

IMPLEMENTING TRAFFIC-MANAGEMENT MECHANISMS AND ENFORCING QoS GUARANTEES IN ROUTERS PRESENT SIGNIFICANT CHALLENGES IN SCALING TO NEXT-GENERATION ROUTERS AND BEYOND. THE BEST APPROACH INVOLVES A MIX OF TECHNIQUES, INCLUDING NEW ALGORITHMS TO SPEED SELECTED TASKS, INNOVATIVE HARDWARE, AND PARALLEL-PROCESSING TECHNIQUES.

Issues in implementing queuing and scheduling for high-performance routers

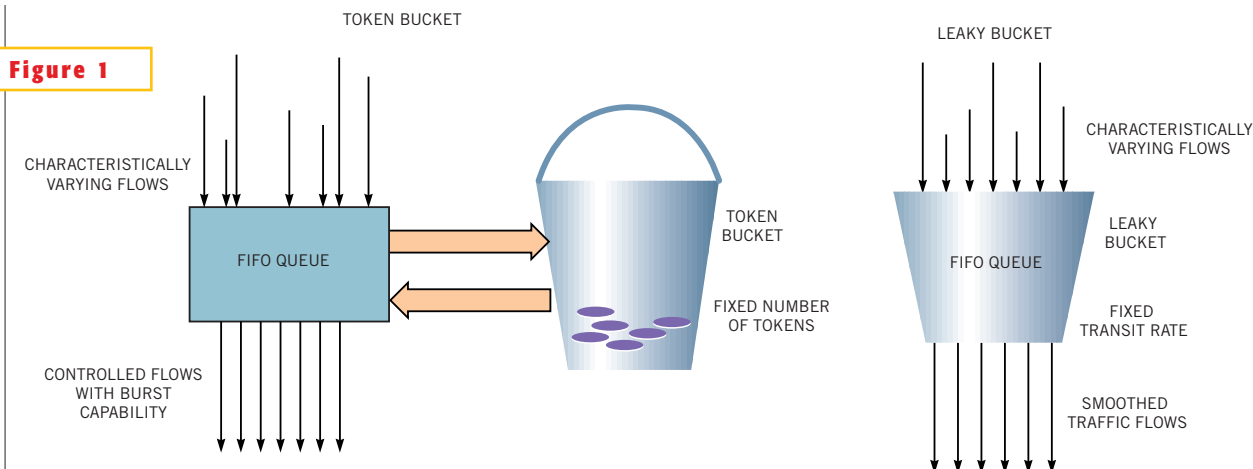
AS THE CORE MARKET for backbone routers moves from OC-48 speeds to OC-192 and beyond, system designers face a host of challenges, including the need for queuing and scheduling mechanisms to implement QoS (quality-of-service) capabilities that are a business requirement for the next generation of wide-area networks.

Suppliers need to maintain QoS via intelligent queuing and scheduling to avoid congestion due to the contention for network resources. A number of approaches exists to address the issues and mecha-

nisms that affect QoS. These approaches include traffic policing, buffer-space management, intelligent traffic discard, and backplane-or switching-fabric overspeed. Wire-speed classification and wire-speed buffering are requirements as well.

CLASSIFICATION

Classification at wire speed is necessary for QoS. Anything less than wire speed becomes a point of congestion that holds up downstream resources. The basic objective of classification is to associate a set



Token and leaky-bucket designs are two approaches to traffic policing.

of traffic-handling and engineering rules with every packet that enters the router. Because many approaches exist for performing this classification, designers use a specialized flexible engine. Addressing schemes such as CIDR (classless interdomain routing) requires a longest-prefix-matching look-up algorithm for finding associated data. Filtering identification in general may require various matching operations, such as greplike capabilities. CAMs (content-addressable memories) can quickly perform many of the simple classification operations; however, other operations require specialized sequencing state machines or network processors.

