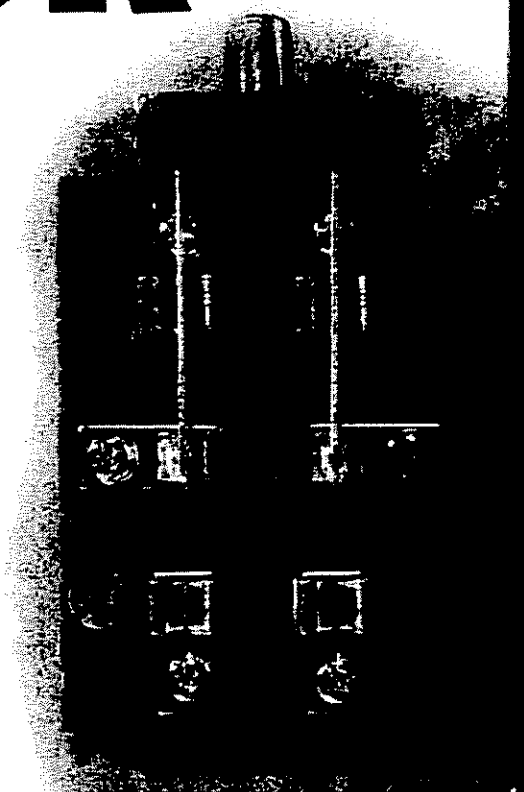
EXHIBIT D

WILEY

The Switch Book

The Complete Guide to LAN Switching Technology



Rich Seifert

Publisher: Robert Ipsen
Editor: Carol A. Long
Managing Editor: Micheline Frederick
Text Design & Composition: North Market Street Graphics

Designations used by companies to distinguish their products are often claimed as trademarks. In all instances where John Wiley & Sons, Inc., is aware of a claim, the product names appear in initial capital or ALL CAPITAL LETTERS. Readers, however, should contact the appropriate companies for more complete information regarding trademarks and registration.

This book is printed on acid-free paper. ♾

Copyright © 2000 by Rich Seifert. All rights reserved.

Published by John Wiley & Sons, Inc.

Published simultaneously in Canada.

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 750-4744. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 605 Third Avenue, New York, NY 10158-0012, (212) 850-6011, fax (212) 850-6008, E-Mail: PERMREQ@WILEY.COM.

This publication is designed to provide accurate and authoritative information in regard to the subject matter covered. It is sold with the understanding that the publisher is not engaged in professional services. If professional advice or other expert assistance is required, the services of a competent professional person should be sought.

Library of Congress Cataloging-in-Publication Data:

Seifert, Rich, 1952–
The switch book : the complete guide to LAN switching technology / Rich Seifert.
p. cm.
ISBN 0-471-34586-5 (cloth : alk. paper)
1. Local area networks (Computer networks) 2. Internetworking (Telecommunication) 3. Telecommunication—Switching systems. 4. Computer network architectures. I. Title.

TK5105.7.S455 2000
621.39'81—dc21 00-020735

Printed in the United States of America.

10 9 8 7 6 5 4 3 2

As silicon density grows, emerging designs are more likely to use embedded memory; that is, the filtering database can be located within the Lookup Engine chip itself. Extremely high-speed memory exchange rates can be achieved since the width of the memory can be virtually unlimited. Rather than having to use a conventional 32- or 64-bit wide data path, internal memory widths can be 256, 512, or more bits, effectively multiplying the available memory bandwidth.[16]

15.3 Switch Fabrics

The Receive data path performs all of the necessary qualification, classification, and table lookup functions needed to determine to which output port(s) a given frame should be forwarded. The Transmit data path takes these forwarded frames, applies additional qualifiers as needed, implements any prioritization and distribution policies, and sends the frames onto the appropriate output(s). A *switch fabric* sits between the Receive and Transmit data paths; its function is to transfer frames among all of the input and output ports of the switch.

The design of the internal switch fabric is critical to the performance of a switch. While a comprehensive treatment of this subject could fill its own book, in this section we look at three switch fabric architectures that have been widely used in commercial LAN switch products:

- Shared memory
- Shared bus
- Crosspoint matrix

Each approach reflects a distinct method used to move frames from input to output ports, and has its own set of characteristics, limitations, and design issues.

15.3.1 Shared Memory

A shared memory architecture entails the lowest level of both design and implementation complexity, and usually provides the lowest-cost solution in those products where it can be used. As a result, it is the most popular approach to fabric design in LAN switches. The primary limitation of a shared

[16] External memories are rarely practical in such wide organizations, since the number of pins required on both the memory and Lookup Engine devices would increase their cost prohibitively. By embedding the memory, we eliminate the pin restriction. A 512-bit wide, 8 ns SRAM can provide 64 Gb/s of raw memory bandwidth; this is the equivalent of a 64-bit wide external memory with a 1000 MHz clock.

memory switch fabric is the memory bandwidth; the speed at which data can be moved into and out of the shared memory will place an upper bound on both the number of ports in the switch and the data rates that can be supported on those ports.

15.3.1.1 Shared Memory Fabric Operation

A shared memory fabric uses a single common memory as the exchange mechanism for frames between ports, as depicted in Figure 15.9. Frames arriving through an input port's Receive data path are stored in the shared memory, where they can be assigned and channeled into the Transmit data path of the appropriate output port(s). Ideally, frames might be deposited directly from the receive port interface into the shared memory; in practice there is usually some amount of per-port memory required in the Receive data path where frames are stored temporarily during reception and classification. However, the bulk of the system memory is in the common buffer pool.

Initially, all of the frame buffers in the shared memory are unused; they begin their life in the pool of free buffers. Free buffers are assigned to frames as they arrive from the various input ports. Depending on the results obtained from the Lookup Engine, frames in the shared memory are linked into the appropriate output queues for the ports onto which they will be forwarded. These output queues represent the stream of frames to be processed in the Transmit data path (see Section 15.4). Frame discard is effected by returning a buffer to the free pool.

There are a number of advantages to a shared memory approach:

- Using a single large memory rather than individual buffer memories for each port lowers the total memory (and memory control logic) cost.
- The task of moving frames from one port to another becomes trivial. As shown in Figure 15.9, each port's output (transmit) queue comprises a linked list of buffer pointers representing frames stored in the shared memory. To move a frame from an input port to an output port simply entails linking a pointer to the appropriate frame buffer into the target output port's queue.
- The Lookup Engine can be implemented either as a single, common engine for all ports or in a distributed manner, with each port performing lookups for its own received frames. This is a function of the performance demands being placed on the Lookup Engine (see Section 15.2.6). Other fabric architectures (e.g., crosspoint matrices, see Section 15.3.3) make the use of a common Lookup Engine problematic.
- Lookup on a given frame may be completed before the frame is stored in the shared memory, or alternatively, frames may be stored while the lookup operation is performed in parallel. That is, a frame may be stored

