where the packet accesses are directed to a few DRAM banks, leading to memory contention and packet loss. Another variation of this approach is to send the incoming packets to a randomly selected DRAM bank. The problem with this approach is that if the packets are segmented into cells, the cells of a packet will be distributed randomly on the banks, making reassembly complicated.

## 21.6 Crossbar

The simplest switched backplane is a crossbar. An $N \times N$ crossbar switch has $N$ input buses and $N$ output buses in a fully connected topology, as shown in Figure 21.5; that is, there are $N^2$ crosspoints, which are either on or off. Each crosspoint $(i, j)$, $0 \leq i < N$, $0 \leq j < N$ is controlled by a transistor that can be either turned on or off. When a line card $i$ wishes to transfer a cell to line card $j$, the crosspoint $(i, j)$ is turned on and the actual cell is transmitted. For instance, if the line card at port 2 has a cell destined for the line card at port 5, the transistor at crosspoint $(2, 5)$ is turned on to enable the data transfer, which is also shown in Figure 21.5. A crossbar is internally *nonblocking* as it allows all inputs and outputs to transfer packets simultaneously.

A crossbar switch is controlled by a centralized scheduler. The scheduler provides a schedule that indicates the inputs that need to be connected to the outputs at a given instant. If the cells arrive at fixed intervals, then the schedule can be computed *a priori*. Otherwise, the switch must compute the schedule on the fly. In such cases, the schedule is generated by

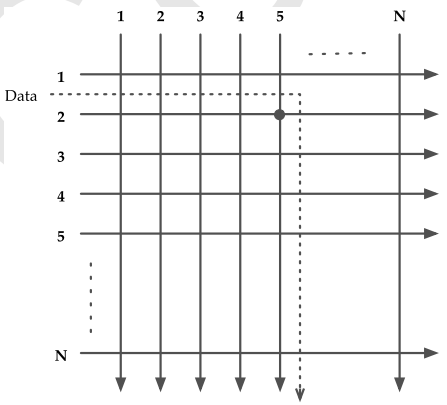


**FIGURE 21.5** A crossbar switch showing data flow from input port 2 to output port 5.

considering all the cells waiting to be transferred across the switch fabric. Then a configuration of crossbar is selected, ensuring that at a given instant each input is connected to at most one output and each output is connected to at most one input. Since the scheduler needs to turn the transistors on or off at the crosspoints, a control line is necessary that connects each of them to the scheduler.

The high performance of the crossbar switch is attributed to two factors. The links connecting the line cards to the switch fabric are simple point-to-point links and hence they can operate at high speeds. With the recent advances in semiconductor technology these links can operate as fast as 10 Gbps. The second factor is that the switch supports simultaneous connections of multiple inputs with outputs. The crossbar switch can close several crosspoints at the same time, thereby allowing the transfer of packets between multiple ports simultaneously. This greatly increases the aggregate bandwidth of the switch. However, the performance can be limited by several factors. First, some of the line cards might not have any data to send. Second, two or more line cards might want to send data destined for the same output port. In this case, only one of them can win and this, as a consequence, limits the data throughput since the other line card cannot send its data.

Before using a crossbar as a switch fabric, it is important to consider the advantages and potential drawbacks. It uses a simple two-state crosspoint (on or off), which is easy to implement. The modularity of the switch design allows large switches to be built by simply adding more crosspoints. Another significant advantage of a crossbar is the ability to natively support multicast. If an input port wishes to transmit its cell to multiple output ports, all the crosspoints corresponding to the input and output ports need to be turned on simultaneously. This allows each output port to receive a copy of the cell. For instance, assume that input port 5 in Figure 21.5 needs to multicast to output ports 2, 3, and 4. This is easily possible by turning on the crosspoints at $(5, 2)$, $(5, 3)$, and $(5, 4)$. Finally, the crossbar provides a low-latency path for connecting input to output compared to other switches since it has the lowest number of connecting points (just one).

The major disadvantage is that the cost of a $N \times N$ crossbar, measured in terms of the number of crosspoints, increases quadratically as $N$ increases. For doubling the number of inputs and outputs of a switch, the number of crosspoints need to be increased to four times the original. For instance, a $50 \times 50$ switch requires 2,500 crosspoints, whereas a $100 \times 100$ switch requires 10,000 crosspoints.

A crosspoint can be implemented using a transistor and hence it takes very minimal space in a chip. With the current chip fabrication technologies, millions of transistors can be easily accommodated in a chip. Consequently, the cost of crosspoints and the area it consumes might not be relevant for configurations $N < 1000$. For higher configurations of $N > 1000$, still the dominant cost is the number of crosspoints.

The number of pins that can be packaged on a chip affects the cost of a crossbar; however, a chip may include other components. Thus, there is a fixed cost associated with each chip. Because of pin issues and the fixed cost associated with pins, most implementations of crossbar switches are restricted to between 8 and 32 ports. The second potential drawback is the difficulty in providing guaranteed QoS. This is because the cells arriving at the switch must compete for access to the fabric with the cells already waiting at the input port and also with cells in other input ports bound for the same output port. The third disadvantage is that there is only a single path between an input and output—thus, any single crosspoint failure

would make this nonfunctional. Finally, even though multicast can be supported easily by connecting the input bus to all the output buses, scheduling becomes tricky and complex.

The interesting algorithmic aspect of crossbar switches is the scheduling algorithm. The objective of the scheduling algorithm is to compute pairs of input and output ports that maximize the number of cells transferred in a timeslot, by taking into consideration the cells that are waiting to be transferred. In the next few sections, we will focus on such crossbar scheduling algorithms. Our discussion begins with a simple and elegant scheduling scheme implemented in DEC's Gigaswitch [653], called "take-a-ticket."

### 21.6.1 Take-a-Ticket Scheduler

The basic idea behind the take-a-ticket scheduler is based on a ticketing scheme used in deli sandwich shops. In these shops, you first go to the counter, order your sandwich, and after payment, the cashier gives you a number that identifies your position in the queue. The cashier calls out the current number and you keep monitoring until your number is called. When your number is called, you can pick up your sandwich.

Similarly, each output port $Q$ in the switch maintains a distributed queue for all the input ports $P$ waiting to send to $Q$. The queue is actually not maintained at the output port $Q$. Instead, it is stored at the input ports using a ticket number mechanism. An input port $P$ that has a cell to send to output port $Q$ obtains a ticket from that port indicating its position in the queue. To obtain a ticket, port $P$ sends a request over a separate control bus to $Q$. In response, the output port $Q$ provides a queue number to $P$, again over the same control bus. The queue number indicates the position of $P$ in the output queue of $Q$.

Port $P$ keeps monitoring the control bus until its queue number is called out. Meanwhile, after port $Q$ finishes serving the current cell, $Q$ sends the next queue number it is willing to serve on the control bus. When $P$ notices that its number is being served, it places its cell on the data bus to $Q$. At this time, the crosspoint connecting port $P$ to port $Q$ is turned on by $Q$ to facilitate the cell transmission. As you can see, at any given instant, each input port works with only one cell and starts with the cell at the head of the queue.

Now we are ready to describe the algorithm, which consists of three distinct phases:

- **Request Phase:** This phase initiates the request for obtaining a ticket number. Each input port sends a request to the output port for which the cell at the head of the queue is destined via the control bus.
- **Grant Phase:** This phase assigns and communicates the ticket number. The output port on receiving the requests from the input ports assigns a ticket number based on order of arrival and sends the number to the input ports again via the control bus.
- **Connect and Transfer Phase:** In this phase, the output port indicates its willingness to serve a request by placing the ticket number on the control bus. When the input port recognizes it is being served, it initiates the actual flow of data and the cell is transferred to the output port. The output ports ensure the appropriate crosspoints are turned on for the transfer to take place.

The algorithm operates iteratively and in each iteration many cells (from different input ports to different output ports) are transferred simultaneously. Let us walk through an example of scheduling cells for a better understanding of the algorithm.

**Example 21.2** *Scheduling and transfer of cells using take-a-ticket scheduler.*

Consider a switch with four input ports denoted by $A$, $B$, $C$, and $D$ and output ports denoted by 1, 2, 3, and 4 as shown in Figure 21.6. It shows the input port $A$ has to send cells to output ports 1, 2, and 3, while port $B$ has three cells destined for output ports 1, 3, and 4. Port $C$ has three cells similar to port $B$. Finally, port $D$ has three cells bound for output ports 2, 3, and 4. In Figure 21.6 notice that the cells are numbered based on the output port to which they are bound. The algorithm operates iteratively and each iteration is referred to as a *round*.

In round 1, during the request phase, ports $A$, $B$, and $C$ send their requests to output port 1 as each has a cell at the head of the queue bound for that port. At the same time, port $D$ sends a request to port 2. Assuming the request from $A$ arrives at port 1 first, followed by $B$ and then $C$, the grant phase assigns the tickets $T_{11}$, $T_{12}$, and $T_{13}$ to $A$, $B$, and $C$, respectively. Similarly, port 2 responds with the ticket $T_{21}$ to port $D$, which concludes the request phase. In Figure 21.6, the requests and the ticket grants are represented using black lines with the direction arrows connecting the appropriate input and output ports. Now, output ports 1 and 2 broadcast the current ticket numbers being served, $T_{11}$ and $T_{21}$, on a separate control bus. As soon as the ports $A$ and $D$ see that their requests are being served, they transfer their

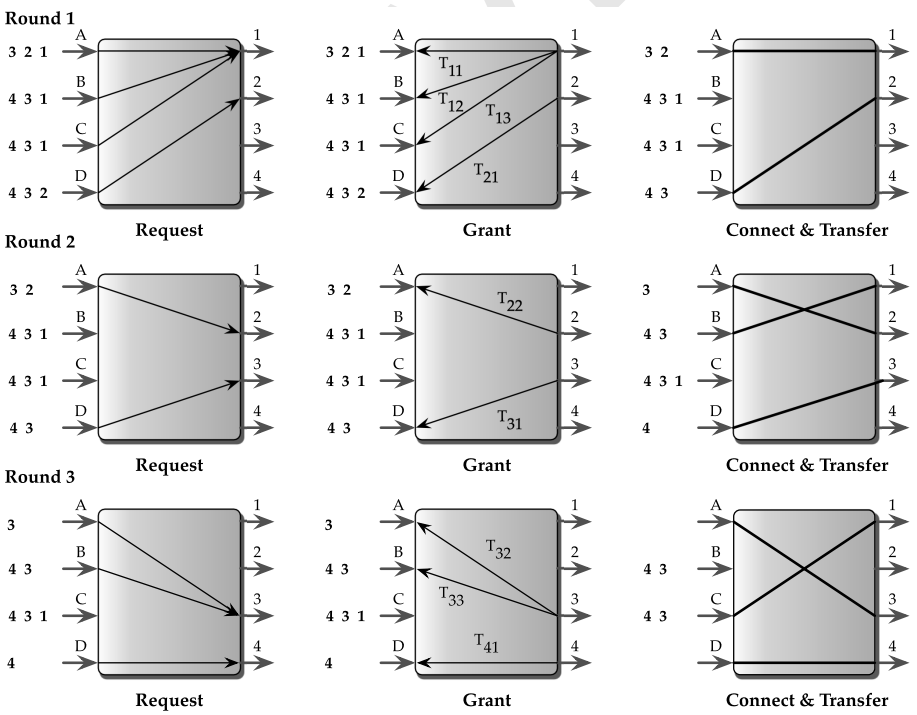


**FIGURE 21.6** Three rounds of a take-a-ticket scheduler in operation.

respective cells 1 and 2 to the corresponding output ports. Before the transfer, the crosspoints at $(A, 1)$ and $(D, 2)$ are turned on by the centralized scheduler. In Figure 21.6, the solid black lines without arrows illustrate the data transfer between the input and output ports. This concludes round 1.

In the next round (round 2), ports $A$ and $D$ send their requests to ports 2 and 4, respectively. Port $A$ is granted the ticket number $T_{22}$, while port $D$ gets $T_{41}$ as it is the first request for port 4. In parallel, the outport ports 1, 2, and 4 indicate their serving ticket numbers $T_{12}$, $T_{22}$, and $T_{41}$, respectively. Note that ticket $T_{12}$ was obtained in the previous round, but it is being served in the current round. During the connect phase, port $A$ transfers its cell to port 2, port $B$ to port 1, and port $D$ to port 4.

Now for round 3, ports $A$, $B$, and $D$ send their requests to port 3. If the requests from $A$, $B$, and $D$ arrive in order, ticket numbers $T_{31}$, $T_{32}$, and $T_{33}$ are granted, respectively. The current serving ticket numbers $T_{13}$ for port 1 and $T_{31}$ for port 3 are broadcast in a separate control bus. Once the crosspoints $(A, 3)$ and $(C, 1)$ are enabled, the cells are transferred from ports $A$ and $C$ to the outport ports 3 and 1, respectively. Finally, port $C$ gets a chance to transfer its first cell to port 1. The algorithm continues in this fashion for six rounds until the remaining cells are transmitted. ▲

The major advantage of this scheme is the ability to handle variable-length packets due to the nonexistence of any dependencies in the algorithm across ports. Each output port can asynchronously grant a ticket number whenever an input port requests it and similarly, can asynchronously broadcast the current serving ticket number once the current transfer is completed. Hence, it obviates the need to break up the original packets into cells of fixed size before transmitting and the need for reassembling at the output line card.

In addition to variable-length packets, the scheme has the advantage of using a small amount of memory to maintain the control state; two $\log_2 N$ bit counters at each output port, one for the current serving ticket number and the other for tracking the highest ticker number granted. Since the control state required is very small, DEC's implementation of the Gigaswitch [653] is scaled to 36 ports.

A major drawback of this scheme is *head-of-line blocking*, which limits the amount of parallelism, and as a result, reduces the throughput of the switch. Furthermore, since the schedulers at the output ports operate independently, it is hard to coordinate a subset of them for a multicast. If output ports must wait until all the requested ports become free, several opportunities for transferring packets from other input ports are lost and, hence, the throughput might be expected to be lower. Therefore, multicasts were handled separately in software running on a central processor.

### 21.6.2 Factors That Limit Performance

As can be seen from Example 21.2, after three rounds only eight cells have been transmitted. In each round, each input port can potentially tranfer a cell to an output port. With three rounds, there have been a total of 12 opportunities (4 input ports $\times$ 3 rounds) to transmit, but only 8 of them have been used. Ports $B$ and $C$ still have two cells left to transmit at the end of three rounds. This shows that the available parallelism has not been fully exploited.