TRAFFIC POLICING

Traffic policing, a well-known technology in the ATM (asynchronous-transfer-mode) world, is another mechanism you can use to prevent congestion. Traffic policing, as the name implies, involves monitoring (metering) the router's incoming traffic flow and making sure it conforms to an expected profile. The proliferation of standards such as Diff-serv currently mandate the incorporation of traffic-policing functions in edge routers.

In some applications, it may be acceptable to police traffic by means of class granularity and coarse classification. But it's generally better to monitor and control traffic using microflow granularity, because it gives the system operator (usually a service provider) fine-grained control over traffic entering the network. One of the biggest advantages of microflow granularity is that it provides the ability to meter bandwidth use on a per customer basis. However, unless the system operator can communicate these negotiated characteristics to other service providers and enforce them in administrative domains throughout the network, such fine-level controls cannot be as effective as desired.

Depending on the line rate, policing may be executed in software or hardware. Software-based methods tend to be more flexible and programmable, but hardware-based implementations may be required for higher line rates.

Common approaches to traffic policing include token- or leaky-bucket regulators (**Figure 1**). The leaky-bucket scheme enforces a constant traffic flow,

MONITORING AND CONTROLLING TRAFFIC USING MICROFLOW GRANULARITY GIVES THE SYSTEM OPERATOR FINE-GRAINED CONTROL OVER TRAFFIC ENTERING THE NETWORK.

and the token-bucket architecture allows burst transmission as well. The system places tokens in a bucket at a constant rate. As packets arrive, they are policed depending on the number of tokens already in the bucket. If a given packet size is equal to or less than the number of tokens in the bucket, the system serves it and removes the tokens from the bucket. When the number of tokens in the bucket reaches the depth of the bucket, the system discards arriving tokens to avoid bucket overflow.

One crucial element of the token- and leaky-bucket architectures is their ability to efficiently update the status of buckets. A number of approaches exists for bucket designs. Some systems constantly scan buckets in the background (implemented as a counter in hardware or a timer interrupt for a software implementation) while continually dropping tokens into buckets with each counter increment. With other methods, when packets arrive, bucket-status computation is based on time stamps and external timers.

Flow-state memory maintains the state of the classified metered flows that are associated with traffic-management parameters. In terms of hardware, bucket schemes implemented with counters are less costly but consume a significant amount of flow-state-memory bandwidth due to background scanning. Using time stamps and timers is preferable if insufficient bandwidth exists in the flow-state memory, but the hardware cost is high. Both techniques are likely to satisfactorily scale to higher data rates.

BUFFERING BANDWIDTH

Wire-speed buffering, a requirement for line rates exceeding OC-48, poses a

significant challenge because the buffering rate must be much higher than the wire rate due to the switching fabric's overspeed requirements. The following OC-48 packet-over-SONET (synchronous-optical-network) example illustrates this point.

To estimate worst-case requirements, you should consider the smallest packet, which is generally 40 bytes. Next, you must make allowances for overhead, such as management information, protocol or other routing headers, and performance monitoring; overhead is usually a fixed size but is a higher overall percentage for smaller packets. For example, if you assume that the packet overhead represents 20% and that 2-times overspeed is required for the switching fabric, you have a worst-case requirement of approximately 8.5-Gbps raw bandwidth across the buffer-memory interface at OC-48 rates. At OC-192, this requirement grows to 34 Gbps.

A 64-bit SDRAM interface running at 143 MHz gives a raw bandwidth (uncompensated for RAS, CAS, or refresh overheads) of 9.152 Gbytes/sec, which is sufficient for OC-48 rates. A 133 MHz DDR (double-data-rate) SDRAM supports a bandwidth of more than 15 Gbits/sec, which is ample for OC-48 performance but insufficient for OC-192.