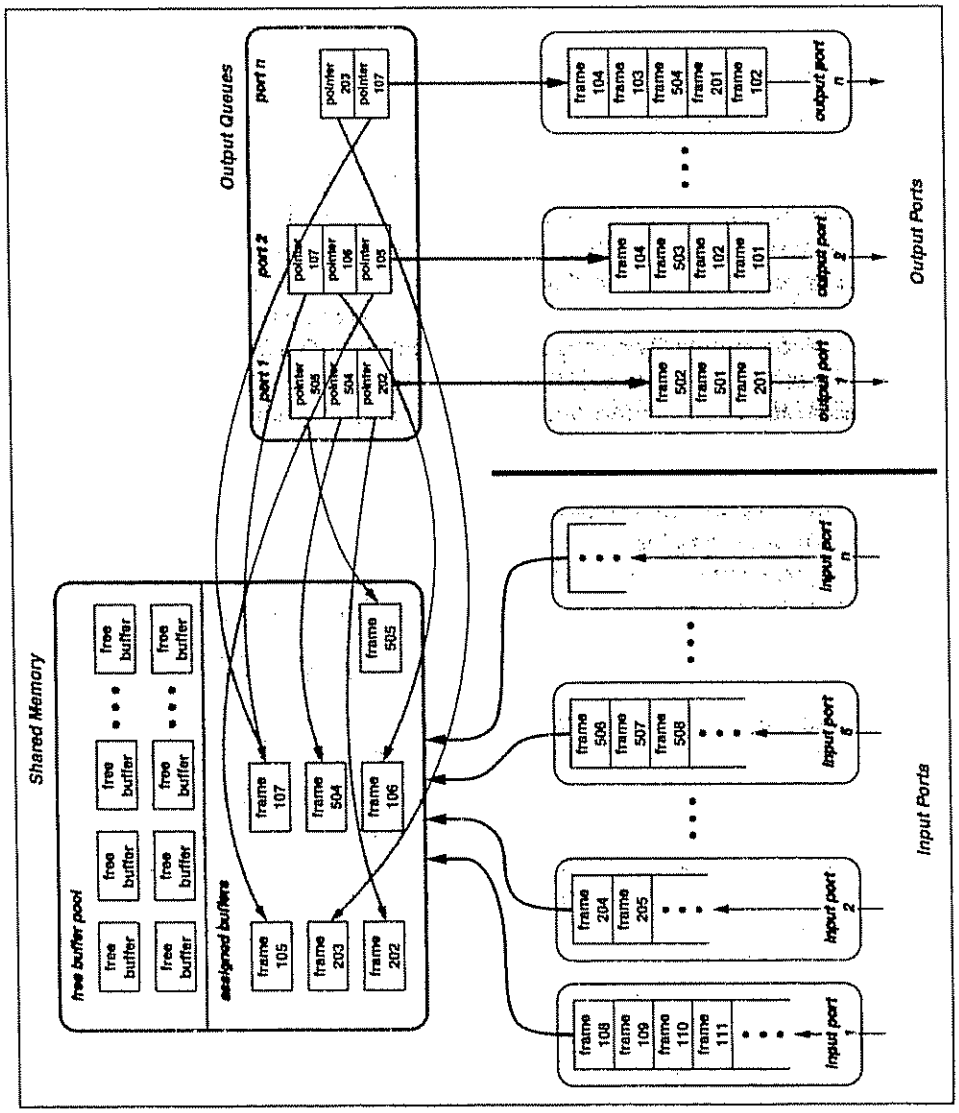


Figure 15.9 Shared memory architecture.

before the switch knows what to do with it; its buffer pointer can be linked to the appropriate output queue(s) once the Lookup Engine completes its work.

15.3.1.2 Multicasting in a Shared Memory Architecture

A shared memory fabric facilitates the ability to move frames from an input port to multiple output ports. Since the common memory is accessible from all ports, there is no need to perform multiple transfers or to provide special data paths for multicast traffic. Each frame buffer can be provided with a Transmit Port Map, as depicted in Figure 15.10.

The Lookup Engine determines the port(s) onto which the frame should be forwarded for transmission. If a frame is destined for more than one output port (e.g., it is a multicast or unknown unicast frame), the appropriate bit for each target output port is set to 1 in the Transmit Port Map. When the output port Transmit data path no longer needs a copy of the frame in the shared memory (either because it has completed transmission or because it has copied the frame into a local port buffer), it clears the bit for its port in the Transmit Port Map and checks whether there are any bits still set. Once all of the bits in the Transmit Port Map are clear, the frame buffer can be returned to the free pool.

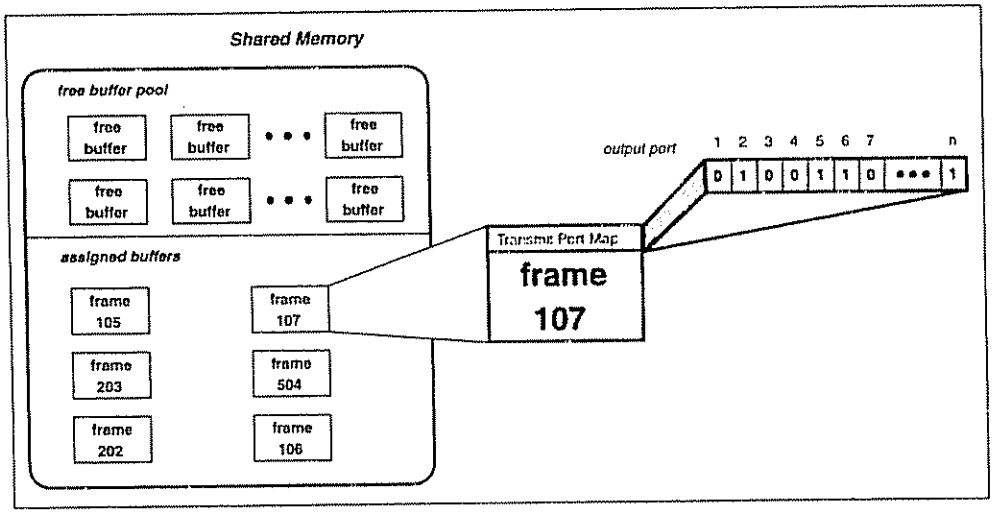


Figure 15.10 Transmit Port Map.

15.3.1.3 Buffer Organization

The shared memory is organized as a sequence of frame buffers (including any memory in the free buffer pool). However, there is no strict requirement that a given frame must be stored in one contiguous block. Frame buffers may be either contiguous or discontiguous. Each approach has its own set of advantages and disadvantages, as discussed in the following text.

15.3.1.3.1 Contiguous Buffers

The design of the memory interface and control logic is much simpler when buffers are contiguous. A contiguous buffer can be identified by a single pointer representing the location of the start of the frame. A Direct Memory Access (DMA) engine used to move frames into and out of the shared memory can work from this single buffer pointer; there is no need to scatter or gather frame fragments into disjoint portions of the shared memory.