To visualize this better, we can depict the order of the transmission of cells at the input ports using a timeslot diagram as shown in Figure 21.7. Each input port in Figure 21.7 is

| | Time Slot 1 | Time Slot 2 | Time Slot 3 | Time Slot 4 | Time Slot 5 | Time Slot 6 |
|---|---|---|---|---|---|---|
| **Input A** | *1* | *2* | *3* | | | |
| **Input B** | | *1* | | *3* | *4* | |
| **Input C** | | | *1* | | *3* | *4* |
| **Input D** | *2* | *3* | *4* | | | |

**FIGURE 21.7** Transfer of cells from input to output ports using take-a-ticket scheduling.

associated with six timeslots, representing the six rounds needed to transmit all the packets. Each round is numbered using the timeslot to which it belongs at the top. Each timeslot either contains an output port indicating the transfer of a cell to that port, or it is empty indicating that it is not used. Note that in our example, the cells are named based on the output port to which they are destined. From Figure 21.7 is it possible to infer that all the 12 input cells are transmitted out of 24 transmission opportunities, which amounts to just 50% utilization. We shall show in the later sections how other scheduling algorithms better exploit parallelism.

To determine the reasons for low utilization, we need to examine three types of blocking. The first type of blocking is called *head-of-line (HOL)* blocking and the second and third types are called *input* and *output blocking*, respectively. We shall see how HOL blocking reduces the throughput, while input and output blocking increase the delay of the cells passing through the fabric. In the next section, we discuss HOL blocking, the conditions under which it occurs, and how it impacts performance.

## 21.7 Head-of-Line Blocking

In a crossbar switch, all the cells waiting at an input port are stored in a single FIFO queue. Once the cell reaches the head of its queue, the scheduling algorithm considers it for transmission. However, this cell must compete with other cells that are at the head of the queues of other input ports but destined for the same output port. Such a tie is broken by the scheduler, which decides which cell will be transmitted next. Eventually, each cell will be selected and delivered to its output port. Using an FIFO queue at the input presents a problem since cells can be held by other cells ahead of them that are destined for a different output. This type of blocking is called HOL blocking. Since the scheduler when generating a schedule considers only the HOL cell, other cells behind it destined for different output ports are essentially blocked.

A good way to understand HOL blocking is to think of yourself in a car traveling on a single-lane road. You arrive at an intersection where you need to turn right. However, there is a car ahead of you that is not turning and is waiting for the traffic signal to turn green. Even though you are allowed to turn right at the light, you are blocked since you cannot pass on a

single lane. Now let us consider an analogous example where in some time slots a few input ports are unable to send to their cells.

**Example 21.3** *Cells blocked by the cell at the head of the queue.*

Going back to Example 21.2, in round 1 there were three cells destined for output port 1 from ports $A$, $B$, and $C$. The scheduler picks the cell from $A$ to transmit. Because of this decision, the queues at ports $B$ and $C$ are essentially stuck waiting for $A$ to complete. Hence, the cells 3 and 4 waiting behind cell 1 are blocked in ports $B$ and $C$. As a result, ports $B$ and $C$ lose their opportunity to transmit. ▲

As mentioned earlier, such a loss of opportunities occurs because the scheduling algorithm considers only the cells at the head of the queue. Sophisticated scheduling algorithms, as we shall see in later sections, take into account the cells behind the head of the queue and allow them to be transmitted. For example, while port $A$ is transmitting its cell to port 1, port $B$ and port $C$ can send their waiting cells 3 and 4 behind the head of the queue to port 3 and port 4, respectively.

The lost opportunities to transmit because of HOL blocking lead to lower throughput of a switch. Let us analytically derive the reduction in throughput by assuming an $N \times N$ switch. Furthermore, assume that all inputs always have a cell to transmit. Now consider the cells at the head of their queue, which in this case is only $N$. If the traffic is uniform, each cell could be destined to each output with an equal probability of $1/N$. If cells from different inputs are bound to the same output, then only one of them can get through and the rest will be blocked.

Now let us consider the probability that an output $O$ is idle. This is possible only when none of the inputs has a cell to transmit to $O$. The probability that an input does not choose output $O$ is $1 - 1/N$. Since an input not sending to output $O$ is independent of the other, the probability that all $N$ inputs are not sending to output $O$ is $(1 - 1/N)^N$. As $N$ increases, this expression converges to $1/e$, which is approximately 0.37. Hence the probability of the output being busy is $1 - 0.37 = 0.63$. Ideally, the throughput of the switch should be $NR$, where $R$ is the data rate at which the outputs operate. Since each output can be busy at most 63% of the time, the maximum expected throughput can be as much as $0.63NR$.

In the analysis, we assumed that the cells considered for scheduling in the current iteration are independent of the previous iteration. In reality, this is not the case. The cells that were not transmitted in the previous iteration have to be reconsidered in the current iteration. Hence the maximum throughput is even lower and is closer to 58% [352]. In the next few sections, we will take a detailed look at some of the solutions proposed to avoid or even eliminate HOL blocking and discuss their advantages and disadvantages.

## 21.8 Output Queueing

The initial set of solutions proposed for HOL blocking was based on the use of output queueing instead of input queueing. If a cell $C$ can be instantly transmitted to an output port without any queueing at the input, then it is impossible for that cell to block other cells behind it, thus eliminating HOL blocking. As a result, all the cells arriving at the input ports are immediately delivered upon their arrival to the output ports. In the worst case, cells at all the

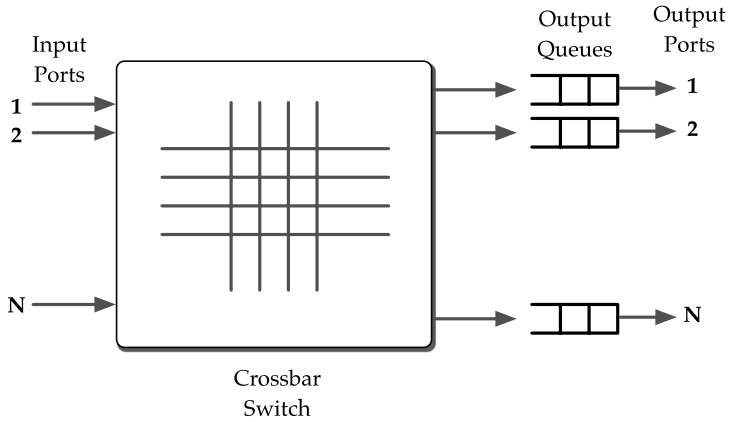


**FIGURE 21.8** An output-queued switch with queues operating $N$ times faster than input ports.

input ports may be destined for the same output port. Since the switch does not have any input buffers and if the packets are not to be dropped, the switch must deliver $N$ cells to the single output port and the output queue must store all of them in the time it takes for one cell to arrive at an input. Hence, the switch fabric and the output queue need to run $N$ times faster than the input ports. This can be expensive or difficult to implement. An output-queued switch is shown in Figure 21.8.

One way to reduce the cost of implementing output queueing is to use the *knockout* principle [755]. This principle states that if there are $N$ inputs in the switch, it is very unlikely that all $N$ cells received in any cell time are destined for the same output port. If the expected number of such cells is $S$, where $S < N$, then the fabric and output queue can be optimized to be run only $S$ times faster instead of $N$. This is less expensive and can be implemented using $S$ parallel buses.

When the expected case is violated by the arrival of a number of cells greater than $S$ in a cell time, the remaining cells must be dropped. In such cases, the packet losses will be fairly distributed among the input ports. For a variety of input distributions, it has been shown that $S = 8$ reduces the packet loss probability to one in a million.

There are two main difficulties in designing a switch based on the knockout principle. The first is how to implement a mechanism to choose $S$ cells when the number of expected cells exceeds $S$. A naive approach is to pair the input cells and choose the winner of the pair. When $S = 4$ and $N = 8$, we require four $2 \times 2$ concentrators to pair eight cells. The concentrators choose one cell randomly out of their two input cells and the winning cell is passed to the output while the loser cells are dropped. While this approach is simple and easy to implement, it is not fair in the sense that the cell drops are not evenly distributed among all the inputs ports. To obtain a better insight, consider the following. Assume two heavy traffic sources $M_1$ and $M_2$ are paired in a concentrator while another heavy traffic source $M_3$ is paired with an occasional traffic-generating source $M_4$. In this case, $M_3$ gets double the amount of bandwidth compared to $M_1$ and $M_2$, and hence the implementation is not fair. Now let us develop the concepts for a fair implementation by considering simpler cases before describing the final solution.

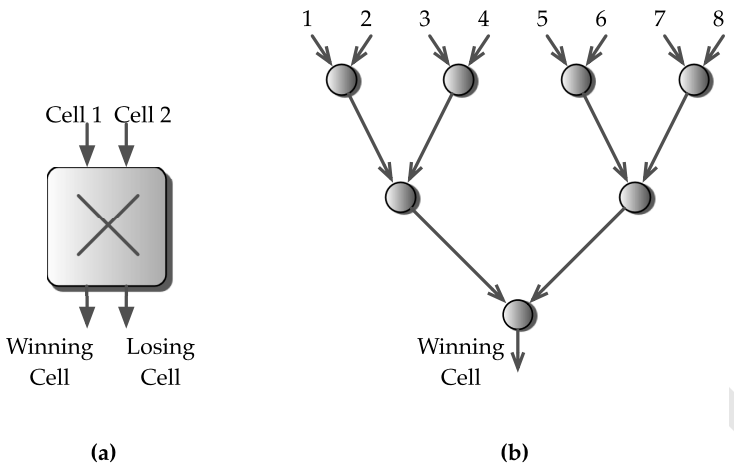


**FIGURE 21.9** Choosing a winning cell when (a) $N = 2$ and (b) $N = 4$.

- *Case* $S = 1$ *and* $N = 2$: For such cases, a simple $2 \times 2$ concentrator can be used to choose one winner randomly from the two input cells. However, the concentrator outputs both the cells, one of them being the winner and the other being the loser as shown in Figure 21.9(a). The loser is interesting in the general case.
- *Case* $S = 1$ *and* $N > 2$: In this case, one winner needs to be chosen among the cells $N > 2$. This is analogous to choosing the winner in an elimination tournament. As in the tournament, each cell is paired with another cell for the first stage using $N/2$ concentrators to identify $N/2$ winners. The rest of the $N/2$ cells are "knocked out," i.e., dropped. In the second stage, these $N/2$ winners are paired using $N/4$ concentrators and a new set of $N/4$ winners is found and so on until the root concentrator chooses the final winner. These concentrators form a tree referred to as the *knockout tree*. The knockout tree for $N = 8$ is illustrated in Figure 21.9(b), where concentrators are shown as nodes.

Now let us consider the general case where the switch needs to choose $S$ out of $N$ possible cells. A straightforward approach is to use $S$ knockout trees as shown in Figure 21.10, one for each winner. Unlike other approaches in which the loser cells are dropped, they are allowed to participate in the subsequent knockout trees to provide fairness. This is why the concentrators provide two outputs—one for the winner and the other for the loser. As can be seen in Figure 21.10, the first knockout tree takes as input all the $N$ cells and produces the first winner. All the $N - 1$ cells that lose enter the competition again at various stages in the second knockout tree, and so on. Note from the figure that winning cells will appear at different times. If all the winning cells need to output at the same time, some concentrators must be added that will be used as delay elements.

The second difficulty is the design of output queues that accept cells $S$ times faster than the speed of the output link. A naive approach is to implement an FIFO that accepts cells $S$ times faster than the output link. However, this is expensive to implement and it might not be worth the effort since these buffers cannot sustain the imbalance between input and output speeds for a longer period. If the objective is to sustain this for a shorter period, a cheaper

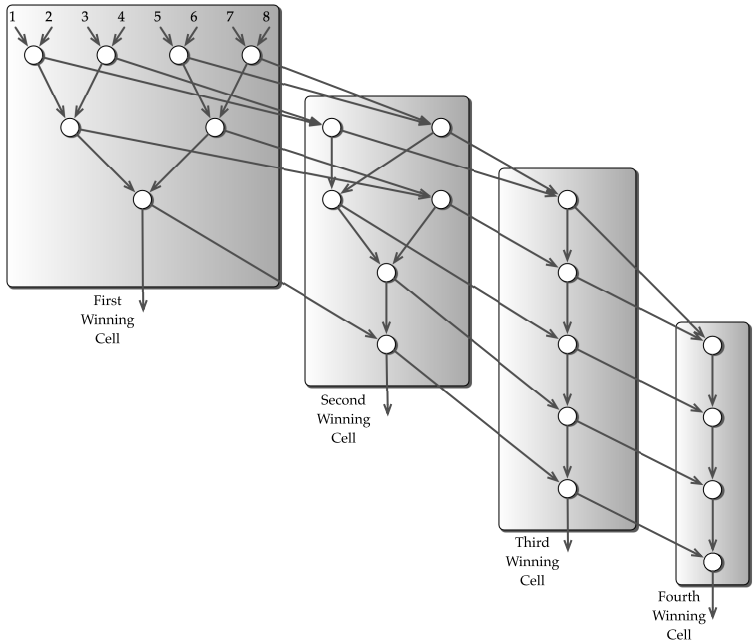


**FIGURE 21.10** Knockout trees for $S = 4$ and $N = 8$.

solution is to use $k$ memory banks and interleave the access to these banks. The knockout switch design [755] uses a shifter to spray the cells to these $S$ memory banks in round-robin order. The cells to be transmitted on the output link are read one at a time in the same order.

## 21.9 Virtual Output Queueing

To understand virtual output queueing, let us take a closer look at HOL blocking. The HOL blocking occurs because the input port is allowed to schedule only the cell at the front of the queue. If we can relax this restriction and allow the cells behind the head of the queue to be scheduled for transmission when the head is blocked, then HOL blocking can be eliminated.

Well, we might think that during scheduling we need to consider all the cells waiting in each input queue, which could be hundreds or thousands. If the state of each queue, (especially the cells waiting), has to be passed to the scheduler, it would become complex and consume too much memory, even assuming only a single bit per cell.

However, a few key observations eliminate the need for such complexity. First, note that all the cells waiting in the input queue can be destined for only $N$ possible outputs. Second, if the cells $C_1$ and $C_2$ from the same input queue $S$ are destined for the same output port $R$, then in order to maintain FIFO order, $C_1$ needs to be scheduled before $C_2$. Hence, it does not make sense to schedule $C_2$ before $C_1$. As a result, any cells other than the first cell destined to every distinct output port need not be considered for scheduling.