Variable-size packets coupled with inefficient allocation and deallocation algorithms often result in ranges of unused free addresses. It's much simpler to design memory controllers for ATM switches than for packet switches. ATM's fixed cell size allows memory to be logically partitioned into fixed-size segments made up of a cell plus its fixed overhead. With packet-based switches, it's desirable to configure memory into logical fixed-size cells to simplify memory management. Such segmentation simplifies management but impacts the spatial usage efficiency of the memory subsystem, especially when cell sizes vary considerably.

DRAMs have a clear advantage in cost and density, but their inefficiency using memory-interface bandwidth is a disadvantage, which is due to the overhead penalties that you incur for precharging and activating rows.

Today's QoS mechanisms require multiple queues and sophisticated scheduling algorithms to control dequeuing. Be-

cause packet dequeuing from memory may be arbitrary, you may need a precharge/activate cycle for every burst, which would significantly reduce DRAM-bandwidth efficiency. Increasing memory-data-bus widths to handle higher speed-line rates is limited by two factors: maximum packet size and precharge-activate duration.

Using SRAMs significantly reduces the precharge/activate overhead and the bandwidth inefficiency. However, the higher device cost and lower density increase the final system cost. Higher line rate requires the use of the largest and fastest parts. But, at such high clock rates, only a few devices can be reasonably driven electrically, limiting the total size of the memory that you can implement.

Despite their limitations, the low cost of DRAMs makes them the most compelling choice today and most likely for the future. Thus, memory-controller architectures based on DRAMs must be designed to maximize bandwidth usage across the interface.

Because DRAMs are usually configured in independent banks, the most common method of minimizing idle time is bank swapping. This method involves precharging/activating one bank while bursting data into the other. The inefficiency of bank swapping lies in the arbitrary order that packets are read from the buffer memory. The order in which packets are read depends on the scheduler, and it's difficult to guarantee that bank interleaving will always occur when the system performs a series of read operations. Another difficulty with bank swapping is ensuring that the time required for the data burst is greater than or equal to the precharge time. Techniques such as pingpong buffering and smart segmentation can work around these problems.

Pingpong buffering effectively doubles bandwidth in memory buffers (**Reference 1**). The system requires two devices for buffer memory, and both devices are accessed simultaneously; one device is read at precisely the same time that the other is written. The scheduler usually controls read sequencing, but no such constraint is placed on write operations directed to the companion device.

You can apply a bank-interleaved variation of pingpong buffering to DRAMs

with multiple banks using a single read/write interface to eliminate wastage due to double buffering of the pingpong technique. In this scenario, write and read operations are always interleaved. Write operations are directed to a different bank, which is not accessed by the preceding or succeeding read operations. This technique requires a DRAM configuration of four banks. Bank-interleaved pingpong buffering effectively eliminates any idle-time penalties for DRAM precharging and activation.

You must also control packet segmentation for memory schemes such as ping-

A SIGNIFICANT AMOUNT OF STUDY AND EFFORT HAS GONE INTO THE DEVELOPMENT OF ALGORITHMS DESIGNED TO DIVIDE BUFFER SPACE BETWEEN COMPETING FLOWS AND TO HANDLE BUFFER-OVERFLOW SITUATIONS.

pong buffering. As noted, it's highly desirable to segment the buffer memory into fixed-size cells to make memory management more efficient. Pingpong buffering also requires a data-burst time that is greater than or equal to the precharge time. Thus, because large packets are segmented into fixed-sized cells you should ensure that cell sizes maintain the minimum size needed to meet precharge/activate times. For example, if buffer memory is partitioned into 64-byte cells and the minimum cell size needed to meet precharge/activate times is 32, a 65-byte packet must be segmented into 32- and 33-byte cells versus 64- and 1-byte cells.

BUFFERING AT OC-192 AND BEYOND

As noted, scaling memory-interface speeds for line rates over OC-48 is challenging. The "brute-force" approach requires the use of SRAM devices. Even though SRAMs may be available with sufficient width and speed, such as QDR (quad-data-rate) clocking, to handle

OC-192 buffering, this solution is too expensive for commercial viability.