The greatest simplicity occurs when the frame buffers are both contiguous and of fixed length. Using this scheme, the entire shared memory can be preallocated into distinct, numbered frame buffers. There is never any need to provide complex dynamic memory allocation when taking this approach. However, each fixed-length buffer must be large enough to store a maximum-length frame; to the extent that the received frames are smaller than the maximum possible length, the efficiency of memory utilization will drop. Thus, there is a clear tradeoff between the cost of the memory and the cost of the memory interface logic.

One common approach used in Ethernet switches is to allocate fixed-length contiguous buffers of 2,048 (2K) bytes. A 2-Kbyte buffer provides more-than-adequate storage for a maximum-length Ethernet frame (1,522 bytes, including VLAN tag), plus the internal switch header, Transmit Port Map, and any implementation-specific memory control information. In fact, the 526 bytes available for this internal overhead is much more than is generally necessary; however, if a round number (in binary) is used for the buffer length, the memory interface logic can use the frame number as the most-significant bits of each memory buffer pointer, as depicted in Figure 15.11. This represents a further simplification of the hardware logic at the expense of lower memory utilization.

While it is possible to design a shared memory switch fabric that uses variable-length contiguous buffers, this will generally result in a significant increase both in memory interface complexity and memory fragmentation. As a result, if the system cannot tolerate the memory efficiency afforded by contiguous, fixed-length buffers, designers generally opt for a discontiguous buffer allocation scheme, since this provides both higher memory utilization and variable-length buffer support without incurring significant memory fragmentation problems.

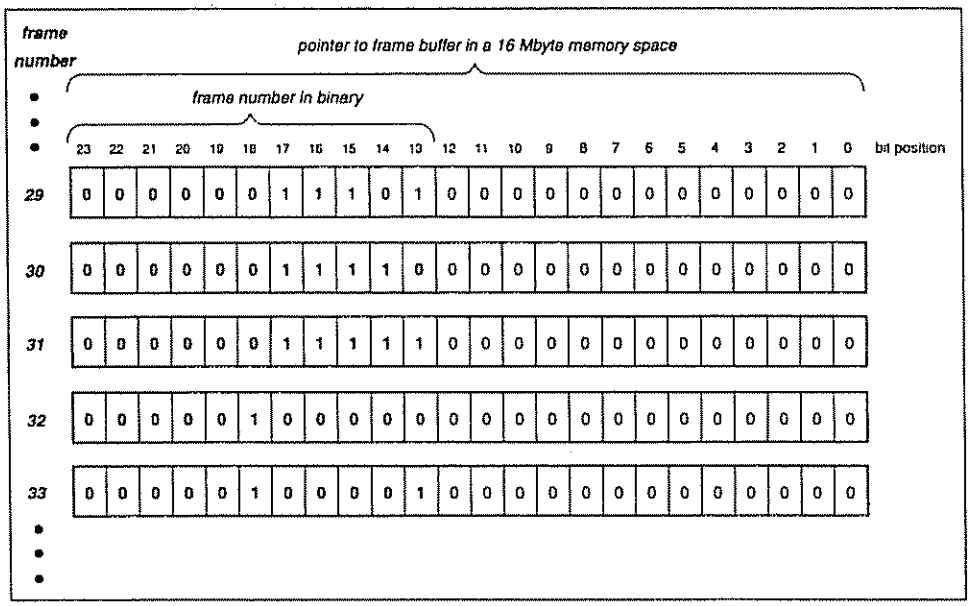


Figure 15.11 Using frame numbers as buffer pointers.

15.3.1.3.2 Discontiguous Buffers

Rather than storing a frame in a single block of memory, the use of discontiguous buffers allows a single frame to be stored as a series of frame segments scattered throughout memory. Thus, the start of a frame is identified by a single buffer pointer, but that pointer only indicates the first of the frame's possible segments. The entire frame is referenced by traversing a linked-list of buffer pointers, as depicted in Figure 15.12.

The linked list of memory pointers may be stored either in the output queue data structure (as shown in Figure 15.12) or within the frame data buffer segments themselves by having the end of each data buffer segment provide either the next memory pointer or a flag indicating the end of the linked chain.

Each frame segment can be variable in length; usually there is some minimum value supported by the hardware, with the total length of each segment being some multiple of this minimum. A discontiguous variable-length buffer approach uses memory much more efficiently than any fixed-length buffer scheme. In the worst case, the amount of memory allocated that goes unused is one minimum-length buffer less 1 byte, rather than a maximum frame.[17]

A linked-list buffer scheme can handle much larger frames than a fixed-

[17] It is common practice to use a 64-byte minimum segment length when using a discontiguous buffer approach in an Ethernet switch, since this is the length of the minimum Ethernet frame. The larger the minimum length of a segment, the fewer links will be required for a given frame, at the expense of somewhat reduced memory efficiency.

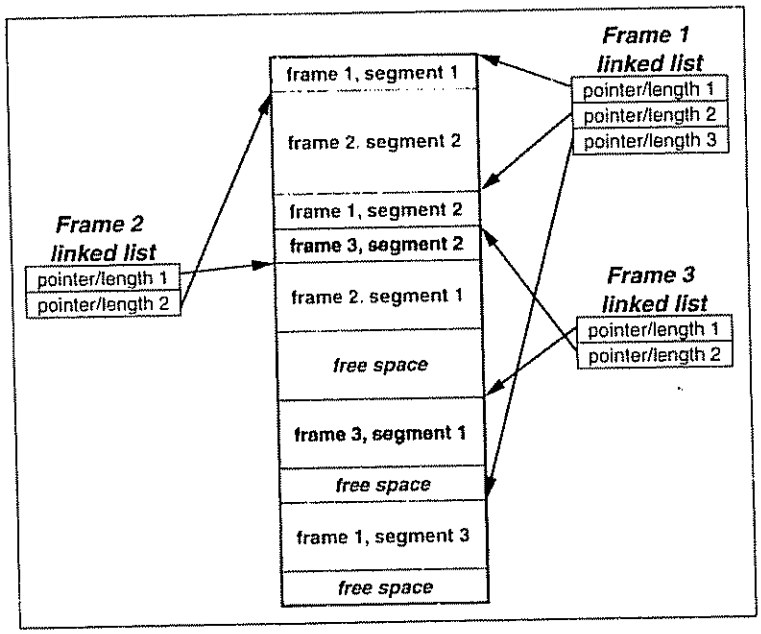


Figure 15.12 Discontiguous buffer pointers.

length approach without sacrificing memory efficiency. Token Ring and FDDI LANs can support longer frames than can Ethernets; while the memory inefficiency of using fixed-length buffers is often tolerated for 1,522-byte Ethernet frames, an increase in memory interface complexity may be justified for the 18-Kbyte frames used in some Token Ring systems.

WHAT'S THE BIG DEAL ABOUT MEMORY ORGANIZATION?

In most computer contexts, we gladly accept greater complexity in the memory data structures in exchange for better efficiency. Why are we so concerned over the organization of the buffers in switch memory?