Therefore, the input queue at each port is split into multiple queues, one queue per output at each input port as shown in Figure 21.11. Hence, a total of $N^2$ input queues is needed for an $N \times N$ switch. This concept is called *virtual output queueing*, although the queueing

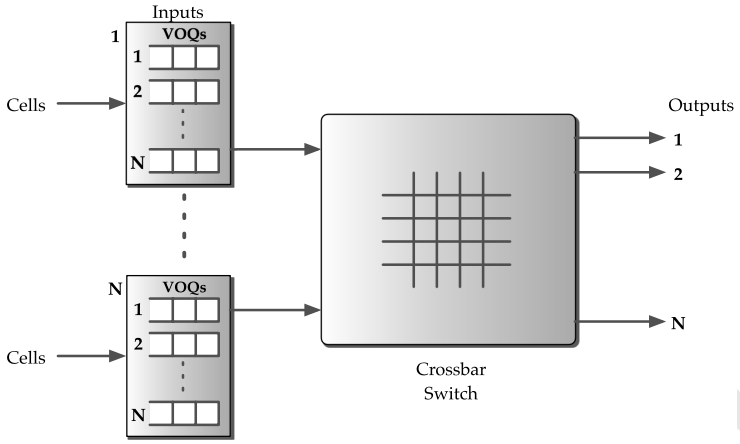


**FIGURE 21.11** A virtual output-queued switch with $N$ queues per input port.

physically occurs at the inputs and is referred to as VOQ for the rest of the chapter. With virtual output queues, scheduling to pair input ports to output ports becomes complex. Hence, in the next section, we digress to examine the theory behind how the scheduling problem can be mapped to a bipartite graph matching problem. It is followed by a discussion of two important scheduling algorithms, *parallel iterative matching* and *iSLIP*.

### 21.9.1 Maximum Bipartite Matching

The use of virtual queues complicates the problem of scheduling as it needs to take into consideration $N^2$ queues at the input ports and pair them with the output port in such a way that maximum transfer of cells occurs in a single time slot. Such a scheduling problem can be viewed as an instance of a bipartite matching problem as shown in Figure 21.12. The inputs and outputs form the nodes of a bipartite graph while the connections needed by queued cells from different inputs to various outputs are considered as edges in a bipartite graph. Additionally, there are no edges among the set of inputs or among the set of outputs; after all, the goal is to transfer cells from input ports to output ports, not from one input port to another input port.

A *maximum* match is one that pairs a maximum number of inputs and output ports together and there is no other pairing that matches more inputs and outputs. We can easily show that such pairings maximize the connections made in each timeslot and, as a result, maximize the instantaneous allocation of bandwidth. There are many algorithms for maximum bipartite matching, and the most efficient requires $O(N^{5/2})$ time [296]. A randomized algorithm [353] comes close to finding a maximum match, but it still requires $O(N + E)$ in an $N \times N$ bipartite graph with $E$ edges. The main drawbacks of these algorithms are that they are too complex to implement in hardware and too slow to be of any practical use. Furthermore, a maximum matching can potentially starve some input queues indefinitely. The following example illustrates such a possibility.

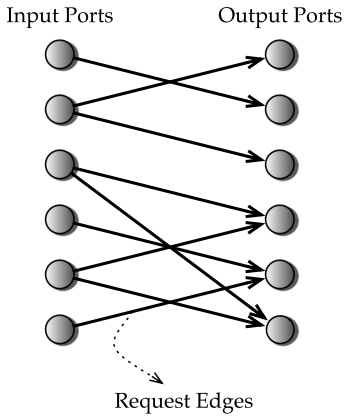


**FIGURE 21.12** Equivalence of scheduling and bipartite matching.

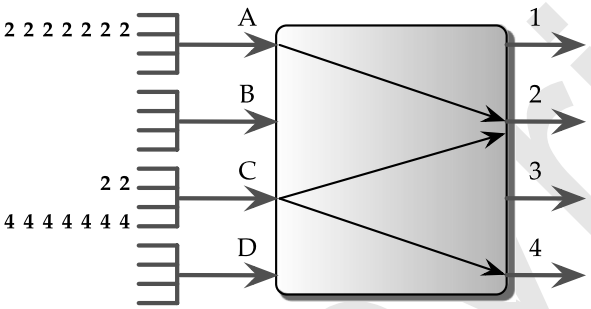


**FIGURE 21.13** Starvation in maximum matching.

**Example 21.4** *Starvation in a maximum matching.*

Consider a $4 \times 4$ crossbar switch with cells shown in Figure 21.13. Assume that a steady stream of cells keeps arriving at input $A$ destined for output 2. At the same time, input $C$ also gets a stream of cells, of which the majority are destined for output 4 and the remaining for 2. In this case, the maximum matching would always connect input $A$ with output 2 and input $C$ with output 4. This is because a maximum matching algorithm pairs as many inputs with outputs and no more pairing is possible. Hence, as long as there are cells at input $C$ destined for output 4, the cells queued at input $C$ destined for output 2 would never have an opportunity to be transmitted. ▲

Due to such drawbacks, practical scheduling algorithms attempt to find a *maximal* match. A maximal match is one for which new pairings of input to output cannot be trivially added; each node is either matched or has no edge to an unmatched node. A maximum match is maximal, but the reverse is not true. There may be a way to add more input and output pairings than a maximal match by reshuffling the pairings of input ports to different output ports. The following example illustrates the difference between maximal and maximum matching.

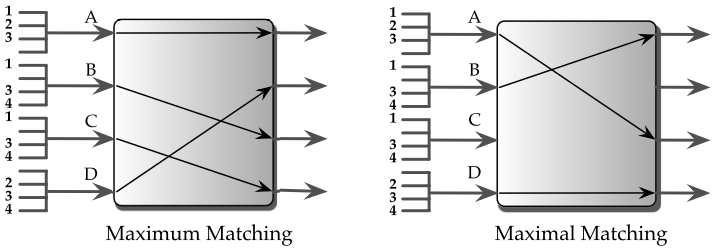


**FIGURE 21.14** Maximum versus maximal matching.

**Example 21.5** *Maximal versus maxiumum matching.*

Consider again a $4 \times 4$ crossbar switch. For the sake of discussion, assume that each input port has three cells to transfer to the output port as shown in Figure 21.14. Port $A$ has cells to transfer to output ports 1, 2, and 3. Similarly, cells bound for outports 1, 3, and 4 are waiting to be transferred to ports $B$ and $C$. Also, port $D$ has cells 2, 3, and 4 to be transferred to their respective output ports.

Now the pairings $\{(A, 1), (B, 3), (C, 4), (D, 2)\}$ constitute a maximum match since they provide maximum parallelism by connecting all inputs to outputs. However, the pairings $\{(A, 3), (B, 1), (D, 4)\}$ constitute a maximal match. This is because it is possible to obtain more parallelism by different pairings of input ports and output ports as in maximum matching. ▲

In practice, to communicate the scheduling needs, each input port must send a bitmap of size $N$ to the scheduler. In the bitmap, a bit at position $i$ indicates whether there is any cell destined for output port $i$. Since each input port sends a bitmap, the scheduler needs to process $N^2$ bits and for smaller values of $N$, say 32, this might not be enough bits to communicate to other components using buses or even to store in memory. Now let us turn our attention to a few scheduling algorithms that achieve close to maximal matches at very high speeds.

## 21.9.2 Parallel Iterative Matching

The key idea behind parallel iterative matching (PIM) [15] is the use of randomness to find a maximal match between input ports and outputs. The algorithm uses multiple iterations to converge quickly on a maximal match so that the number of cells transferred in a time slot is maximized. Before outlining the algorithm, let us attempt to understand it using the example in Figure 21.15.

**Example 21.6** *Scheduling and transfer of cells using parallel iterative matching.*

The algorithm starts with the request phase, where all the input ports $A$, $B$, $C$, and $D$ send requests to output ports for which they have a cell to forward to as shown in Figure 21.15. As we can see, $A$ sends requests to ports 1, 2, and 3 while ports $B$ and $C$ both send their requests to 1, 3, and 4. Finally, port $D$ communicates its request to ports 2, 4, and 3.

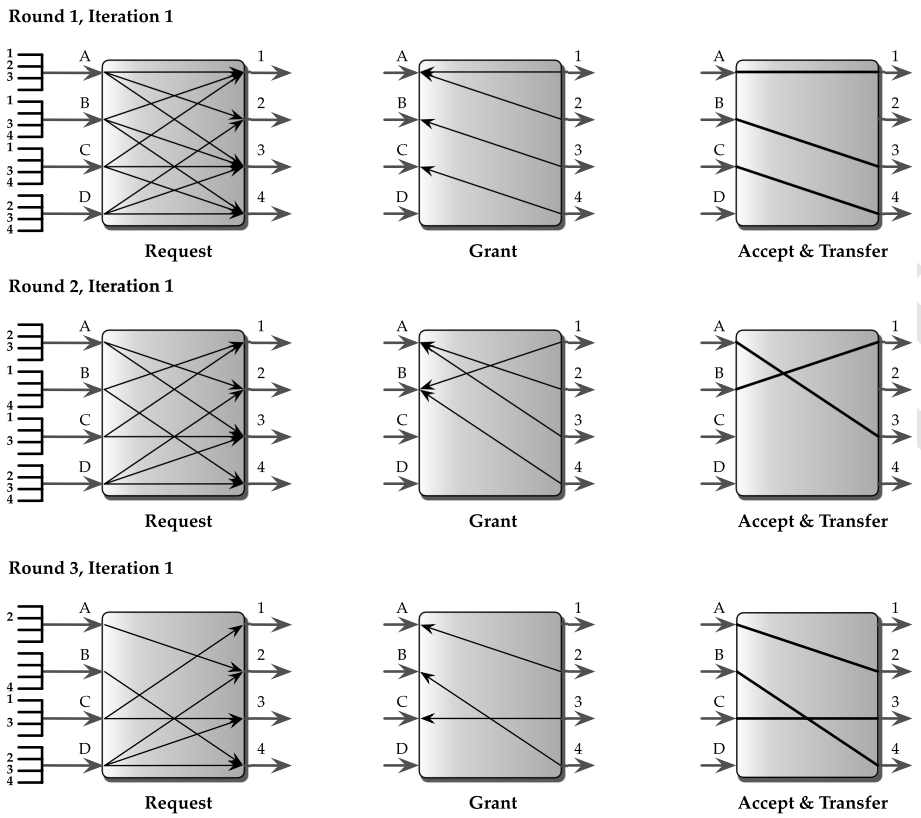


**FIGURE 21.15** Single-iteration PIM in operation.

Now observe that output port 1 gets three requests from $A$, $B$, and $C$. Of course, it can service only one port at a time. If that is the case, how will output port 1 choose which request to grant? A simple way is to choose randomly among the requests. Let us say port 1 chooses to serve $A$. Since port 2 also received requests from both $A$ and $D$, it has to break the tie by choosing one of them randomly, say port $A$ again. Finally, assume that ports 3 and 4 agree to serve port $B$ and port $C$, respectively. Since grants to requests are issued, this phase is referred as the *grant phase*.

Notice that port $A$ has been chosen by both ports 1 and 2, leading to input port contention. Now how do we break the tie? Well, again we can randomly choose one of them. Assume that port $A$ picks port 1. Hence, a third accept phase is needed in which each input port randomly chooses an output port. The final pairings are $(A, 1)$, $(B, 3)$, and $(C, 4)$. Port $D$ does not have a pairing since it lost among the choices made randomly. The appropriate crosspoints are turned on and cells are transferred. This concludes round 1 of the algorithm.

For round 2, again the requests for the remaining cells are sent from the input ports to the output ports. In this case, port $A$ sends to output ports 2 and 3 and port $B$ to output ports 1 and 4. Since the cells remaining in port $C$ have to be transferred to ports 1 and 3, it sends the requests to those ports. Port $D$ has not been able to transfer any cell in the first round and, hence, its requests are sent to ports 2, 3, and 4 as it has a cell bound for these ports.

Output ports 1 and 4 grant the requests from port $B$. On the same note, port $A$ receives a grant from output ports 2 and 3 while ports $C$ and $D$ receive none. Port $A$ accepts the request from port 2 and port $B$ from 1. Now the actual data transfer takes place, which concludes round 2. Note that an opportunity to transfer a cell from port $D$ to port 4 is lost because of randomness. ▲

The algorithm continues in this fashion until all the cells are transferred to their respective output ports. Figure 21.15 illustrates only the first three rounds of PIM. The remaining three rounds of the PIM operation are left as an exercise to the interested reader.

Based on the understanding of the operation, we are now ready to formally describe the algorithm. The PIM algorithm consists of three phases, similar to the take-a-ticket scheduler, which are described as follows:

- **Request Phase:** Each unmatched input sends a request to every output for which it has a buffered cell. This notifies an output port about the input ports that are interested in communicating.
- **Grant Phase:** If an unmatched output received any requests, the algorithm chooses randomly to grant a request. The output port notifies each input if its request was granted.
- **Accept Phase:** If an input receives any grants, it accepts one of them and notifies that output.

As we saw earlier, note that two or more input ports can request the same output port, leading to output port contention; the grant phase chooses one of them randomly. Similarly, input port contention can occur when two or more grants are issued to the same input port from multiple output ports; the accept phase chooses one of them randomly.

For Example 21.6, it may take as many as six time slots to transfer the cells, as shown in Figure 21.16. Each box in the figure corresponds to an opportunity to transfer a cell from an input port to an output port. In round 1, which corresponds to time slot 1, cells 1, 3, and 4 are transferred from their input ports $A$, $B$, and $C$, respectively. Similarly, during time slot 2, port $A$ transmits cell 3 and port $B$ transmits its cell 1 and so on. Out of 24 (6 *time slots* × 4 *ports*) opportunities to transmit cells, only 12 of them were used, which gives a throughput of 50%.

| | Time Slot 1 | Time Slot 2 | Time Slot 3 | Time Slot 4 | Time Slot 5 | Time Slot 6 |
|---|---|---|---|---|---|---|
| **Input A** | *1* | *3* | *2* | | | |
| **Input B** | *3* | *1* | *4* | | | |
| **Input C** | *4* | | *3* | *1* | | |
| **Input D** | | | | *2* | *3* | *4* |

**FIGURE 21.16** Transfer of cells using single-iteration PIM.

The low throughput can be attributed to a single iteration of the algorithm in each round, where some input ports might have been paired with output ports. However, some input ports still can be paired with other unpaired outputs. This is because two or more output ports can grant to the same input port while it chooses only one of them. The output ports whose grant is not accepted can be paired with some other unpaired input port. Hence, the algorithm is repeated in each round, retaining the matches made in the previous iterations to find matches for unpaired input and output ports.

