Scheduling Computations on a Software-Based Router[*]

Xiaohu Qie, Andy Bavier, Larry Peterson, Scott Karlin

Department of Computer Science
Princeton University
Princeton, NJ 08544, USA

ABSTRACT

Recent efforts to add new services to the Internet have increased the interest in software-based routers that are easy to extend and evolve. This paper describes our experiences implementing a software-based router, with a particular focus on the main difficulty we encountered: how to schedule the router's CPU cycles. The scheduling decision is complicated by the desire to differentiate the level of service for different packet flows, which leads to two fundamental conflicts: (1) assigning processor shares in a way that keeps the processes along the forwarding path in balance while meeting QoS promises, and (2) adjusting the level of batching in a way that minimizes overhead while meeting QoS promises.

1. INTRODUCTION

Software-based routers have always played a role in the Internet, beginning with early implementations [10]. Although there has recently been a significant focus on hardware support for routing packets at ever-increasing line speeds [9, 14], software-based routers continue to be important due to the ease with which they can be programmed to support new functionality. Pressure to extend the set of functions that routers support is happening in several different arenas:

- Routers at the edge of the Internet are programmed to filter packets, translate addresses, make level-n routing decisions, translate between different QoS reservations, thin data streams, run proxy code, and support extensible control functions.
- A new market in home routers is emerging, where in addition to running firewall and NAT code, the router is subsuming functionality that cannot be supported on computationally-weak consumer electronics devices.
- The distinction between routers and servers is blurring as routers that sit in front of clusters run application-specific code to determine how to dispatch packets to the most appropriate node.
- At the fringe, the active network research community is designing an architecture that will allow future generations of routers to run arbitrary code, thereby enabling the deployment of application-specific virtual networks.

[*]This work supported in part by NSF grant ANI-9906704, DARPA contract F30602–00–2–0561, Intel, and Cisco.

The key characteristic of a software-based router is that it implements packet forwarding as a program running on a general-purpose processor; e.g., a PC with multiple network interface cards (NICs). This paper describes our experiences implementing such a system, with a particular focus on the main difficulty we encountered: how to schedule the router's CPU cycles. Like any computing system, a router must schedule its cycles in a meaningful way—it must decide when to apply its cycles to forwarding standard IP packets, as opposed to using its cycles to re-write addresses, run a proxy, process a control message, or execute a router extension.

This paper is a follow-on to earlier work by Mogul and Ramakrishnan [11], who studied the phenomenon of livelock: a situation in which a router spends all of its time servicing interrupts at the expense of actually forwarding packets. The main way in which we go beyond this earlier work is to also consider the problem of differentiating the level of service given to different flows, while at the same time allowing each flow to be processed by a different forwarding function. We discovered that allowing the router to promise to forward certain packets at some sustained rate introduces a fundamental tradeoff between achieving maximum performance for best-effort packets, and being able to keep QoS promises. This tradeoff manifests itself in two different ways: (1) how to best assign processor shares, and (2) how to adjust the level of batching.

This paper makes two contributions. First, it describes the relevant features of a software-based router that combines support for both QoS and extensibility (Section 2). Second, it discusses the two conflicts identified above, (Section 3–4), where for each conflict, we present experiments that provide insight into the problem and help to evaluate the effectiveness of our approach.

2. EXTENSIBLE ROUTER

This section describes the router we implemented. Our design generalizes existing software-based routers in that it supports both extensibility and service differentiation. We built the router using the Scout operating system [13].

2.1 Process Architecture

The simplest possible architecture for a software-based router implements all router functionality in a single process. Such a process executes the following loop:

```
READ:      read a packet from an input port
CLASSIFY:  select an output port
PROCESS:   perform required packet processing
WRITE:     write packet to the output port
```

where this single process services the various input ports according to some policy (e.g., round robin). This simple architecture is important because it represents the most efficient base case, but it has two serious limitations. First, it processes packets in FIFO order, and so is unable to differentiate the level of service it gives different packet flows. Second, if the PROCESS step is unable to complete in a small/fixed amount of time, the process is not able to read packets off the input ports at line speed, and thus risks having packets dropped by the input port.