A computer program can easily manipulate complex memory data structures; other than requiring additional care and thought in the coding of the program, there is no cost impact associated with the implementation of discontiguous, variable-length buffers as linked-list queues in software. For performance reasons, high-speed switches cannot implement the shared memory data structures in software. All of the buffer manipulation and queue management must be performed by dedicated hardware—typically a set of finite-state machines—within the switch. Any complexity imposed on the organization of the memory buffers results in more complex and expensive logic for the memory interface. Thus, there is a real cost tradeoff between memory efficiency and data structure complexity that must be addressed by the switch fabric designer.

15.3.1.4 Memory Bandwidth Limitations

Quite simply, if a system *can* use a shared memory switch fabric, then it probably *should* use a shared memory fabric. A shared memory architecture is usually the simplest and lowest-cost solution available to the switch designer. Given this conclusion, why would any switch designer not choose a shared memory approach?

Since all frames must pass through the single, common shared memory, the speed of the memory interface will ultimately limit the capacity of the switch. For example, consider a memory system with a 32-bit wide data path, using 10 ns synchronous static RAM (SSRAM) and a 100 MHz clock. Each frame must pass across the memory interface at least twice—once when written from the input port and once when read into the output port.[18] This effectively halves the available memory bandwidth. Even in the best case of streaming data (i.e., 10 ns memory access times can only be achieved for multiple sequential reads or writes of data, not for individual random accesses) the aggregate switch capacity is limited to:

$$32 \text{ bits} \times 100 \text{ MHz} \div 2 = 1.6 \text{ Gb/s}$$

In practice, we cannot get the full benefit of data streaming through synchronous RAM; the actual performance improvement due to streaming will be a function of the design of the messaging system used to move frame data into and out of the memory. Typically, the usable memory bandwidth will be on the order of 50 to 70 percent of the maximum. Furthermore, to prevent unbounded queuing delays, the effective memory bandwidth should be undersubscribed by a factor of at least 30 percent. This results in a usable nonblocking switch capacity on the order of 35 to 50 percent of the raw memory bandwidth, or about 550 to 800 Mb/s in the example just given.

Assuming full to duplex operation on all ports, this capacity is adequate for a 10 Mb/s switch with a port count of 24 to 48 or more, or even a 48-port 10 Mb/s switch with two 100 Mb/s uplinks.[19] The switch cannot provide nonblocking operation for more than eight 100 Mb/s ports. Thus, the design example given is appropriate only for desktop or small workgroup switch applications.

15.3.1.5 Increasing the Memory Bandwidth

A number of approaches can be taken to increase the memory bandwidth so that a shared memory fabric can be used for higher-performance switches:

[18] Multicast traffic may have to be read into multiple output ports, as discussed in Section 15.3.1.2

[19] If the ports are operating in half duplex mode, the design can either support more ports or provide additional margin for non-blocking operation.

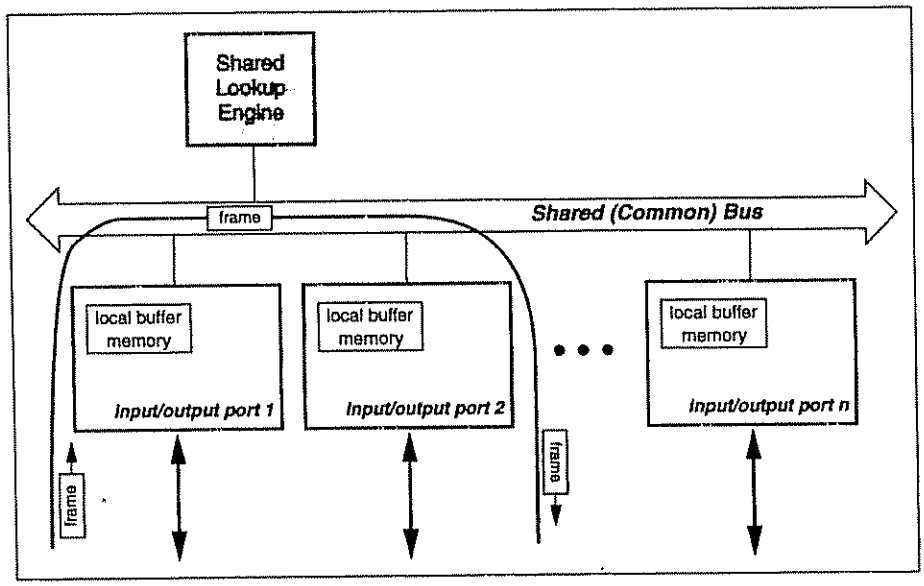


Figure 15.13 Shared-bus architecture.

15.3.2 Shared Bus

In the old days, memory was much slower than it is today, and most embedded processor systems were designed around 16-bit-wide internal data paths. In that environment, the available switching capacity of a shared memory fabric would be quite limited, typically on the order of about 150 Mb/s.[20] Thus, shared memories were practical only in 10 Mb/s switching systems with low port densities (15 or fewer ports for non-blocking operation). If a switch comprised a greater number of ports, or ports operating at higher data rates, it was usually necessary to change the fabric design.

A shared bus architecture (depicted in Figure 15.13) uses a common bus as the exchange mechanism for frames between ports, rather than a common memory. Each port (or small group of ports) has its own memory, both for input and output queues, depending on the design. The main advantages of this approach are:

- A shared bus can often provide higher bandwidth than an older-style shared memory. For example, a 32 bit, 33 MHz PCI has a theoretical maximum capacity of slightly over 1 Gb/s—more than 7 times the 150

[20] This is for a system using 25 ns SRAM and a 16-bit data path, after allowance for overhead, undersubscription, and two memory operations (write/read) per frame.

Mb/s available from the preceding example. Proprietary buses can be used to achieve even higher capacities through the use of wider bus widths or faster clocks. A 1-Gb/s shared bus architecture can support a moderate number of 100 Mb/s ports (typically eight) and/or a very large number of 10 Mb/s ports.

- A shared bus is accessed only once per transaction (rather than once in, once out as in the shared memory case), so the effective capacity is doubled relative to the shared memory approach.
- Shared bus fabrics provide a natural mechanism for multicast transfers; it takes the same amount of time and bus capacity to transfer a frame to a single output port or to multiple output ports.
- A shared bus architecture can support either a distributed Lookup Engine (i.e., one engine on each interface) or a shared, common engine, depending on the performance demands of the system. A shared engine can be connected either to the same bus as is used for data transfers (as depicted in Figure 15.13) or to a separate, dedicated control bus. In the former case, some bus capacity is used to transfer header information to the Lookup Engine (and lookup results back to the requesting interface); this capacity cannot also be used for data transfers. The latter case imposes the greater cost and complexity of providing two bus structures.