You can also speed DRAM-based buffering using advanced device technologies, such as Rambus or DDR clocking. These approaches address only memory-interface speed. DRAM-precharge/activate requirements still prevent random access at higher rates. As noted, the proliferation of sophisticated scheduling algorithms for QoS call for random access. Thus, current DRAM-based buffering is unlikely to yield the required four- to 16-times speed increase under all traffic patterns. You may find another architectural approach, such as incorporating bank-interleaving constraints in the packet-scheduling domain. TDM-scheduling techniques, in which departure schedules are deterministic, are better suited to solving these problems.

BUFFER-SPACE MANAGEMENT

Managing buffer space is as important for providing QoS guarantees in routers as bandwidth. A significant amount of study and effort has gone into the development of algorithms designed to divide buffer space between competing flows and to handle buffer-overflow situations. Two popular choices include threshold-based buffer management and longest queue-drop algorithms. Threshold-based buffer management uses queues controlled by threshold allocations based on buffer occupancy; for example, queues accept packets as long as the queue occupancy is below a programmed threshold. With the longest-queue-drop approach, packets are simply dropped from the longest queue when the buffer is about to overflow.

From a computational standpoint, threshold-based buffer management is simple. When you implement it in hardware, this approach requires only one compare operation per packet arrival. Scalability to higher rates is also straightforward.

In contrast, longest-queue-drop algorithms require a sorting operation to identify the longest queue. For this reason, these algorithms would be less popular for line rates of 10 Gbps and beyond.

Congestion management is one of the fundamental capabilities of a router. The most common congestion-management mechanism for TCP/IP routers is the

RED (random-early-detection) algorithm (Reference 2). Several variants of RED exist, but the basic RED algorithm maintains an “average queue size” based on an EWMA (exponentially weighted moving average). The algorithm drops packets based on user-programmable thresholds. Because the algorithm performs computations on every packet arrival, hardware-based implementations are preferable for applications with high data rates.

Computing the EWMA is a critical element in a hardware-based implementation of RED. With the weight factor (w_q) set appropriately, the RED gateway controls the actual queue size. When you choose negative exponents of two for w_q , you can accomplish the computation using shift and add operations. A software implementation of RED requires a lookup table to scale down the EWMA during idle periods. This implementation is unnecessary with a hard-wired approach, because you can periodically derate the EWMA.

OC-192 SPEED-UP

Despite the above-noted simplifications, running the RED algorithm at OC-192 rates and beyond is extremely difficult. One possible approach is to modify the algorithm to avoid performing computations on every packet arrival. Some variations proposed for RED do not require average computation to be performed on every arriving packet (Reference 3). The ultimate motivation for these variations has more to do with the efficacy of the end-to-end congestion detection than aspects of implementation.

ATM switches employ a different set of congestion-control algorithms; some are more appropriate for hardware implementation. Literature proposes both stateless and state-based algorithms. Stateless algorithms are likely to be more popular for high-data-rate applications than algorithms that require flow-state information.

SCHEDULING

You use scheduling to identify the order that packets are dequeued from buffer memory; for example, packets in buffer memory are written into queues based on classification criteria. Because buffering delay is the primary cause of

YOU CAN IMPLEMENT SCHEDULERS USING A NUMBER OF TECHNIQUES, INCLUDING GPS/WFQ, HIERARCHICAL SCHEDULERS, AND PRIORITY QUEUING.

packet delays in routers, the scheduler is a primary mechanism for controlling QoS in a router. As line rates increase, the scheduler easily becomes a bottleneck, especially for small packets.

When estimating the maximum speed of the scheduler, designers often overlook the overspeed factor at the fabric’s interface. If the scheduler handles packets that are buffered at the ingress memory before the fabric, it must be able to run at the fabric-interface rate.