**Example 21.7** *Scheduling and transfer of cells using two-iteration PIM algorithm.*

Continuing with Example 21.6, we saw that port *A* received two grants (one from port 1 and the other from port 2), before it chose port 1. Also note that port *D* did not have a pairing. Now since port 2 is free, it could very well have been paired with port *D*, as it has a cell destined for. If one more iteration of the algorithm is allowed with unmatched input and output ports, then the algorithm would have paired port *D* with port 2 and four cells could have been transferred in round 1. This is shown in Figure 21.17. The requests and grants are shown as solid black arrows; the matches in the previous iterations are shown as light black arrows. ▲

Hence, to maximize the pairings, the algorithm needs to be rerun preserving the matches from previous iterations. A follow-up iteration will increase the number of matches by 1, if maximum pairing has not occurred. Subsequent iterations cannot worsen the matches since the previous matches are retained. The cells transferred in each time slot using two-iteration PIM for Example 21.7 are shown in Figure 21.18. In contrast to single-iteration PIM, it requires only four time slots (or rounds). From a total of 16 opportunities to transmit, 12 are utilized

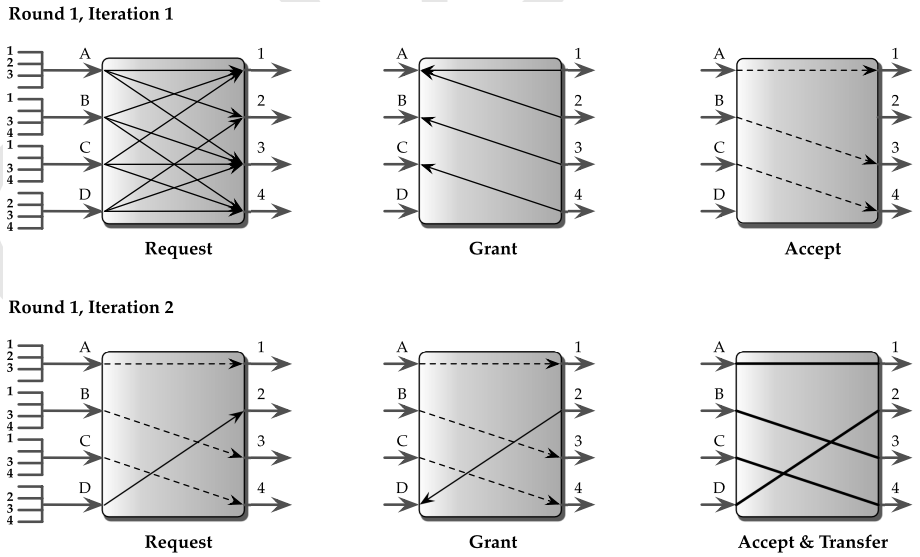


**FIGURE 21.17** Round 1 of a two-iteration PIM in operation.

| | Time Slot 1 | Time Slot 2 | Time Slot 3 | Time Slot 4 |
|---|---|---|---|---|
| **Input A** | *1* | *3* | *2* | |
| **Input B** | *3* | *1* | *4* | |
| **Input C** | *4* | | *3* | *1* |
| **Input D** | *2* | *4* | | *3* |

**FIGURE 21.18** Transfer of cells using two-iteration PIM.

and, hence, the throughput is 75% ($12 \times 100/16$), which is a 25% improvement over single-iteration PIM and "take-a-ticket" scheduling.

Since PIM requires a variable number of iterations to find a maximal match, it is important to understand the number of iterations it will take to converge. In the worst case, if all outputs grant requests to the same input, only one match will be made in a single iteration. If this sequence is repeated, it will take $N$ iterations to reach a maximal match. Hence, it is no faster than a sequential approach. In the best case, every output grants to a unique input, achieving a maximal match in one iteration. For the average case, it takes $O(\log_2 N)$ iterations to converge [15], independent of the pattern of requests. This is based on the observation that each iteration, on average, resolves 75% of the remaining unresolved requests.

For a large value of $N$, from an implementation perspective, it might not be possible to iterate until the maximal match is reached. This is because of the fixed amount of time required to schedule the switch. Hence, a small fixed number of iterations is used. Also, note that the algorithm might not necessarily forward cells through the switch in the order in which they arrive.

Since the algorithm randomly selects a request among contending requests, all the requests will eventually be granted, ensuring that no starvation occurs in any input queue. Hence, no state is needed to keep track of how recently a VOQ has been served. The algorithm begins all over, at the beginning of each cell time, independently of the matches that were made in the previous cell times. While the use of randomness does not require maintaining the state, it is expensive to implement, as a selection has to be made randomly among the requests of a time-varying set.

If a single iteration is used, the throughput of PIM is limited to approximately 63%, which is only slightly higher than a switch that uses FIFO [460]. The rationale for this is as follows. The probability that an input port will not be granted its request is $(1 - 1/N)^N$. As $N$ increases, the throughput tends to be $1 - 1/e \approx 63\%$. However, the algorithm typically finds a good maximal match after several iterations. Since each iteration requires the execution of three phases, the time for scheduling increases, which affects the rate at which the switch can operate. As a result, the switch provides lower throughput for moving packets between line cards. Hence, it is desirable to have a matching algorithm that uses one or two iterations to find a close enough maximal match.

### 21.9.3 iSLIP Scheduling

The iSLIP algorithm was designed to overcome the problems of complexity and unfairness in PIM. It is a simple iterative algorithm that achieves close to maximal matches in just one or two iterations. As discussed in the previous section, PIM chooses randomly among a competing set of requests or grants in order to provide fairness. However, iSLIP provides fairness using rotating pointers that track which input (output) port needs to be served next. These pointers allow the "winning" request or grant to be chosen among multiple contenders in a round-robin fashion. Even though these pointers are synchronized at the start of the algorithm, they tend to desynchronize, which results in maximal matches as time progresses. For the sake of discussion, let

$I_i$ = Accept pointer at input port $i$.

$O_j$ = Grant pointer at output port $j$.

The algorithm starts by initializing $I_i$ for all values of $0 < i < N - 1$ to the first output port. Similarly, $O_i$, for all values of $0 < j < N - 1$ is initialized to the first input port. The algorithm is invoked at the start of each time slot and uses single or multiple iterations to match input ports that have cells to transmit to output ports. The following steps are in the iteration:

- **Request Phase:** Each input port sends a request to every output port for which it has a cell queued in its VOQs.
- **Grant Phase:** When an output port $j$ receives one or more requests, it chooses the lowest input port number that is equal to or greater than $O_j$. After choosing the input port to serve, the output port notifies each input whether its request has been granted.
- **Accept Phase:** When an input port $i$ receives multiple grants, it chooses to accept the lowest output port number that is equal to or greater than $I_i$. Once the input port accepts a grant, $I_i$ is incremented to the next output port in a circular order. In other words, if input port $i$ accepts a grant from output port $X$, then $I_i$ is updated to $(X + 1)$ mod $N$. Pointer $O_j$ of output port $j$ is also incremented in circular order to the next input port beyond the granted input. If the accepted input port is $Y$, then $O_j$ is assigned the value of $(Y + 1)$ mod $N$. These pointers are updated only after the first iteration, not in subsequent iterations.

As mentioned earlier, the steps of the iteration are repeated a predefined number of times with unmatched inputs and outputs, retaining the matches from the previous iterations. Finally, the cells are transferred from the matched input port to the output port. To obtain a concrete understanding, let us use an example of how iSLIP schedules a set of cells.

**Example 21.8** *Scheduling and transfer of cells using iSLIP.*

Consider scheduling the same set of inputs as in Example 21.6 using two-iteration iSLIP. The first three rounds of operation of iSLIP are shown in Figures 21.19, 21.20, and 21.21. Each round consists of two iterations before the actual data transfer occurs. As can be seen from

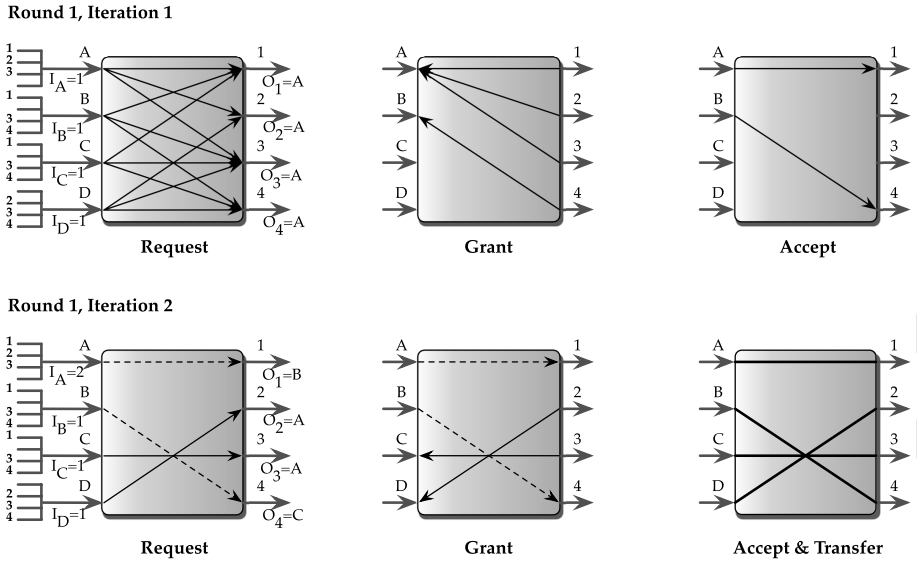


**FIGURE 21.19** Two-iteration iSLIP scheduling in operation—round 1.

the figures, each output port is associated with a *grant* pointer $I_i$ for $1 < i < N$ and all of them are initialized to the first input port $A$. Similarly, each input port maintains an *accept* pointer $O_j$ for $1 < j < N$ and is initialized to the first output port 1.

In Figure 21.19, each input port sends requests to each output port for which it has a queued cell. According to the algorithm, each output port grants to the lowest input equal to or greater than its grant pointer. For example, when output port 1 receives requests from input ports $A$, $B$, and $C$, it grants to $A$ since its pointer $O_1$ points to $A$. Similarly, output ports 2 and 3 also grant to $A$ as their grant pointers contain $A$ while the output port 4 grants to input port $B$.

As the grants are communicated to the input ports, $A$ finds that it received three grants, one each from ports 1, 2, and 3. The accept pointer $I_A$ indicates that $A$ can accept grants from port 1 or greater. Therefore, it chooses to accept the grant from port 1 and rejects the grants from ports 2 and 3. Similarly, $B$ accepts the single grant from port 4. The accepted grants are communicated to the respective output ports, $A$ to 1 and $B$ to 4, as shown in the upper rightmost column in Figure 21.19. At this time, the grant pointers $O_1$ and $O_4$ are updated to $B$ and $C$, respectively. Similarly, accept pointers $I_A$ and $I_B$ are updated to 2 and 1, respectively. Note that the value of $I_B$ continues to be output port 1 since the next port after port 4 is 1 in circular order.

Observe that the grant pointers $O_2$ and $O_3$ are not incremented despite granting for port $A$. This is because their grants are not accepted by port $A$. If these grant pointers are incremented, even after the grant is rejected, they might be synchronized in lock-step (for details refer to [460]), thereby reducing the number of cells transmitted in a time slot.

At the end of the first iteration, a match of size only 2 has been achieved. It can be further improved by a second iteration, shown in the lower half of Figure 21.19. The second iteration begins with unmatched inputs only requesting unmatched outputs. Input ports $C$ and $D$ send

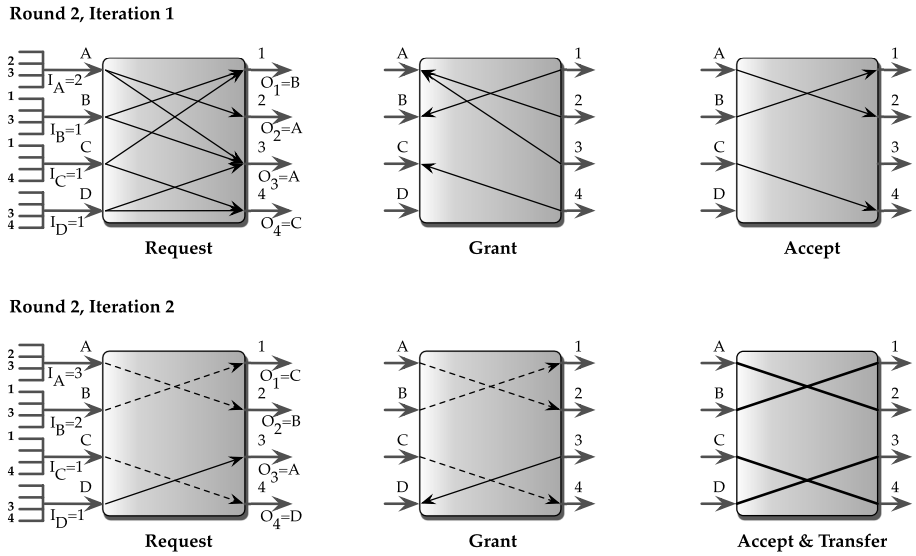


**FIGURE 21.20** Two-iteration iSLIP scheduling in operation—round 2.

requests for ports 3 and 2, respectively, which are granted. Unlike the first iteration, the grant and accept pointers are not incremented to avoid starvation [460].

Thus, the accept pointer at $C$ and $D$ remains at $A$ while the grant pointers at 2 and 3 remain at 1. At the end of the second iteration, the paired inputs and outputs are $(A, 1)$, $(B, 4)$, $(C, 3)$, and $(D, 2)$. The crosspoints are turned and the actual data transfer occurs. The data transfers are shown as thick solid lines in the bottom rightmost switch in Figure 21.19.

The next two rounds of the operation of iSLIP are shown in Figures 21.20 and 21.21. The reader can trace through the algorithm and identify the cells transferred in each round. By the end of three rounds, all the cells except cell 3 at input port $B$ are transferred to the respective output ports. The remaining cell can be transferred in the fourth round. ▲

How do the accept and grant pointer break away from each other? Observe the first row in Figure 21.20. At the start of round 2, since output port 1 has been granted to input port $A$, it moves on to provide priority for serving ports beyond $A$, in this case $B$. Hence, even if port $A$ has another cell destined for port 1 (unlike our example), port 1 will grant only the requests for port $B$ and beyond.

The algorithm requires that $2N$ pointers be maintained, one for each input port and one for each output port. Each pointer should have $\log_2 N$ bits to address the ports from 0 to $N - 1$.