The disadvantages of a shared bus architecture include:

- *A separate memory is needed for each port (or group of ports).* As there is no common, shared memory among ports, each interface module must provide local memory for both input and output-queued frame storage. The memory on each interface must be sized for the anticipated worst-case load through the port. Depending on traffic patterns, some of the ports may become memory-starved while ample free buffers are available on other port interfaces. When all of the frames shared a common buffer pool, memory could be used more efficiently; that is, the statistical nature of load distribution allows a shared memory to support more traffic for a given memory capacity. Thus, for a given level of performance, a shared bus architecture will require more memory than a shared memory system, which increases cost.
- *Some means is required to arbitrate for the shared bus.* A shared bus appears very much like an internal, high-speed, shared-media LAN. As such, each port must contend for use of the bus. Once access is granted, the port uses the bus to transfer a frame from its local memory (input queue) to the local memory (output queue) of the target destination port.
- *Multiple input queues may be required.* Depending on the worst-case bus latency, there may be a need for a prioritized arbitration mechanism

and/or multiple input queues at different priority levels. This effect is much more prevalent in a crosspoint matrix fabric, and is discussed in detail in Section 15.3.3.5.

The product literature for many early modular internetworking devices prominently displayed and touted their backplane capacity. This was usually a clear sign that the system used a shared bus architecture; the capacity of the shared backplane bus guided the potential user as to the number and data rates of the interfaces that could be configured in the system while sustaining non-blocking operation.[21]

A shared bus architecture does not scale well; it is generally not possible to increase either the data rate or the bus width without redesigning all of the switch port interfaces (in particular, any ASIC devices designed to connect to, and arbitrate for, the shared bus). Most modern switches use either a shared memory or a crosspoint matrix fabric; rarely is there much call for a shared bus anymore. As memories have gotten cheaper, faster, more sophisticated, and/or embedded, the small (if any) improvement possible from a shared bus is usually not enough of a change to provide sufficient benefit.

15.3.3 Crosspoint Matrix

As depicted in Figure 15.14, a crosspoint matrix switch fabric creates a transient connection between an input and an output port for the duration of a frame (or subset of a frame) exchange. There is effectively a matrix of electronic switches available to create a connection between every possible input and output port pair. The fabric control logic connects a given input to the appropriate output(s) on a frame-by-frame basis, in a matter of nanoseconds, and then disconnects the ports in preparation for the next frame exchange. A crosspoint matrix fabric is often called a *crossbar*; the terms are synonymous.

Because every input port can be connected to some output port at the same time, the total data transfer capability of the crosspoint matrix can be much larger than that of a shared bus, and actually grows with the number of ports.

15.3.3.1 Multicasting in a Crosspoint Matrix Fabric

Unlike a shared memory or shared bus architecture, a crosspoint matrix provides no natural means to transfer frames from one input port simultaneously to multiple output ports. There are two common approaches to dealing with multicast traffic in a crosspoint matrix:

[21] For example, the first family of products from Wellfleet (later Bay Networks, then Nortel Network) used a standard VMEbus as the shared backplane. Operating with a 10 MHz clock and a 32-bit-wide bus, the system had an aggregate capacity of 320 Mb/s; this was considered quite a high-performance device at the time (late 1980s).

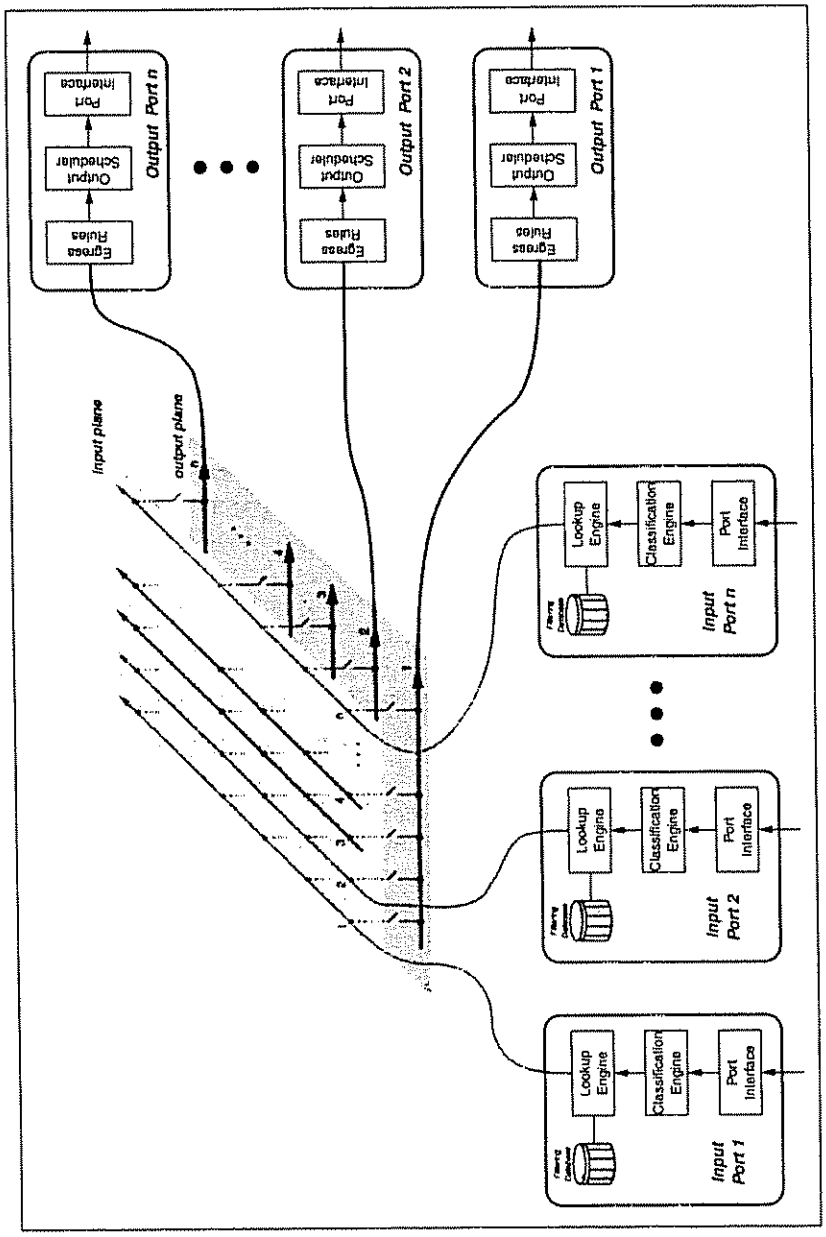


Figure 15.14 Crosspoint matrix fabric.

- The capacity of the shared multicast bus may be a limiting factor if multicast traffic comprises a significant fraction of the total load.[23]

15.3.3.2 Crosspoint Matrix Implementation

There are two popular approaches to implementation of the crosspoint matrix:

- A *physical matrix* can be created by laying down strips of conductive material in parallel on one plane (input ports) and strips of conductive material at right angles to the input ports on a second plane (output ports).[24] Each input port trace is connected to every output port trace by a transistor switching circuit; the switch control logic decides which of the transistor switches are closed or open based on the requests made by the input port controllers and the resolution of those requests by the fabric arbitrator (see Section 15.3.3.5). That is, the depiction in Figure 15.14 represents the physical configuration of the crosspoint matrix.