Consider an OC-48 line card that connects to the ingress of a fabric with a 2-times overspeed factor. Because the fabric sinks data at as much as twice the line rate during periods of congestion, the scheduler must be able to schedule packets at twice the incoming line rate during these periods. For example, running at OC-48 line rates (subtracting SONET and PPP (point-to-point protocol) overhead) with a scheduler using a 125-MHz clock, a packet must arrive from the line once every 18 cycles. Thus, to maintain 2-times overspeed across the switch fabric, the ingress scheduler must be able to schedule a packet every nine cycles. At OC-192 rates, the packet arrival time drops to one every four cycles, and the ingress scheduler must deliver one packet every two cycles. The line-card egress scheduler, which handles packets from the fabric to the line, does not necessarily have the same requirements as the ingress scheduler. However, it is possible for packets to arrive from the fabric at twice the line rate.

You can implement schedulers using a number of techniques, including GPS/WFQ (generalized processor sharing/weighted fair queuing), hierarchical schedulers, and priority queuing.

Engineers generally regard GPS as an effective mechanism for enforcing QoS guarantees (references 4 to 6).

GPS/WFQ schedulers are commonly based on time stamps, which are applied to packets on arrival (optionally based on arrival time and weight). In its simplest form, this approach requires the system to maintain time stamps for every packet in buffer memory.

Alternatively, time stamps can be maintained for only the top packets in a queue, because packets contained in a given queue must be scheduled in the order that they arrive. Using this approach, the scheduler identifies the queue with the smallest time stamp and then schedules the top packet in that queue.

Class-based schedulers are simpler to implement than per-flow schedulers because a class-based scheduler must scan fewer queues (roughly equal to the number of destination ports times the number of priorities). When implementing a class-based scheduler in hardware, it’s reasonable to assume that all associated scheduling state can be stored on-chip. If you have only a few time stamps to sort, you can accomplish parallel-sort comparisons in hardware to meet required timing constraints—a relatively simple implementation.

Microflow schedulers must maintain flow states. This task typically calls for independent flow-state memory resources. You should consider the following issues when designing per-flow packet schedulers:

- How much bandwidth must be available in flow-state memory: wire rate or twice the wire rate?
- What are the performance requirements (number of cycles) for the fast-sorting algorithm?
- What granularity is necessary for bandwidth reservation? (Ideally, this granularity should be in byte increments, although this approach could increase the cost of hardware.)

You can break packets into fixed-size cells to perform bandwidth allocation at a cell level. This approach trades off implementation complexity for scheduling accuracy.

Hierarchical schedulers can pose interesting design challenges, depending on how many levels they support because the number of scheduling operations per packet is equal to the total levels of hierarchy. Generally, it’s desirable to allow the

user to modify the number of hierarchical levels supported.

FABRIC-FLOW CONTROL

Engineering literature rarely addresses the incorporation of fabric-flow control in scheduler designs. This incorporation can cause trouble for heavily pipelined implementations that schedule packets well before transmission. You can avoid this problem by incorporating mechanisms to internally discard previously scheduled flow-controlled packets and then waiting to schedule these packets until flow control is deasserted.

Calendar queues and heap sorts are two common techniques for implementing WFQ. A calendar queue consists of a clock and an array of pointers directed at lists of eligible packets (Reference 7). Each pointer indicates a slot, and the list of packets identified by the top slot is served first. For every clock tick, the pointer moves to the next slot and serves the identified packets. Multiple packets within the same slot can be linked according to the order of their arrival.

When it comes to scaling to higher rates, the two potentially largest bottlenecks in calendar queues are the mechanisms for identifying next-available slots and for performing state updates, such as linking.

You can also accomplish time-stamp sorting by using a heap structure, with the top of the heap holding the smallest time stamp. Because it takes only $\log(n)$ cycles to update the heap (where n is the number of time stamps to be sorted), a packet can be scheduled once every $\log(n)$ cycles.

SCHEDULING AT OC-192 RATES

As noted, class-based schedulers can achieve rates of OC-192 and higher by performing parallel comparisons to sort time stamps in hardware. This approach assumes that the state information the scheduler needs can be accessed in parallel.