The time slot at which the cells depart from the input port for Example 21.8 is shown in Figure 21.22. As shown, all the cells are transmitted in four time slots. While comparing it with two-iteration PIM, it might appear that iSLIP performs only as well as PIM even with two iterations (both consume four time slots). This is due to the startup penalty of iSLIP because of the synchronization of pointers. Once the pointers are desynchronized, iSLIP performs well with just a single iteration. Also, notice that despite the startup penalty, iSLIP uses all the

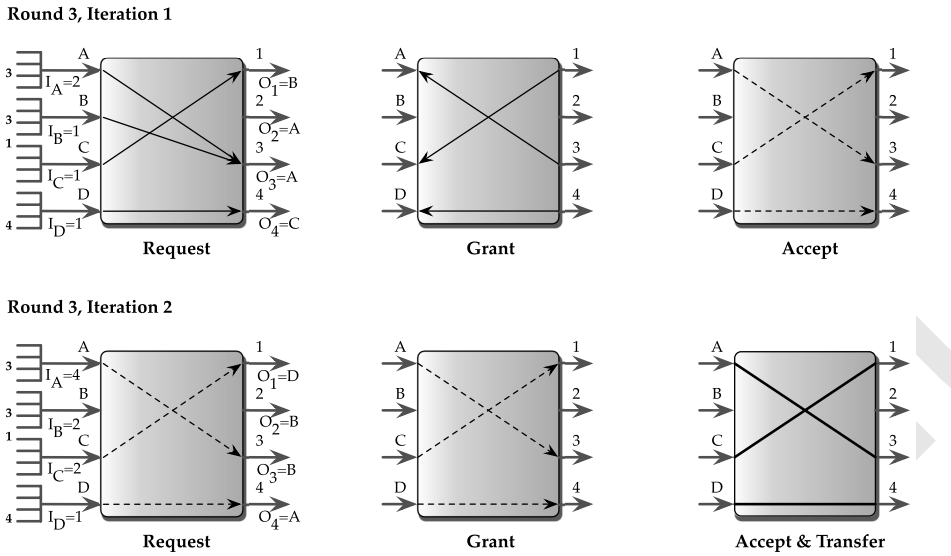


**FIGURE 21.21** Two-iteration iSLIP scheduling in operation — round 3.

| | Time Slot 1 | Time Slot 2 | Time Slot 3 | Time Slot 4 |
|---|---|---|---|---|
| **Input A** | *1* | *2* | *3* | |
| **Input B** | *4* | *1* | | *3* |
| **Input C** | *3* | *4* | *1* | |
| **Input D** | *2* | *3* | *4* | |

**FIGURE 21.22** Transfer of cells using two-iteration iSLIP.

transmission opportunities in the first two rounds compared to PIM. Most of the commercial implementations of iSLIP use only a single iteration.

### 21.9.4 Priorities and Multicast in iSLIP

For many Internet applications such as VoIP and video, their traffic must be scheduled through the router ahead of lower-priority traffic to guarantee latency and jitter requirements. For this the switch fabric must let this traffic pass as quickly as possible. Similarly, applications like video conferencing require support for their packets to be replicated at the router for multicasting. Since the growth of such traffic is on the rise, first-class support for multicasting is becoming more important.

The iSLIP algorithm can easily accommodate priorities using a separate VOQ for each priority at each input port per output port. For instance, if there are four priorities that need to be supported on a 16 × 16 switch, each input port needs 64 (4 priority levels × 16 out-

put ports) VOQs. Also, for each output port $j$ and priority $k$ a separate grant pointer $O_j^k$ is maintained. Similarly, an accept pointer $I_i^k$ is also maintained at each input port for each priority. Scheduling the prioritized traffic necessitates performing the original iSLIP algorithm on each input and output port on the highest priority level for which there are cells to be transmitted.

To be precise, each output port accepts a request from the highest priority request it receives; in addition, each input port also accepts a grant from the highest priority level it sees. Consider a situation in which an input port $I$ issues a request for priority level 1 for output port 2 and another request for priority level 3 for output port 3. If both requests are granted, $I$ chooses the one with the highest priority grant. Note that the choice is not based on accept pointers since they are at different priorities; instead it is based on priority. Once the grant is accepted in the first iteration for a priority $k$ between input port $I$ and output port $O$, the corresponding accept and grant pointers are incremented.

A straightforward approach for implementing multicast is to replicate the input cell and transmit a copy to the output port for every time slot for the respective output ports. But the disadvantage is that the same input cell competes with other cells multiple times for the switch. This reduces the available switch bandwidth for other traffic at the same input. However, as seen in Section 21.6, crossbar naturally supports multicasting, which can be used to achieve higher throughput.

The iSLIP can be extended to support multicast, and the variant is referred to as ESLIP [459]. To accommodate multicast, ESLIP includes an additional queue per input port. The use of a single queue might introduce HOL blocking for multicast. Indeed, it will. For instance, consider cell $C_1$ destined for outputs $O_1$ and $O_3$, which in the queue occurs before cell $C_2$ bound for outputs $O_2$ and $O_4$. If outputs $O_1$ and $O_3$ are busy, the cell $C_1$ will block cell $C_2$ even if outputs $O_2$ and $O_4$ are idle. If HOL blocking needs to be avoided for multicast, it will require a queue for each subset of output ports, which might not be practical ($2^{16}$ for 16 ports). The set of output ports to which an input cell needs to be replicated is called its *fanout*. For instance, if input port $I$ contains a cell that needs to be copied to output ports 1, 2, and 4, then its fanout is 3.

Now the multicast traffic can be scheduled in two ways. In the first approach, referred to as *no fanout-splitting*, all copies of the input cell are transmitted in a single time slot. In this case, if there is contention for one of the output ports, none of the copies is transmitted and it has to be retried in some other time slot. In the other approach called *fanout-splitting*, the cells to be multicast are delivered to output ports over multiple time slots. The cells that are not transmitted due to contention in some output ports will continue to try in the next time slot. Studies [287], [300], [570] show that fanout-splitting leads to higher throughput with a slight increase in implementation complexity.

ESLIP implements a variation of fanout-splitting in which a particular input is favored until it completes the transmission of its fanout completely before the next input cell, which is different from multiple inputs competing for output ports and transmitting their fanouts partially. This version of fanout-splitting is implemented in ESLIP using a shared multicast grant pointer that is different from the separate grant and accept pointers per port for unicast.

When a mix of multicast and unicast requests arrives at an output port, how does the output port choose which one to grant? ESLIP solves this problem by giving preferential

treatment to unicast and multicast in alternate time slots. Let us consider an example of how a mix of unicast and multicast traffic is handled.

**Example 21.9** *Scheduling of unicast and multicast traffic using ESLIP.*

Consider two input ports $I_1$ and $I_2$ that have a unicast cell and a multicast cell to be transmitted, respectively. Let us assume that the unicast cell from $I_1$ is destined for output port $O_4$ and the multicast cell from $I_2$ is destined for output ports $O_1$, $O_2$, and $O_4$. Also assume that the switch provides preference for unicast traffic in odd time slots and for multicast traffic in even time slots.

If the current time slot is odd, then the output $O_4$ will grant the request for unicast from $I_1$. Since the outputs $O_1$ and $O_3$ do not have any requests for unicast traffic, they will choose to grant the multicast request to the first port that is greater than or equal to the current shared multicast grant pointer. Assuming that $I_2$ is chosen, the outputs $O_1$ and $O_3$ will grant request $I_2$.

Unlike unicast, all the multicast grants are accepted by input $I_2$. Also, the shared multicast grant pointer is not increased beyond $I_2$, since $I_2$ has not yet completed its fanout. In the next time slot, when the multicast traffic gets priority, the grant will be issued to $O_4$, which completes the fanout of $I_2$. Then the shared multicast grant pointer is increased by one past $I_2$. ▲

## 21.10 Input and Output Blocking

So far, our attention has been focused on HOL blocking and on how it affects the throughput of the crossbar; various solutions that eliminate HOL blocking have been outlined. But the delay experienced by a packet inside a router as it travels from the input interface through the crossbar to the outgoing interface can be unpredictable. However, with the growth of the Internet, large amounts of multimedia and delay-sensitive traffic require that packets arrive at their destination within a predictable time.

Routers, as we shall see in Chapter 18, in their outgoing interfaces employ an output link scheduler, which determines the exact time at which each packet needs to be transmitted. Such output scheduling alone cannot guarantee a predictable delay for the packets forwarded by the router. Within the router, the packets could experience unpredictable delays as they try to pass through the fabric. These delays in the crossbar can be attributed to input and output blocking.

Input blocking occurs when there are multiple input cells in different VOQs on the same input port contending for the fabric. Since the scheduler selects only one cell from a VOQ to be served in a time slot, the other cells are blocked. For a better understanding, consider the VOQs shown in Figure 21.11. Assume that the scheduler selects a VOQ to be serviced by the scheduler at one input. The other nonempty VOQs at the same input must wait until a later time slot for service. In fact, it will be difficult to predict when a nonempty VOQ will be scheduled to receive service. This is because new cells might keep arriving for VOQs in other inputs in every time slot, changing their occupancy, and the scheduler tries to pair VOQs in input ports with output ports to achieve a maximal match so that throughput is maximized. Hence the VOQ needs to compete with other VOQs that might block it for an unpredictable number of time slots.

To understand output blocking, consider two cells in different input ports destined for the same output port. Since each output line in a crossbar switch can connect to one input during a cell time, only one cell can be transferred. The other cell will be *blocked* until a later cell time. In such cases, we say that *output blocking* has occurred since the transfer of a cell bound to an output blocks other cells bound for the same output. Similar to input blocking, in output blocking the time when a cell will be delivered to its output can also be unpredictable.

**Example 21.10** *Unpredictable delay due to input blocking and output blocking.*

Consider the set of input cells in VOQs shown in Example 21.8. Assume that all the cells arrived at their VOQs at the same time. At the end of round 1, input port $B$ is able to transfer its cell bound for output port 4. Note that port $B$ contains other cells bound for ports 1 and 3, which are blocked as it is serving the cell for port 4. These cells are transmitted later in round 2 and round 4. Even though all the cells arrived at the same time, the cell for port 4 did not experience any delay, but the cells for ports 1 and 3 were delayed by one time slot and three time slots, respectively.

For output blocking, consider the cells bound for port 4 at input ports $B$, $C$, and $D$. In round 1, port $B$ is allowed to transmit to port 4 while ports $C$ and $D$ were blocked since the output can serve only one input port in a time slot. Port $C$ gets a chance to transmit in round 2, thus experiencing a delay of one time slot. In round 3, port $D$ sends its cell to port 4, and hence the delay is two time slots. As you can see, the delay is unpredictable in both cases and to a large extent depends on the cells in other VOQs and the scheduling algorithm. ▲

Two techniques to control the delay a packet experiences through the crossbar switch have been described in [459]. The first technique is to segregate packets into different priority classes based on the delay requirements. Higher-priority packets that belong to delay-sensitive traffic are given preferential treatment access to the switch. While prioritization does not eliminate input and output blocking, it mitigates the delay experienced when higher-priority packets are affected by lower-priority packets. Results [459] indicate that if the traffic in the higher-priority class is kept relatively small, the delay is close to zero. Therefore, the high priority cells will be transferred to the output port with a fixed but negligible delay. Scheduling of higher-priority cells ahead of lower-priority cells can be incorporated as described in Section 21.9.4.

The second technique is to run the switch faster than the external input and output links (as described in Section 21.8). For instance, when the switch is run $S$ times as fast as the external line, $S$ cells can be transferred from each input port to each output port during each time slot. Such a speedup delivers more cells per time slot and, hence, reduces the delay of each cell through the switch. In an ideal case, every input cell can be transferred to its output port immediately upon arrival when the switch runs with sufficient speedup. The worst-case scenario is when all input ports need to transmit cells to the same output port, which requires a speedup of $N$, in theory. As shown in Section 21.8, this is impractical.

## 21.11 Scaling Switches to a Large Number of Ports

As the Internet traffic enjoys tremendous growth, network operators require routers that are capable of moving more than a few terabits per second of traffic. Since most of the traffic

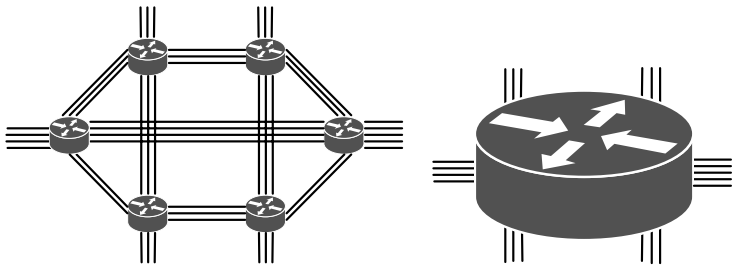


**FIGURE 21.23** Replacing a cluster of routers (a) with a large-capacity scalable router (b).

passing through the router has to traverse the internal switched backplane, there is a need for large-capacity switches. Unfortunately, the switched backplanes that we studied so far cannot scale to large capacity to satisfy current and expected future routing demands. As a result, a number of routers are interconnected with numerous links in a cluster-like form as shown in Figure 21.23(a). In these architectures, routers need to employ expensive line cards to connect links to other routers in the cluster. These links carry intracluster traffic rather than the revenue-generating user traffic. Thus, it has been proposed that such router clusters be replaced by a single scalable router (see Figure 21.23(b)). Such large routers are advantageous as they save the cost of numerous line cards and expensive links. Furthermore, there will be fewer such routers that need to be configured and managed.

A key requirement for such a router is the support for a large number of line cards. This is required since the router needs many links to connect access routers, PoPs, and WAN. Hence, the internal switched backplane needs to support a large number of ports. Another key requirement is the need to accommodate high-speed links as their bit rates can be as high as 40 Gbps because of recent advances in optical technologies. As a result, the backplanes need to scale in two orthogonal dimensions: *number of ports* and *link speed*. In the next few sections, we discuss switching architectures that scale to a larger number of ports. In Section 21.14, we examine in detail how switches can be scaled for higher link speeds.

## 21.12 Clos Networks

A single-stage network like a crossbar can be scaled to a larger number of ports using a naive approach. Simply build a larger crossbar by interconnecting smaller crossbar chips. However, the cost of the crosspoints still increases quadratically. In such a composite switch, other dominant costs include the cost of the pins and the number of links connecting these chips. Hence, these switches tend to be expensive for a large number of ports. Furthermore, a crosspoint failure isolates an input from the output, making it less robust for failure.