  Depending on the data rate supported by the fabric, it is often necessary to use multiple material traces for each port (both input and output). Multiple bits of data can then be transferred at each clock tick, which reduces the clock speed for a given rate of data transfer at the expense of increasing the pin count for the switch ports on the fabric chip.

- A *logical matrix* can be created by defining the signals for each output port as a logic function derived from the signals at the input ports, under control of the fabric arbitrator. That is, a crosspoint matrix can be built from standard digital logic blocks. As with the physical matrix, clock speeds can be reduced by defining each input and output in terms of multiple parallel signals at the expense of increasing the pin count at the fabric interface.

In general, a physical matrix can be smaller and less complex, resulting in a lower-cost switch fabric. The disadvantage is that the integrated circuitry must typically be custom-designed and laid out; it is not generally possible to build a physical matrix in silicon using the conventional IC design tools used for synthesized logic. The advantage of a logical matrix is that because it uses

[23] At least one switch fabric IC offers the capability to make a tradeoff between the amount of shared multicast capacity and the number of unicast ports that can be used. That is, the system designer can assign any of the fabric's output ports for multicast use; the shared multicast bus aggregates the selected number of multicast ports into a single logical channel. Those ports not assigned to multicast service can be used as unicast outputs.

[24] Normally, polysilicon is used for the conductive material in an integrated circuit.

synthesizable logic it becomes much more portable across semiconductor manufacturing processes and requires less specialized design skills.

Depending on the clock speed and the width of the port interface to the matrix, a huge switching capacity can be created. For example, a 16-port, 16-bit-wide (16 signals per port) matrix running on a 100 MHz clock has an effective data transfer capability of 25.6 Gb/s. Such a fabric would be suitable for huge numbers of 100 Mb/s ports and/or quite a few Gigabit Ethernet ports [26]

15.3.3.3 The Head-of-Line Blocking Problem

Crosspoint matrices can provide an extremely port-scalable, high-capacity switch fabric. However, they suffer from a systemic problem known as *head-of-line blocking*. Consider what happens when multiple input ports must contend for the availability of a single output port. Figure 15.16 depicts a situation where a lot of traffic is destined for output port 9; ports 1, 2, and 3 each have frames queued and waiting for port 9 to become available. Assuming that the switch arbitration algorithm is fair, those frames will be transferred to port 9 in an orderly manner. The switch capacity will be evenly distributed among all of the input ports that have traffic destined for the single target output port.

The problem is with port 4. Port 4 has frames queued and waiting to be transferred to a variety of output ports. However, the first frame in the queue is destined for output port 9, which we know to be experiencing short-term congestion. Since only one input port at a time is able to transfer frames to output port 9, each input port with a frame queued for port 9 must wait its turn. Assuming fair, round-robin scheduling, port 4 may have to wait for ports 1, 2, and 3 to transfer their respective frames to port 9 before it gets to unload the frame at the head of its queue. In the meantime, there are frames waiting in port 4's input queue behind the frame destined for port 9. These frames are targeted at uncongested ports; that is, the target output ports (ports 6, 1, and 3 in Figure 15.16) are idle and awaiting traffic. However, port 4 cannot proceed to transfer those frames because the frame at the head of the line is blocking progress.

Note that this situation is not a result of any flaw in port 9; there is a transient overload condition where the traffic targeted at port 9 temporarily exceeds that output link's capacity. The switch is (properly) resolving that short-term congestion by spreading the offered load over time. During that spreading time, frames must wait in queues within the switch. Ideally, we would like the frames to wait in the output queue for port 9, but we need to transfer frames across the switch

[26] An ASIC implementation of such a crosspoint matrix would require (16 × 16 × 2 =) 512 pins just for the input and output port connections. Additional pins would be needed for switch control, power, and so on.

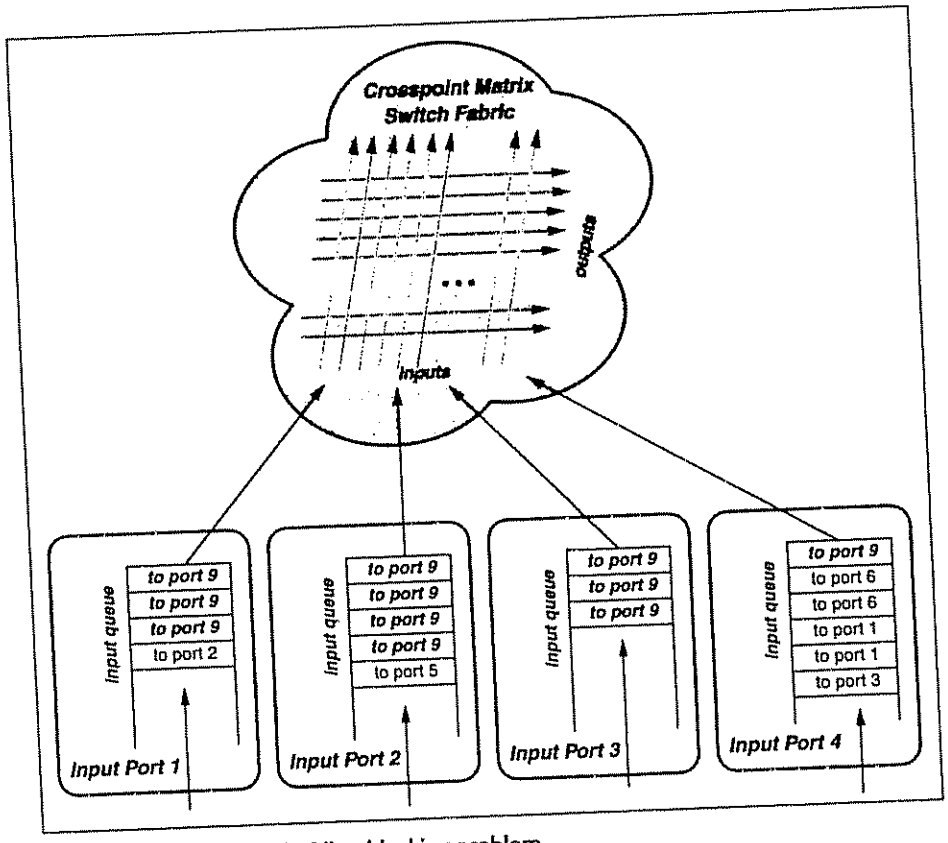


Figure 15.16 The head-of-line blocking problem.

fabric to achieve that state. To the extent that the switch fabric capacity is also experiencing transient congestion due to the many-to-one traffic pattern, some of the frames are still in the input queues waiting to cross the fabric. It is this condition that leads to the blocking problem.

Head-of-line blocking reduces the aggregate throughput of the switch as a function of traffic patterns. If some ports regularly experience short-term congestion (e.g., ports connected to servers that are often the target for heavy offered load), throughput can be reduced on other ports that are completely uncongested. This is an undesirable situation; there is no good reason to reduce the throughput between ports 4 and 6 just because ports 1, 2, and 3 are sending a lot of traffic to port 9.