You can use pipelining to accelerate hierarchical scheduling. For example, you can imagine a design that employs a three-level hierarchical scheduler. In the first epoch, the first-level scheduler picks a session within the second-level scheduler for a packet that departs three epochs later.

In the second epoch, the second-level

scheduler picks a session within the third-level scheduler for a packet that departs two epochs later. During the same time, the first-level scheduler picks a session within the second-level scheduler for a packet that departs three epochs later.

In the third epoch, the third-level

THE TWO POTENTIALLY LARGEST BOTTLENECKS IN CALENDAR QUEUES ARE THE MECHANISMS FOR IDENTIFYING NEXT-AVAILABLE SLOTS AND FOR PERFORMING STATE UPDATES.

scheduler picks the packet that departs in the next epoch. At the same time, the first- and second-level schedulers pick sessions within the second- and third-level schedulers, respectively.

Thus, the pipelined hierarchical scheduler effectively schedules a packet in every epoch. On the other hand, the approach requires three sets of resources that must be available for use in parallel.

With calendar queues, the primary bottleneck is identifying the next-available slot. To speed this process, you can store the active or inactive status of slots hierarchically. For example, if there is a total of N slots, all slot statuses can be stored in an $N_2 \times N_1$ array: N_2 rows with N_1 elements, where $N_2 \times N_1 = N$. If any element in a row is active, the row is considered active. Thus, there are exactly N_2 status bits, one per row, and you can store these bits using an $N_3 \times N_1$ array, where $N_3 \times N_1 = N_2$.

You can extend this process to k levels, bounded by N_1 . For better performance, you must choose a high number for N_1 . Limitations include the intended device process technology and the width of the flow-state memory. This approach allows the next active slot to be identified in k cycles.

SWITCH-FABRIC INTERFACE

Line cards typically connect to a central switch fabric using serial links. Currently, state-of-the-art CMOS-based serial-link technology can attain serial-line rates, such as 3 Gbps. You scale to higher rates by using multiple serial links per

line card. Another option is to use GaAs SiGe-based devices that can handle rates of 10-Gbps and higher.

A number of device vendors currently offer high-performance, self-routing switching fabrics that provide excellent port and rate scalability. To take full advantage of these advanced devices and gain the flexibility to choose between multiple vendors' offerings, router vendors must promote standardization of these interfaces. One effort currently under way is the CSIX (Common Switch Interface), which is working to standardize the interface between network processors and switch fabrics for OC-48 and OC-192 rates. □

AUTHOR'S BIOGRAPHY

Praveen Kumar joined Vitesse Semiconductor Corp in January 1997. He is currently manager architecture in the Network Products Division. Before joining Vitesse, Kumar was a designer in the Communications Products Division of Cirrus Logic. In 1993, Kumar received his masters from BITS (Pilani, India).

REFERENCES

1. Joo, Y, and N McKeown, "Doubling memory bandwidth for memory buffers," *IEEE Infocom* 1998.
2. Floyd, S and V Jacobson, "Random early detection gateways for congestion avoidance," *IEEE/ACM Transactions on Networking*, Volume 1, Number 4, August 1993, pg 397 to 413.
3. Firoiu, V and M Borden, "A study of active queue management for congestion control," *IEEE Infocom* 2000.
4. Demers, A, Keshav, S, and S Shenker, Design and Analysis of a Fair Queuing Algorithm, *Proceedings of ACM Sigcomm* 1989, September 1989.
5. Ferrari, D and D Verma, "A scheme for real-time channel establishment in wide-area networks," *IEEE Journal on Selected Areas in Communication*, April 1990.
6. Varma, A, and Stiliandis, D, "Hardware implementation of fair queuing algorithms for asynchronous transfer-mode networks," *IEEE Communications Magazine*, December 1997.
7. Brown, R, Calendar Queues: A fast $O(1)$ priority queue implementation for the simulation event set problem, *Communications of the ACM*, October 1998.