Clos [146] described a method to reduce the number of crosspoints, in addition to providing more than one path between input and output ports, using multistage switching techniques. A Clos (pronounced as "Close") network is a three-stage network constructed by using smaller crossbar switches as shown in Figure 21.24. Since the number of inputs and outputs is the same in the Clos network shown in Figure 21.24, it is sometimes referred to as a *symmetric Clos* network. The first stage divides the inputs into smaller groups of $n$ each and

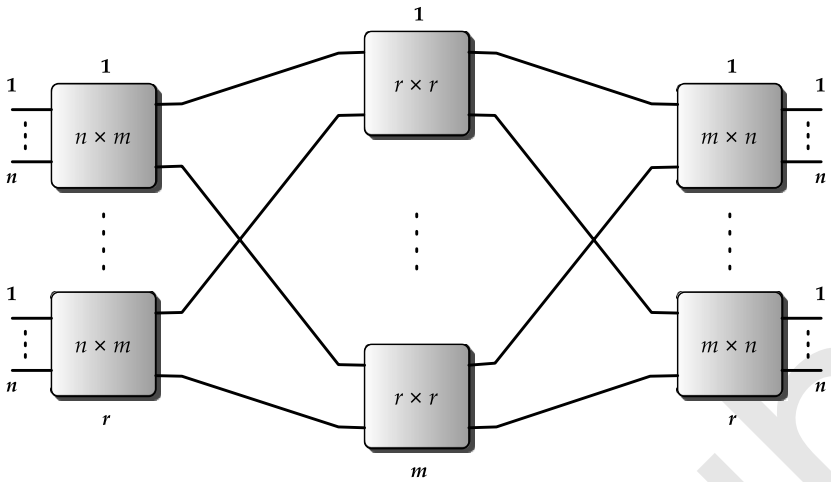


**FIGURE 21.24** A $(m, n, r)$ Clos network with $m$ middle switches and $r$ input and output switches.

switches each group to the middle stage. It uses a simple crossbar switch of type $n \times m$ for each group. Assuming there are $r$ groups, then $r$ such switches are needed for the first stage. The middle stage uses $m$ switches of type $r \times r$, and each switch has one input link from every switch in the first stage in order. In other words, output $j$ of switch $i$ in the first stage is connected to input $i$ of switch $j$ in the midde stage. Similarly, each output link of a middle stage switch is connected to one input link of a switch in the final stage, again in order. The final stage involves $r$ switches of type $m \times n$ connecting all $m$ middle switches to its outputs. As a result, there exists $m$ distinct paths for a given pair of input and output ports through the Clos network.

Often, such a Clos network is referred to by a triple $(m, n, r)$, where $m$ represents the number of switches in the middle stage, $n$ denotes the number of input (output) ports on each input (output) switch, and $r$ is the number of input or output switches. For a router that switches packets from $N$ input line cards to $N$ output line cards, we need a $(m, n, \lceil N/n \rceil)$ Clos network. A Clos network need not be restricted to only three stages. Clos networks with an odd number of stages more than three can be recursively constructed by replacing the switches in the middle stage with a three-stage Clos network.

Recall that a switch is nonblocking if there is no configuration of connections that can prevent the addition of a new connection between an idle input $i$ and an idle output $o$. Now the question is, can a Clos network be nonblocking? It might appear that perhaps it is not nonblocking as an input switch has at most $m$ connections to the middle stage and each middle stage switch has at most one connection to an output switch. This might be true for small values of $m$ where an input switch $I$ might not be able to find a path to a middle switch with a free link to an output switch $O$.

However, Clos showed that when $m \geq 2n - 1$, the resulting Clos network is nonblocking. The proof for Clos's observation is relatively simple. Consider a scenario where an input port $i$ that has been idle so far wants to transmit to an idle output port $o$. Assume that $i$ belongs to input switch $P$ and $o$ to output switch $Q$. If $P$ is an $n \times m$ switch, in the worst case, at most

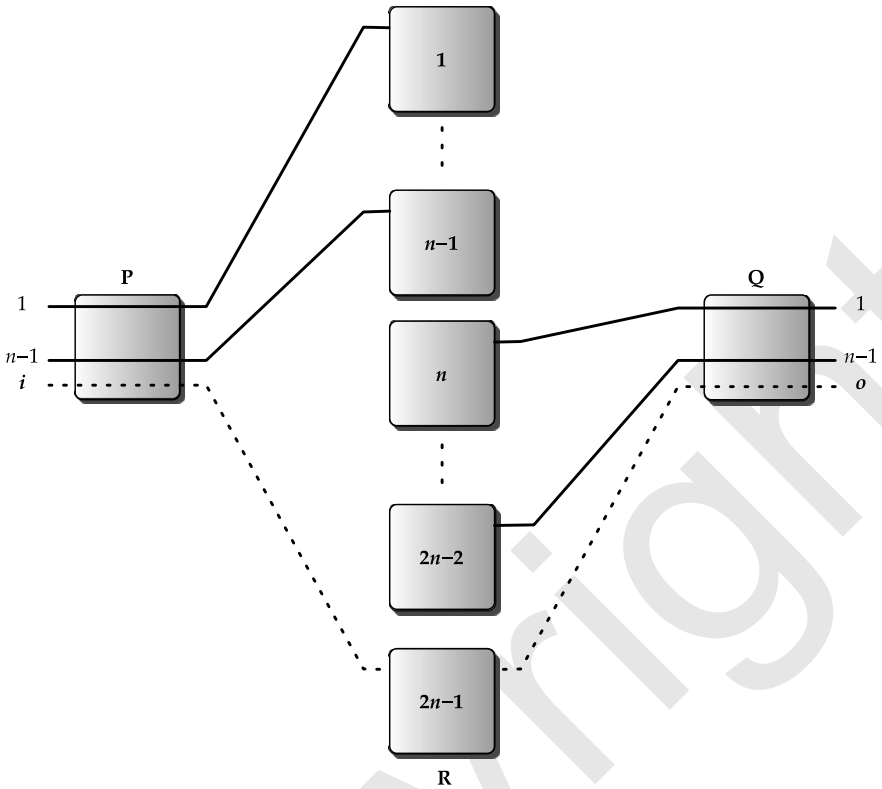


**FIGURE 21.25** Proof of the Clos theorem.

$n-1$ of its inputs can be busy. These inputs can be switched to at most $n-1$ switches in the middle stage. Similarly, if $Q$ is an $m \times n$ switch, at most $n-1$ outputs can be busy and these outputs are fed from at most $n-1$ middle-stage switches.

We can assume, without loss of generality, that the connections from switch $P$ use the first $n-1$ middle switches and the connections to switch $Q$ use the last $n-1$ middle switches. This scenario is depicted in Figure 21.25. To connect $i$ to $o$, a middle switch not used by both $P$ and $Q$ is required. Since $n-1$ middle switches are used by $P$ and different $n-1$ middle switches are used by $Q$, there must be at least $2n-1$ middle switches for a switch to be available for communication between $i$ and $o$.

Note that there is an implicit assumption that switches $P$ and $Q$ are crossbars or other nonblocking switches. This implies that switch $P$ always finds a path to connect $i$ to the corresponding input link of the middle switch $R$ and, similarly, switch $Q$ finds a path to connect to the corresponding output link of $R$ to $o$.

Clos' result is interesting since it showed, for the first time, that nonblocking switches can be constructed with less than quadratic complexity. If $m = 2n-1$ and $r = N/n$, then the total number of crosspoints of a Clos switch is

$$(2n-1) \times N + (2n-1) \times N/n^2 + (2n-1) \times N,$$

which is less than $N^2$. The number of crosspoints is minimized when $n = \sqrt{N/2}$ and it is approximately equal to $5.76N\sqrt{N}$. For instance, if $N = 1024$, then the total number of crosspoints for a crossbar is approximately a million. For a nonblocking Clos switch, the number of crosspoints is approximately 190,000, which represents a savings of 81%. However, this benefit of reduced crosspoints is achieved at the expense of increasing the latency (two additional stages of switching), but typically this is acceptable.

For instance, consider the design of a multichassis router where individual routers are connected by high-speed switching fabric.

While nonblocking Clos networks are clearly desirable, they come at a cost. Can we do better? By reducing the number of middle switches, the cost of the switch can be further reduced. However, in this case, the Clos network is no longer nonblocking. Instead, when $m = n$, the Clos switch becomes what is called *rearrangeably nonblocking*. A switch is rearrangeably nonblocking when a new connection from input $i$ to an unconnected output $o$ may require rearranging some existing connections to use different middle-stage switches. While a nonblocking network might be desirable, it is not necessary for a router. Let us examine why.

Looking at a bit of history, the Clos networks were originally used in telephone networks that are circuit switched. In these networks, a circuit is established before the actual conversation takes place. During the establishment of a circuit, a path is chosen through the Clos network and held for the entire duration of the conversation, which could be seconds or even minutes. However, routers use the Clos network to move packets between the line cards. The packets are often further divided into fixed-sized cells (typically between 40 and 64 bytes) and transferred from their input ports to output ports for every time slot, as long as a path can be established in the switch. Unlike circuit-switched networks, a path from input $i$ to output $o$ is held only during the duration of a time slot (which is extremely small—on the order of nanosec or picosec) and freed at the end of the time slot. Again, at the beginning of the next time slot, a new set of paths between input and output ports is established, cells are transferred, and so on. This approximately achieves the same effect of rearranging circuits in a circuit-switched network.

### 21.12.1 Complexity of Scheduling Algorithms

With a rearrangeably nonblocking Clos switch, the scheduling becomes more complex. To schedule cells from input ports to output ports, it is necessary to address two issues. First, an input-buffered Clos switch, like crossbar, will suffer from HOL blocking. To avoid HOL blocking, each input port maintains a separate queue for each output port in such a way that cells in a VOQ do not block cells in any other VOQ except when contending for an input port. Hence, a scheduling algorithm is required to determine a set of nonconflicting cells from $N^2$ input queues to be transferred to $N$ output ports. This is similar to the bipartite graph matching discussed in Section 21.9.1, where input ports form one set of nodes and the output ports form another set of nodes. Some algorithms that find fast maximal matches are discussed in the context of scheduling crossbar switches in Sections 21.9.2 and 21.9.3.

Second, given a maximal match of input and output ports, we need to identify internally conflict-free paths connecting an input port to an output port through the middle-stage switches. Such a route assignment can be mapped onto an edge-coloring problem in a bipartite multigraph. Note that this bipartite multigraph is different from the bipartite graph

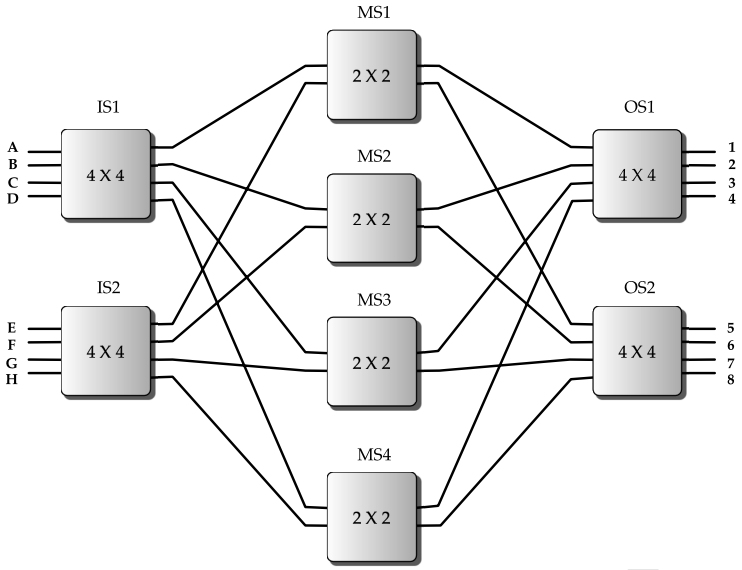


**FIGURE 21.26** A (4, 4, 2) rearrangeable Clos network.

used for matching input and output ports. In this bipartite multigraph, the input switches are mapped to one set of nodes $I$ and the output switches to the other set of nodes $O$. The edges in the bipartite multigraph represent the routes needed through the middle switches between a node in $I$ and a node in $O$. There can be more than one edge connecting a node in $I$ to a node in $O$ and hence it is a multigraph.

If a unique color is assigned to represent each middle switch, the assignment of the routes is equivalent to coloring the edges of the bipartite multigraph such that no two edges coming out of a node have the same color. Well, what does this mean, intuitively? Coloring of the edges represents the use of a distinct middle switch to connect one input switch to an output switch. Recall that in a Clos network each middle switch has a single link to one input switch and output switch. If two paths are needed for the same pair of input and output switches, then it requires the use of two different middle switches. This is reflected by the constraint that no edges incident on the same node should be of the same color. The following example will provide a better understanding of this.

**Example 21.11** *Scheduling in a three-stage Clos switch.*

Consider the three-stage Clos switch illustrated in Figure 21.26 with four middle-stage switches. For the sake of discussion, assume a matching algorithm based on cells waiting in VOQs identifying the pairs $(A, 3)$, $(B, 6)$, $(C, 1)$, $(D, 7)$, $(E, 2)$, $(F, 4)$, $(G, 5)$, and $(H, 8)$ of input and output ports as the maximal match. This maximal match is shown as a bipartitie graph in Figure 21.27(a).

With these pairings, we can identify the number of paths needed from an input switch $i$ to an output switch $j$. For instance, consider pairs $(A, 3)$ and $(C, 1)$. These require two paths connecting switch IS1 and switch OS1. For all the pairs, the paths required are represented as a switch bipartite multigraph shown in Figure 21.27(b). Now if each of the four middle

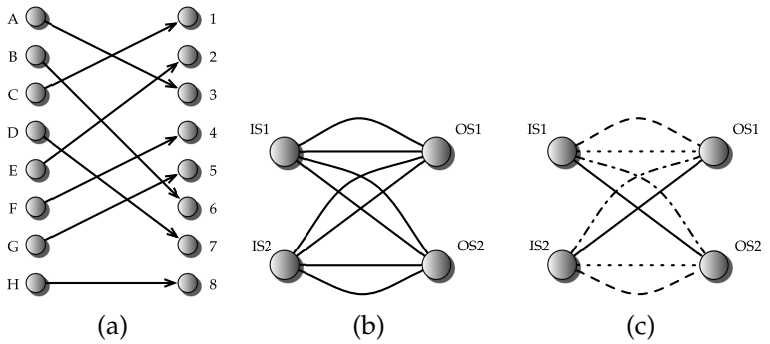


**FIGURE 21.27** Route assignment (a) and edge coloring for scheduling (b, c) in a Clos network.

switches is assigned a distinct color, the edges of the switch bipartite graph should be colored such that no two edges incident on the same node are assigned the same color. A possible color assignment is shown in Figure 21.27(c). In Figure 21.27(c), instead of colors, we use different line shades to illustrate edges with colors. ▲

Thus, the scheduling algorithm must not only compute the maximal match of the input and output ports but also the path through the middle stage. Hence, the time required for the scheduler to make a decision becomes longer as the fast known edge-coloring algorithm takes $O(N \log D)$, where $D$ is number of distinct colors. Therefore, with the increase in switch size (number of ports) and port speed, the time available for the scheduler decreases even further. As a result, some implementations instead of using slow edge-coloring schemes resort to approximate algorithms that distribute the traffic from each input across the middle switches using some form of load balancing. Recently, a class of algorithms that solves matching and the computation of paths simultaneously has been outlined [122], [123], [125].