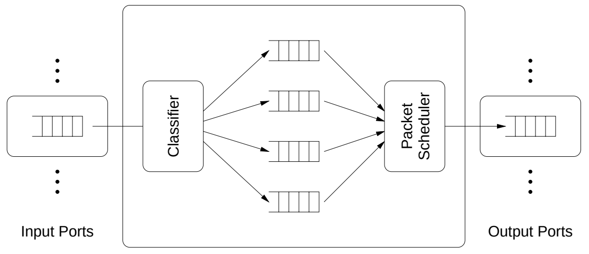

Figure 1: Supporting Differentiated Service

Recognition of the first limitation has prompted the architecture shown in Figure 1, where the key idea is to segregate incoming packets into multiple queues. Our architecture addresses the second limitation by adding a third stage to the packet pipeline. The result is shown in Figure 2, where for simplicity, we focus our attention on a single input/output port pair. The following discusses each stage in more detail.

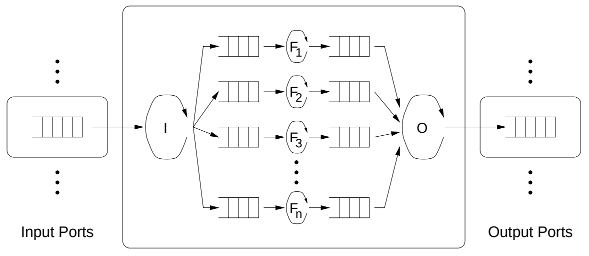

Figure 2: Supporting Differentiated Service and Variable Processing

At the first stage, an input process (denoted **I**) executes the following loop:

```
READ:      read a packet from the input port
CLASSIFY:  classify the packet
ENQUEUE:   enqueue packet on appropriate queue
```

Although not shown in our simple router, we assume a separate input process for each input port. This places responsibility for selecting which input port to service next with the process scheduler, as opposed to embedding the policy in the input process.

At the third stage, an output process (denoted **O**) associated with each output port performs the following loop:

```
SELECT:    select queue for next packet to transmit
DEQUEUE:   dequeue the packet from this queue
WRITE:     write the packet to the output port
```

In this case, the link scheduling algorithm is embedded in the SELECT step of the output process, meaning that this process has to run in order for a packet to be selected for transmission.

The middle stage in the pipeline, corresponding to processes F_1 through F_n in Figure 2, performs whatever processing the packets require. We say each of these processes implements some *forwarding function* F. In the simplest case, this forwarding function manipulates the TTL and checksum fields of the IP header and modifies the link-level header. In general, any number of different functions might be applied to a packet—standard IP forwarding, IP option processing, control processing, proxy code, router extension, active code, and so on. Each of these different forwarding functions has different processing requirements. Table 1 gives a representative sample of forwarding functions we have implemented, where "Active Protocol" corresponds to an active capsule running in the ANTS active network environment [18] on our router.

Forwarding Function	Per-Packet Cost (μs)
IP Fast Path	0.3
General IP	3.0
Transparent Proxy	10.7
Classical Proxy	12.8
Active Protocol	37.3

Table 1: Costs of various forwarding functions, measured in microseconds, on a 450 MHz Pentium II. These times are independent of the costs of the input and output processes.

There are two general questions about the processes that implement these forwarding functions. The first is why we need any processes at all; why not just execute these functions as part of the input or output processes? The problem with moving the forwarding function to the input process is that it may take an arbitrary length of time to execute, thereby causing the input process to not keep up with link speeds. Postponing this function to the output process suffers from much the same problem: there may be an arbitrarily long delay between when a packet is selected for transmission and it can actually be sent, and packet schedulers do not take such delays into account. The packet scheduler assumes that the selected packet is immediately available, and any delay in preparing the packet may cause the link to become idle.

Once we have determined that we need a third process in the pipeline, the second question is how many different forwarding processes are required. Here we have several options. One is to dispatch a process for every packet. That is, the classifier running in the input process produces ⟨packet, function, queue⟩ triples, and assigns a process to each such triple. When the process runs, it applies function to packet and enqueues the result in the specified queue. There are at least two problems with this *process-per-packet* approach. First, it results in a potentially huge number of processes—tens or hundreds of thousands per second—which is well beyond the design of most thread packages. Second, rather than having all messages contained in one message queue or another, messages are "hidden" in the thread queue. This makes it much more difficult to reason about the system's behavior.

At the other extreme, a single process could perform all the required packet processing. As before, each classifier produces ⟨packet, function, queue⟩ triples, and enqueues them with this forwarding process. The obvious problem is that packets belonging to flows that have been promised a particular level of service can be queued behind best effort packets. In effect, this single forwarding process ignores the separation of flows achieved by the classifier. This approach should not be discarded too quickly, however, because it works perfectly well for best effort flows which can live with the FIFO queue that this forwarding process services.