15.3.3.4 Solving the Head-of-Line Blocking Problem

A number of solutions to the head-of-line blocking problem have been implemented in practical switches, including:

- Queue lookahead
- Priority queues
- Per-output-port input queues

Each one differs in complexity of implementation and effectiveness in eliminating the blocking effect.

15.3.3.4.1 Queue Lookahead

One of the simplest approaches is to implement a lookahead mechanism on the input queue. Each input port arbitrates for the target output port as determined by the frame at the head of its queue. When multiple input ports contend for the same output port (e.g., port 9 in Figure 15.16), one of the input ports will be granted access and the others will be temporarily blocked. When this occurs, the blocked input port can look ahead to the next frame in its input queue to see if the target output port needed by that second frame is available. If it is, the port can transfer that second frame rather than the one at the head of the line. Once transferred, the input port can once again arbitrate for the frame at the head of the queue and, if necessary, repeat the lookahead algorithm if the congested output port is still unavailable.

The hardware required for a simple one-stage queue lookahead is not especially complex and does not significantly increase the cost of the queue management logic. Of course, the mechanism is effective only if there is a single frame blocking the head of the line. If both the first and second frames in the queue are destined for congested output ports, then all of the other frames in the queue are still blocked (e.g., input port 1 in Figure 15.16). However, the probability that two frames are both head-of-line blockers is lower than the probability that only one frame is causing trouble.[26]

Queue lookahead is most effective when the fabric arbitration mechanism provides a means to make a *standing request*, that is, to request an output port and have the switch provide notification when it is available without further action on the part of the input port. That way, the frame at the head-of-the-line will always be assured of delivery, regardless of the time distribution of the traffic destined for the congested output port. The input port can continue to process lookahead frames until the fabric arbitrator tells it that that (congested) output port has become available, ending the head-of-line block.

15.3.3.4.2 Priority Queues

An alternative approach to solving the head-of-line blocking problem is to equip each input port with multiple queues representing different priority levels, as depicted in Figure 15.17.

[26] The probability can be reduced further by incorporating a two-level lookahead, although this increases the hardware complexity. At some point (typically one or two levels), lookahead schemes provide diminishing returns, and it becomes more practical to implement one of the multiple-input-queue mechanisms discussed in Sections 15.3.3.4.2 and 15.3.3.4.3.

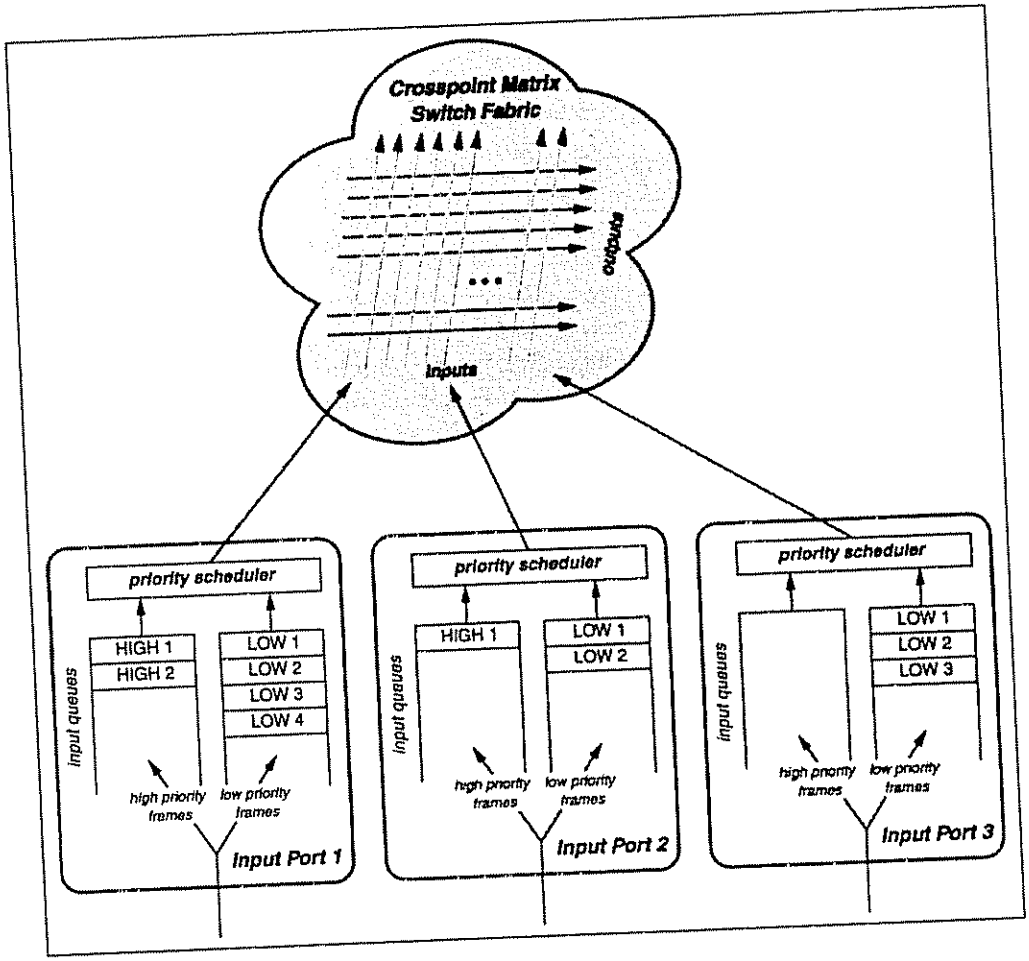


Figure 15.17 Priority input queues.

Frames are placed into one of the input queues based upon the priority as determined by the Classification Engine (and recorded in the internal switch header). The input port nominally schedules frames for fabric arbitration in priority order; that is, frames in the high-priority queue(s) receive preferential treatment relative to frames in the lower-priority queue(s).[27] Frames will be

[27] In theory, any of the mechanisms discussed could be used for output queue scheduling (e.g., weighted fair queuing; see Chapter 13). In practice, strict priority is generally adequate and implies simpler queue management logic. A strict priority policy will starve lower-priority queues only when the sustained, steady-state, high-priority offered load exceeds the capacity of the switch fabric or its output ports. In a LAN switch environment, this is presumed not to be the case; sustained overload indicates a configuration problem (i.e., inadequate capacity) that cannot be resolved by complex priority mechanisms.

processed from lower-priority queues when all higher-priority queues are either empty or currently experiencing head-of-line blocking.

Thus, head-of-line blocking may still occur, but at least a low-priority frame will never block one of higher priority. In addition, queue lookahead can be implemented in addition to priority queuing, further reducing the probability of blocking (especially for the higher-priority queues).