An example of a commercial router using the Clos network is the T-series from Juniper Networks. It uses a Juniper-designed $16 \times 16$ crossbar chip as the building block for all stages of the Clos network [631]. The Clos network also provides the interconnection between line cards in a multichassis T-series router where up to 16 single-chassis routers are connected by a separate switch chassis.

There is another network type, called the Beneš (pronounced as "Be'nesh") network, that is very similar to the Clos network; Recall that that each switch at input in Clos networks is an $n \times m$ switch, where $n$ is the outer ports while $m$ is the inner ports, and in the inner core is an $r \times r$ switch. If we now set $n = m = 2$, and $r = n/2 = 1$, we have four switches where each one has two input and two output ports, i.e., a $2 \times 2$ fabric serves as the basis—this building block is recursively used in constructing the Beneš network. An example of a commercial router that has implemented the Beneš network is the CRS-1 routers from Cisco systems [140].

## 21.13 Torus Networks

So far, many of the switch architectures we examined use some form of centralized control for scheduling cells in every time slot. With the increase in line rates of the links and the need

for large number of ports, the switching backplanes in routers need to scale to bandwidths more than 1 Tbps. Such a large bandwidth requires the centralized scheduling algorithm to operate at high speeds and still find pairs of inputs and outputs that maximize the number of cells transferred. Since the currently used matching algorithms already trade accuracy for time, a further increase in speed could reduce accuracy and in turn affect the throughput of the backplane. A torus network provides an alternative that does not employ any centralized control. It belongs to a class of networks called *direct networks* where each node serves as an input port, output port, and a switching node of the network.

A $k_1 \times k_2 \times \cdots \times k_n$ torus network contains $N = k_1 \times k_2 \times \cdots \times k_n$ nodes placed in an $n$-dimensional grid with $k_i$ nodes in each row of dimension $i$. Each node is assigned an $n$-digit address $(a_1, a_2, \ldots, a_n)$ where a digit at position $i$ corresponds to dimension $i$. The digit at position $i$ uses the radix of the corresponding dimension, $k_i$. The nodes are connected by a pair of channels, one in each direction, to all nodes whose addresses differ by $\pm 1 \pmod{k_j}$ in exactly one address digit $j$. Hence, each node requires two channels per dimension for a total of $2nN$ channels. In a torus, there are many distinct paths between every pair of nodes. For instance, in an $8 \times 8 \times 8$ torus network, the packet can choose between 90 different 6-hop paths from the source node to the destination node. By dividing the traffic over these paths, the load can be balanced across the channels, even for irregular traffic patterns. This enables the torus to be fault tolerant. By quickly reconfiguring around faulty channels, the traffic can be routed along alternative paths.

A $4\times$ 4 torus network is shown in Figure 21.28. Each node of the network uses a two-digit address and both the digits use a radix of 4. Observe that the neighbors of the node whose address is 11 are the nodes with addresses 01, 12, 21, and 10, all of which differ by a single digit $\pm 1$. Figure 21.28 also shows two distinct paths by which packets from node 11 can be sent to node 23, which are indicated by thick black lines. One path uses the intermediate nodes 12 and 13 while the other uses the intermediate nodes 21 and 22.

In torus networks, routing packets from one node to another node need to load balance the traffic across multiple paths for any traffic pattern. Otherwise, using a single path could lead to overloaded processing of the intermediate nodes, which might result in delay and eventually dropping of packets. To achieve load balancing, the following randomized algorithm described in [708] might be used. A packet from source node $s$ to destination node $d$ is first sent from $s$ to a random intermediate node $r$ and then from $r$ to $d$. For instance, in Figure 21.29, we show how a packet is to be delivered from node $s = 11$ to node $d = 23$ in a $4 \times 4$ torus network. It is routed via a randomly selected node 32.

How do we find the route from node 11 to node 32 and then from node 32 to node 23? Recall that adjacent nodes in a torus network differ by one digit in their address. This property can be exploited to direct routing [676]. At every intermediate node, the destination address is used to determine the dimension and the direction the packet should take for the next hop. This is referred to as *dimension-order routing* [163]. Alternatively, the source node itself computes the route and prepends the packet with the route information, which is known as *source routing*.

The torus network has been adopted in the Avici terabit switching router (TSR) [161]. It uses a three-dimensional torus topology $k_x \times k_y \times k_z$ that can be scaled to a maximum configuration of $14 \times 8 \times 5$ (560 nodes). Each line card carries one node of the torus and is assigned a three-coordinate address $(x, y, z)$. Each node is connected using bidirectional

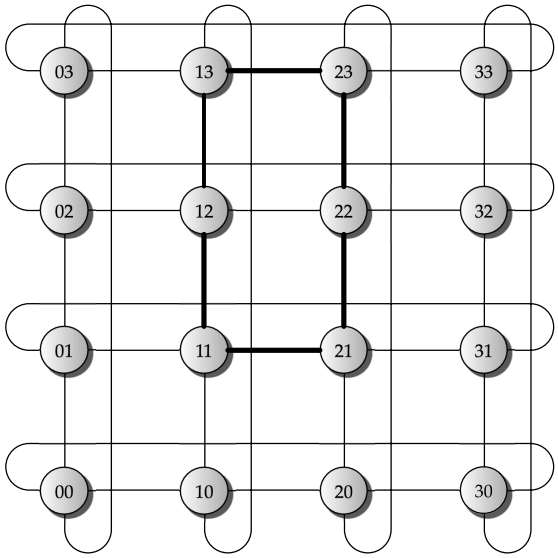


**FIGURE 21.28** A 4 × 4 torus network.

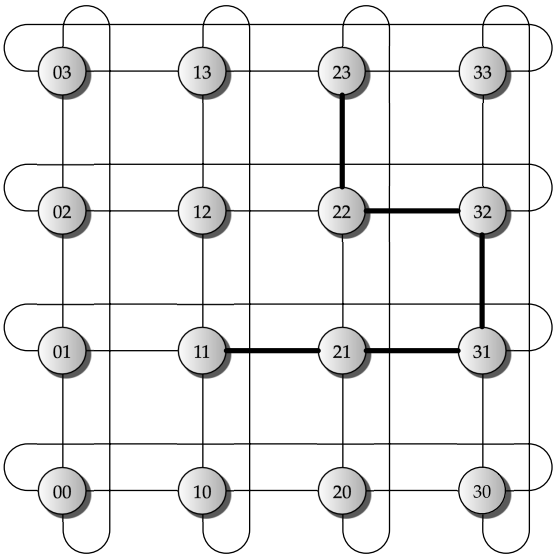


**FIGURE 21.29** Routing in a 4 × 4 torus network.

channels to six neighbors with the addresses $[(x \pm 1) \bmod k_x, (y \pm 1) \bmod k_y, (z \pm 1) \bmod k_z]$. Each of these channels provides a bandwidth of 10 Gbps in both directions.

The Avici TSR, unlike other router architectures that have dedicated switching fabrics, distributes the switching task to each node in the network. Hence, each line card should handle its own incoming and outgoing traffic in addition to the traffic from other line cards

that pass by it. Therefore, all the active components related to fabric are carried on the line card. This allows the TSR to expand incrementally starting with a few line cards; as traffic grows, more line cards can be added, as needed. Also, line cards can be added or removed while the router is in operation without affecting the forwarding of packets in other line cards.

Such flexibility can lead to partial torus networks because the router is not completely populated with all the line cards. Even a fully populated router can lead to irregular torus topologies because some line cards might have failed. To facilitate routing of packets in the fabric, even in such scenarios, the TSR uses source routing. The exact route for a packet through the fabric is determined by the sending line card or the source. This routing information is expressed in the form of a string such as $+x, +y, -z, -y, -x$. Each route entry specifies a single hop of the route from the source line card to the destination line card. For instance, $+x$ means that for the corresponding hop the packet should be forwarded in the positive $x$ direction.

The routes between any source line card $s$ and any destination line card $d$ are computed depending on the current and possibly irregular topology in a software process that populates a table in hardware. Whenever a source line card needs to send a packet to a destination line card, it consults the table and appends the route information to the packet header. This route is used at every hop to determine the next line card to which the packet needs to be forwarded.

As noted earlier, the torus fabric provides many different paths for packets from one line card to another line card. To accommodate various IP traffic patterns, without overloading any of its internal fabric channels, the TSR balances the load by distributing packets across different routes. Each packet bound from a source line card $s$ randomly chooses one of the routes to the destination line card $d$. Since packets belonging to the same flow should be kept in order, a flow identifier is used in the random selection of a route. This ensures that all the packets in the same flow use the same route.

### 21.13.1 Packaging Using Short Wires

Another advantage of torus networks is the ability to package it using short wires for connecting nodes [161]. Why is this considered significant? Intuitively, the longer the wire is, the longer it takes for the signal to propagate, which causes increased delay. Furthermore, the bandwidth (bit rate) is inversely proportional to the square of the wire length [162]. For instance, when the wire length is doubled, the bandwidth decreases by a factor of 4. However, Clos and Beneš networks need to use longer wires for connections between stages. Hence, they have to operate at low bit rates or use more expensive signaling to compensate for the distance limitation of electrical signaling. Now let us see how a torus network uses short wires.

Figure 21.30(a) shows a one-dimensional torus network containing four nodes. As can be seen, the one-dimensional torus network is nothing but a ring that requires shorter wires to connect the nodes, except for the connection between the first node 0 and the last node 3, which requires a longer wire. A natural question is why a a shorter wire cannot be used to connect node 0 and node 3. Such an approach, while simple, restricts the placement of nodes in a higher-dimensional torus network. Hence, instead of reducing the wire length, the nodes can be placed at equal distances from each other over the entire length of the wire as shown

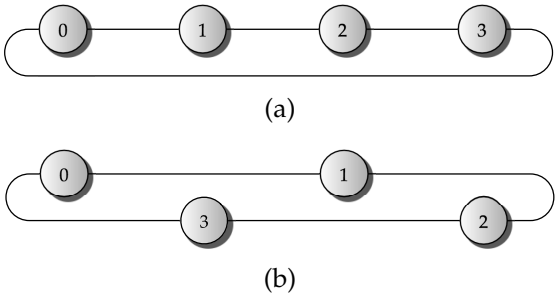


**FIGURE 21.30** Using short wires instead of long wires in a one-dimensional torus network.

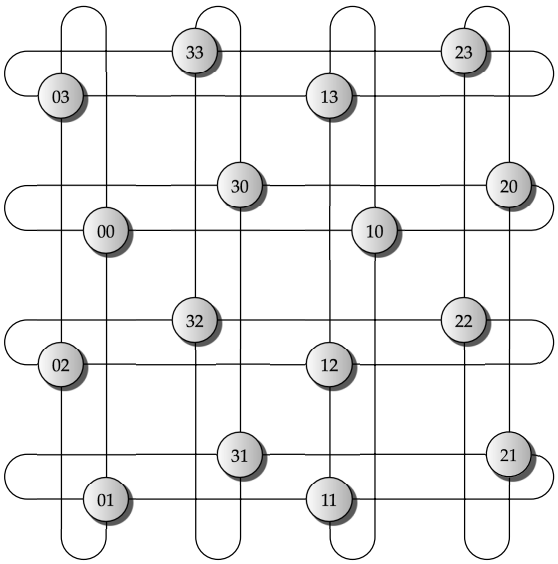


**FIGURE 21.31** A 4 × 4 torus network with short wires.

in Figure 21.30(b). In this approach, the length between nodes 1 and 2 might have doubled, but all the connections between the nodes have the same length and there is no longer wire. This approach can be easily extended for use in a higher-dimensional torus network. A 4 × 4 torus network can be easily packaged with uniformly short wires as shown in Figure 21.31.

## 21.14 Scaling Switches for High-Speed Links

So far, we have studied fabrics that scale in the number of ports. With advances in optical technologies physical network links connecting to the router can be as fast as 10 Gbps. It is anticipated that the link speeds can become as fast as 40 Gbps. Such an increase in the link rate places more of a burden on the switches to transfer more data per second between the line cards. In this section, we outline various techniques on how switches can be scaled to accommodate higher link speeds.

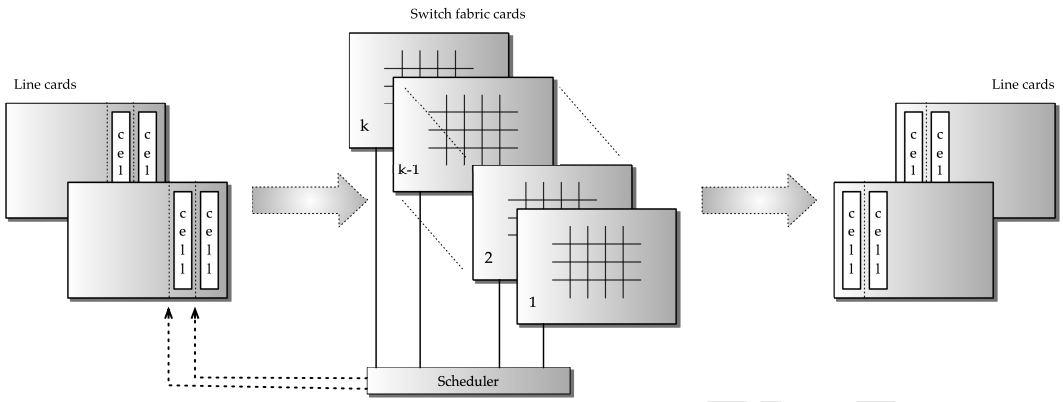


**FIGURE 21.32** Bit slicing using $k$ identical switching planes and central scheduler.

## 21.14.1 Bit Slicing

Rather than a single, monolithic fabric, bit slicing utilizes $k$ parallel, individual fabrics referred to as *fabric planes*. The bit-slicing technique is shown in Figure 21.32. Here each cell of width $C$ bits is placed across $k$ identical planes and each plane carries a slice of size $C/k$ bits. A centralized scheduler ensures that all the switches are set to the same configuration during each time slot. These slices carried by each plane need to be reassembled at the output port to restore the original cell. This implies that the reassembly logic needs to operate at the same speed as the fabrics.

**Example 21.12** *Transferring cells using bit slicing*.

For the sake of discussion, assume a switch fabric with eight fabric planes containing three input ports, referred to as $A$, $B$, and $C$, and three output ports referred to as 1, 2, and 3. With a cell size of 8 bits, each bit can be transferred by a fabric plane. Also, assume that each fabric plane is a crossbar and the scheduler uses one of the algorithms, such as PIM or iSLIP. Based on the cells waiting in the VOQs, let us assume for the current time slot that the scheduler decides to pair port $A$ with port 2, port $B$ with port 3 and port $C$ with port 1. Subsequently, it turns the crosspoints $(A, 2)$, $(B, 3)$, and $(C, 1)$ in each of the fabric plane. The cells at ports $A$, $B$, and $C$ to be transferred are sliced into individual bits and each bit is transferred to a separate plane. Upon their arrival at the output ports, these bits are assembled to restore the original cell. Note that each bit of the three cells is transferred simultaneously in each fabric plane. ▲

## 21.14.2 Time Slicing

A different approach is to transfer an entire cell in a single fabric plane within a time slot. The line card distributes the incoming cells evenly across all the fabric planes. At the begining of time slot $i$, the scheduler makes the decision for fabric plane $i$ to transfer the cells. Thus, the scheduler works in turn on each of the $k$ fabric planes in a round-robin fashion and it takes $k$ time slots to transfer $k$ cells. Observe that in a given time slot only one of the fabric

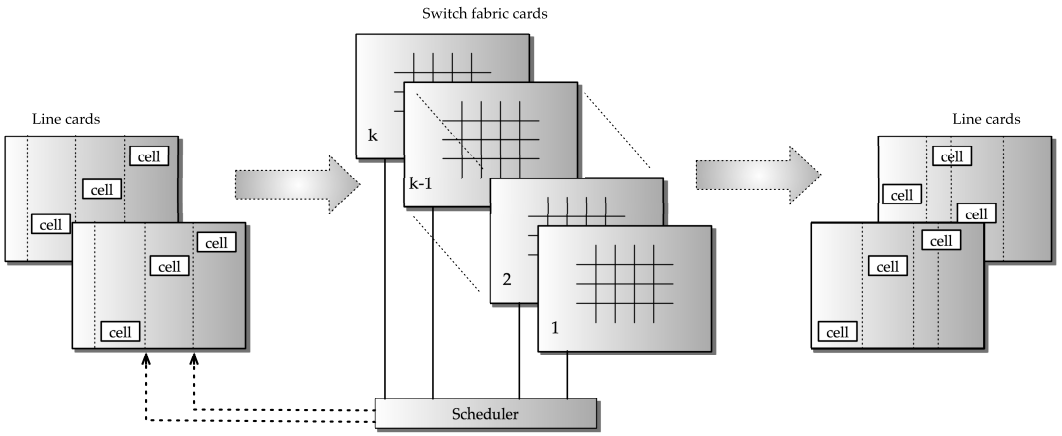


**FIGURE 21.33** Time slicing with $k$ identical switching planes and a central scheduler.

planes is actively transferring a cell. This approach is known as *time slicing* and is shown in Figure 21.33.

**Example 21.13** *Transferring cells using time slicing*.

Assume that the switch fabric is similar to the one described in Example 21.12 except that the number of fabric planes is three. To simplify the discussion, assume that each input port has three cells. Port $A$ has cells to ports 2, 3, and 1 in that order. Similarly, port $B$ needs to transfer cells to ports 3, 1, and 2 and port $C$ to ports 1, 2, and 3. In the first time slot, the scheduler chooses fabric plane 1 and connects port $A$ to port 2, port $B$ to port 3, and port $C$ to port 1 and turns those crosspoints to enable the transfers. Similarly, in the next time slot, fabric plane 2 is arranged in such a way that port $A$ is connected to port 3, port $B$ to port 1, and port $C$ to port 3. Finally, in the third time slot, fabric plane 3 carries the rest of the cells to their output ports. ▲

While bit slicing and time slicing provide simple ways to scale the switch to faster link rates, both have the disadvantage of a centralized scheduler. The design of the scheduler becomes challenging when the link rate increases as it has to operate at high speeds. Furthermore, the failure of the scheduler renders all the fabric planes nonoperable, which implies the failure of the router. Clearly, this is not desirable.

### 21.14.3 Distributed Scheduling

In commercial routers, a variation of the time-sliced approach is adapted. In this approach, shown in Figure 21.34, each fabric plane has its own scheduler and, hence, operates independently. Therefore, many cell transfers occur simultaneously across the fabric planes. This approach is advantageous since the scheduler design becomes simpler as it needs to operate at lower speeds when compared to approaches using a centralized scheduler. The failure of a scheduler affects only one fabric plane, and the other fabric planes can still continue forwarding cells.

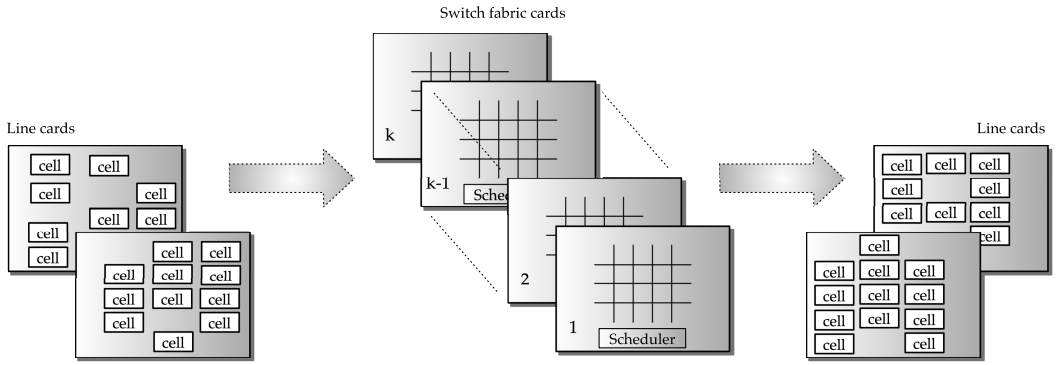


**FIGURE 21.34** Distributed scheduling with $k$ identical switching planes and independent schedulers.

Commercial routers like Juniper T-series [631] use a switch fabric with five identical fabric planes, but only four of them are active simultaneously and the fifth acts as a backup for redundancy. Each fabric plane carries a portion of the required bandwidth and when one of the active fabric plane fails, the redundant fabric plane takes over. If more than one active plane fails, the router will still continue to operate at a reduced bandwidth. Similarly, Cisco CRS-1 [140] also uses eight fabric planes and each fabric card implements two planes of the switch fabric. The traffic is evenly distributed across all the planes so that every plane carries an equal amount of traffic. The loss of a single plane does not affect router operation, and failure of multiple planes results in a linear and graceful degradation of performance.

## 21.15 Conclusions

A shared bus is a simple way to interconnect line cards. Because of their simplicity and low cost, many low-end enterprise routers from various vendors use a shared bus. These routers typically provide a throughput ranging between 1 and 2 Gbps. However, the shared bus limits throughput and, hence, it is not used in medium-sized routers that need to provide a throughput of 40 Gbps. Shared memory switches provide an attractive alternative for medium-sized routers, but the memory bandwidth limits its throughput. Instead, crossbar switches are used, which allow transfer of traffic between multiple line cards simultaneously. The limiting factors in a crossbar are the HOL blocking and the scheduling speed.

HOL blocking can be eliminated using VOQs but requires $N^2$ VOQs for $N$ ports. When the number of ports increases, the scheduling speed needs to increase so that it is fast enough to pair input ports to output ports. Hence, the design of such schedulers becomes complex. Furthermore, the number of crosspoints grows with $N^2$, which increases the complexity of implementation. Hence, the crossbar is the switch topology of choice for routers with a low to modest number of ports (up to about 64).

To scale switches to ports greater than 64, multistage switches are more appropriate as they have reduced crosspoint complexity. The three-stage Clos network with a crosspoint complexity of $O(N\sqrt{N})$ provides multiple paths from the input to output port. Typically, in routers, Clos networks are used in a rearrangeably nonblocking configuration and the sched-

uler ensures that the path exists in the switch when inputs and outputs are paired. Clos networks can scale as many as 256 to ports.

Unlike the aforementioned switches, torus networks do not employ a centralized scheduler. This eliminates the bottleneck of scheduling and the switch can scale to a large number of ports. Each node using a small routing table routes the packets from the source node in the torus network to a destination node by choosing a random intermediate node. These switches can scale as many as 512 ports.

## 21.16 Summary

In this chapter, we studied different type of backplanes that facilitate the movement of packets from one line card to another line card in a router. At a very high level such switches can be broadly categorized as shared backplanes or switched backplanes. At any given instant, a shared backplane transfers packets between any two line cards and hence the throughput is limited. However, a switched backplane transfers multiple packets simultaneously. In switched backplane, we started our discussion with single-stage fabrics—shared memory and crossbar. We examined in detail the scheduling algorithms for crossbar such as take-a-ticket, PIM and iSLIP and analyzed their pros and cons.

We then studied the need to scale switches along two different dimensions: number of ports and link speed. This is followed by a detailed discussion about various types of multistage switching fabrics, CLOS and Beneš, and the complexity of scheduling algorithms in these switches. We then examined the architecture of a torus network that belongs to a class of direct networks. Finally, we explained various techniques about how to scale switches for higher link speeds.

## Further Lookup

Excellent treatises on switching can be found in [124], [163], [191], and [548]. In the context of routing, a separate chapter is devoted to switches in Varghese's book [712]. Keshav [365] provides a nice introductory discussion about circuit and packet switches. Excellent surveys about switching are also available [4], [41], [172], [333], [534], [699]. A more recent survey of architectural choices for switches can be found in [705].

Prototypes of shared memory switch designs have also been described in [154], [175], [199], and [346]. A scalable memory switch using inexpensive DRAMs that emulates an output queueing switch has been presented in [332]. A recent study [342] outlines techniques for scaling the memory bandwidth of network buffers.

Karol et al. [352] showed that HOL blocking results in reduced throughput. Various techniques for reducing HOL blocking have been described [299], [351]. The idea of output queueing to eliminate HOL blocking was outlined in the knockout switch implementation [755]. VOQs were proposed in [682]. A PIM scheduling algorithm for a crossbar was discussed in [15], iSLIP in [458], and wavefront arbiter in [132] and [681]. In addition, a variety of scheduling algorithms have been proposed [9], [394], [433], [482], [483]. For excellent coverage on gigabit switching, see [547].

Clos published his seminal paper [146] on nonblocking networks that introduced the idea of Clos networks. Furthermore, it derived the conditions under which these networks are

strictly nonblocking. It was then discovered [70], [192], [647] that much smaller Clos networks were rearrangeably nonblocking. There has only been a few attempts to find good matching schemes for three-stage Clos networks. A random dispatching scheme that evenly distributes cell traffic to the second stage is given in [117] and [134]. Newer fast matching algorithms for Clos networks have been outlined [122], [123], [125]. Fast edge coloring algorithms are described in [147], [148].

Beneš networks [70] were first introduced in Beneš' classic book [71]. Torus networks have been used in some of the earliest parallel computers [56], [648]. A randomized routing algorithm in switches was first described in [708]. A high-level overview of switches used in commercial routers can be found in [121], [140], [161], and [631]. A comprehensive discussion of telephone switching can be found in [697].

## Exercises

21.1. Enumerate different types of backplanes and explain the advantages and disadvantages of each of them.

21.2. What are the disadvantages of a crossbar?

21.3. What is HOL blocking? How can you prevent it?

21.4. Explain a shared memory switch and why it is difficult to scale such switches to higher capacity.

21.5. What are the differences between the Clos network and the Beneš network?

21.6. What is the main difference between torus network and crossbar?

21.7. You are given the task of designing a router with 8 line cards. Each line card is capable of operating at 1 Gbps. If you were to use a shared bus using an internal clock rate of 100 MHz, what should be the width of the bus? If the electrical loading on the bus is 0.6, what should be the width of the bus?

21.8. A router needs to be designed using a shared memory switch with 8 line cards. Each line card is capable of 10 Gbps. The minimum size of the packet is 64 bytes. Assuming an interleaved memory design is used, how many memory banks will be required if the memory access time is 40 nanosec?

21.9. Consider a $3 \times 3$ crossbar shown in Figure 21.35. Each of input ports have cells 1, 3 and 2 destined for the respective output ports. How many time slots will be required to transfer all the cells using take-a-ticket scheduler scheme?

21.10. Consider a $3 \times 3$ crossbar shown in Figure 21.35. Can you provide an example stream of cells for each input where head of line blocking yields the lowest throughput?

21.11. For the crossbar shown in Figure 21.35, assume that the input port has the following cells. Port A—1, 2, 3, 1 Port B—1, 3, 2, 1 Port C—2, 3, 3, 2.

How many time slots will be required to transfer the cells to output ports using one iteration PIM and two iteration PIM? What is the switch throughput in both the cases?

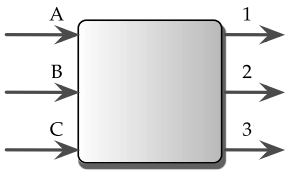


**FIGURE 21.35** A 3 × 3 crossbar.

21.12. For the crossbar shown in Figure 21.35, assume that the input port has the following cells. Port A—1, 3, 2, 1 Port B—2, 1, 2, 2 Port C—2, 3, 3, 2.

How many time slots will be required to transfer the cells to the output ports using one iteration iSLIP and two iteration iSLIP? What is the switch throughput in both the cases?

21.13. For the crossbar shown in Figure 21.35, assume that the input port has the following cells. Port A—1, 1, 1, 1 Port B—2, 2, 2, 2 Port C—3, 3, 3, 3.

How many time slots will be required to transfer the cells to the output ports using take-a-ticket scheduler? one iteration PIM and one iteration iSLIP? What do you observe?

21.14. Design an 8 × 8 three-stage Clos switch. Under what conditions will this be a nonblocking switch?

21.15. For a nonblocking 8 × 8 Clos switch, how many crosspoints will be required? If the switch is rearrangably nonblocking, how many crosspoints will be required? Do you see any saving in the number of crosspoints? If there are any, how much do you save?

21.16. Design an $n = 2$, $m = 3$, and $r = 4$ Clos network.