A few points should be noted regarding the implementation of priority input queues:

- Priority queues can be implemented on any input port without regard to whether this mechanism is being used on other input ports. That is, the fact that an input port has multiple queues and is selecting frames for fabric arbitration according to priority classification is invisible to the switch fabric or to other ports on the switch. Thus, to the extent that it is known in advance on which ports head-of-line blocking is anticipated, the additional cost can be incurred only when needed.
- Queue priority can be effectively combined with arbitration priority (see Section 15.3.3.6). If the switch fabric itself supports arbitration at multiple priority levels, the input queues can be mapped to the fabric arbitration priorities.
- The number of priority levels or queues provided does not have to be the same as either the number of priorities resolved by the Classification Engine or the number of Classes of Service provided at the output ports. The purpose of prioritization at the input queue is to avoid having high-priority traffic head-of-line-blocked by lower-priority traffic. Regardless of input queue priority, all frames will still end up in an output queue and be scheduled according to their (end-to-end) user priority. Just as user priority is mapped to the available Classes of Service in the output queues, user priority is mapped to the available number of input queues and/or switch arbitration priority levels.

15.3.3.4.3 Per-Output-Port Input Queues

One complete solution to the head-of-line blocking problem is to provide an input queue for each possible output port, as depicted in Figure 15.18.

At the point where the Receive data path is ready to transfer a frame across the fabric, it must of course know the target output port. Rather than placing all of the frames waiting to be transferred into a single input queue (or a set of prioritized queues), a port can maintain a separate input queue for each possible output port. A scheduler selects frames from the non-empty queues in a fair (e.g., round-robin) manner. If the target output port is available, the input port transfers the first frame in that queue across the fabric. If the target output port is busy, the input port moves on to the next non-empty queue. This mechanism completely eliminates head-of-line blocking, since the unavailabil-

- *Reduced the available bandwidth by using some of the data interface capacity for arbitration exchanges.* In practice, a switch designed for in-band arbitration signaling must include extra capacity to account for this effect.

15.3.3.6 Priority Levels in the Switch Fabric

Now, to make things *really* complicated, we can provide multiple priority levels for the switch arbitration. Under this scheme, an input port makes requests not only for a particular output port and request type but at a specified priority level. The arbitrator uses the request priority level as an input to the algorithm used to resolve contention for an output port. Higher-priority requests will be granted ahead of lower-priority requests for the same port. Note that the number of priority levels provided by an arbitrator is completely independent of any other priority mechanisms in the switch (e.g., classification priorities or output queue Classes of Service). The purpose of prioritization here is to schedule the switch fabric in the manner that moves the most important traffic to the output ports in the most expeditious manner.[32]

15.3.4 Input versus Output Queues

Regardless of the design of the fabric, any switch works best when the frames get to the output queue(s) as soon as possible. If there are frames in an output queue and the output port is available, those frames can be sent immediately. If the output port is available, but frames are still waiting on the input side of the switch fabric, the output port is unnecessarily idled; we can't send frames that are still sitting in an input queue. Thus, if frames have to wait, it is better to wait in an output queue than in an input queue; this maximizes channel utilization at the output.[33]

15.3.4.1 Input Queues and Shared Memory Switch Fabrics

Note that with a shared memory architecture there really isn't any input queue with respect to the switch fabric. Frames are stored in memory in one of two states:

[32] Typical fabrics provide two, four, or eight levels of priority.
[33] This is a practical application of the "hurry up and wait" syndrome. We want to speed frames through the Receive data path processing and across the switch fabric, even if they will have to sit in an output queue waiting for transmission.

- *The frame is awaiting processing by the Lookup and/or Classification Engine.* At this point in the process, we do not yet know the output port(s) to which we need to transfer the frame. The memory is being used for temporary storage within the Receive data path; there is no issue of head-of-line blocking on such frames, since they are not yet queued for transfer across the switch fabric.
- *The frame has been inserted into the appropriate output queue(s) and is awaiting transmission.* Once the Lookup Engine has completed its task, it knows the output port(s) to which it wants to transfer the frame. This transfer is effected by linking the appropriate buffer pointer into the output queue for the target port(s). As this can be an atomic operation performed by the Lookup Engine, we never have frames awaiting transfer across the fabric for which we know the target output port. While there may be buffers within the Receive data path to provide time for frame processing, a shared memory architecture needs no input queue to the fabric itself.

15.3.4.2 Input Queues, Output Queues, and Flow Control

Recall from Chapter 8 that the PAUSE flow control mechanism allows a device to request that its full duplex partner stop sending frames for a period of time. Typically, this action would be taken because the instant device (e.g., a switch port) sees that there will be imminent buffer overflow if frames continue to arrive.

Once frames are transferred to an output port within a switch, it becomes rather difficult to use the PAUSE mechanism to prevent buffer overflow. Frames on output queues have typically arrived from a multitude of different input ports. When the output queue begins to grow, how does the switch know on which input port to assert flow control? Without a crystal ball (or an amazing traffic-anticipation heuristic, which amounts to the same thing), it is impossible to predict the port on which future frames will arrive that will cause the troubled output queue to overflow.

PAUSE flow control is effective only with an architecture that maintains input queues to the switch fabric. If an input queue is in danger of overflow, we know a priori that any overrun will be caused by frames arriving from the input port feeding that queue. Thus, we can apply high-water marks on the input queue and assert PAUSE flow control as needed.

Even with an input-queued system, flow control becomes problematic if link aggregation is being used. Since neither the fabric arbitration mechanism nor the aggregator's frame collector will typically know the algorithm being used by the link partner to distribute frames across the multiple links, it is not generally possible to determine which physical link within the aggregation will

receive the frames that ultimately cause queue overflow. Unless we are willing to throttle an entire aggregation as a result of a single, traffic-intensive conversation, flow control can only be used to prevent overflow of the individual physical link queues within the frame collector, not the aggregated input queue to the switch fabric. IEEE 802.3x flow control was designed specifically to prevent buffer overflow under transient overload conditions in memory-constrained, input-queued switches. As a result of:

- Rapidly decreasing memory costs
- The emphasis on rapid transfer of frames from input ports to output queues in modern switch designs
- The widespread use of link aggregation

PAUSE flow control has becomes less useful and effective than originally intended. While it is often provided as a product feature in commercial switches, most users disable the function.

15.4 Switch Data Transmit Path Functions

Ultimately, the switch fabric delivers a stream of frames to each output port. These frames may have entered the switch from a single input or from a multitude of input ports. In any case, we need to perform some additional checks and processing before sending them onto the physical link. Figure 15.20 depicts the logical blocks present in the output port (transmit) data path. The sections that follow look at the operations performed within each of these blocks.

15.4.1 Output Filters

For each frame targeted at a given output port, we need to perform two VLAN-related qualification tests before actually queuing it for transmission. As discussed in Chapter 12, the Egress Filter determines whether the output port is in the member set for the VLAN to which the frame belongs. A frame should never be sent on a port that is not in the member set for its associated VLAN. Under most conditions, frames will never be forwarded through the switch fabric to output ports not in the VLAN member set. However, it is possible that, due to inconsistent static entries in the Filtering Database or the operation of Shared VLAN Learning, frames will appear at the output that should not be sent on that port. The Egress Filter detects and prevents this condition.

For each frame that should properly be transmitted on the output port, the Egress Rules function determines whether it must be sent tagged or untagged. This decision is made on a per-frame basis, as a function of the VLAN to which