We settle on a compromise approach that establishes a separate forwarding process for each flow. In effect, this model isolates all the *switching paths* through the router. The input process dispatches each packet to a single switching path that consists of an input queue, a forwarding process, and an output queue. The output process then determines from which switching path a packet should be transmitted next. Exactly what constitutes a flow is a policy question. Certainly each QoS flow is treated as a distinct switching path—and thus has its own forwarding process—even if it involves the same function F as some other path. On the other hand, multiple best effort flows that share the same forwarding function are assigned to the same switching path.

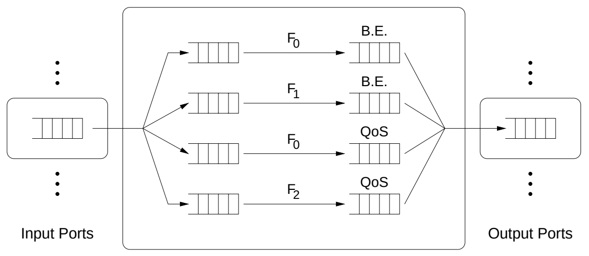

Figure 3: Example Switching Paths

Figure 3 illustrates an example set of switching paths. It is representative of the cases we study in the next two sections. To simplify this diagram, we replace the process that implements each forwarding function with a labeled edge that connects the input queue to the output queue, and neither the input or output processes are explicitly shown; they correspond to the demux (classifier) and mux (packet scheduler) points, respectively. Also, the output queues are labeled according to how the link scheduler treats its packets—best effort (BE) or QoS—and we assume the minimal IP forwarding function F_0 and two non-standard functions F_1 and F_2. Thus, this simple router—again focusing on a single input/output port pair—includes a best-effort/minimal switching path, a QoS/minimal path, a best-effort/non-standard path, and a QoS/non-standard path. Note that the two best effort paths aggregate many end-to-end flows, while the QoS paths carry a single end-to-end flow.

While we restrict our discussion to a router with a single processor, it should be noted that Figure 3 maps to an environment with multiple processors. VERA [8] uses a hierarchical processing model where the input and/or output processes run on the processors of intelligent NICs. This allows the main processor (the Pentium II in our case), to spend more of its cycles on forwarding functions.

We conclude this discussion by noting that there is a question of exactly where to draw the line between the CLASSIFY step in the input process and the processing done in the forwarding process. The answer is that, by definition, classification is that processing which can be completed in a fixed number of cycles—selected so the input process is able to match the link speed—and packet processing is everything else. This means that to implement application-level classification, which may take an arbitrary length of time, the input process partially classifies the packet and selects some function F to complete the classification. It also means that the input process could implement the minimal forwarding function, cycle budget permitting, although our scheduling framework argues for minimizing the work done in the input process.

2.2 Performance

We conclude our description of the base system by reporting a series of experiments designed to determine whether or not our process architecture is prohibitively expensive in practice. The machine running our prototype router has a 450 MHz Pentium II processor with a 512 KB L2 cache, and three Tulip (21143 chip) 100 Mbps network interface cards. Three additional PCs—labeled A, B, and C in the following discussion—serve as packet sources.

During the tests, each source PC generates a stream of 64-byte IP packets at rates up to 140 Kpps. Using all three sources, we can generate an aggregate maximum offered load of 420 Kpps. We measure the time the router takes to forward a certain number of packets, yielding an average forwarding rate. While this artificial workload is clearly not representative of the Internet at large, our experiments are designed to stress the CPU rather than the network. It is for this reason that our experiments emphasize switching small packets; a larger number of small packets place a greater load on the CPU than fewer large packets.

We measure the relative overhead by plotting the offered load to the router versus the forwarding rate achieved by the router. The parameters we chose to vary for this series of experiments were:

Input Servicing Scheme This is the choice to use interrupt driven input versus input device polling.

Number of Processes We use one, two, or three processes to forward the packets from input to output. When we use one process, a single thread handles input, forwarding, and output. When we use two processes, we use an input thread and a forward/output thread (this case allows different forwarding functions but no link scheduling). For three process experiments, each of the input, forwarding, and output tasks are assigned to its own thread.

Batching To reduce the overhead of context switches, we can enable batching. With batching enabled, each forwarding process attempts to handle as many packets as possible up to an arbitrary limit of 16 packets. Without batching, each process will handle at most one packet before yielding the processor. (The input and output processes always batch.)

Figure 4 summarizes the results of five experiments. In each experiment, the source PCs gradually increase their aggregate offered load from 0 to 420 Kpps, while the router attempts to forward the packets as best it can.

In the interrupt implementation, the router is able to keep up with the sender up to 48 Kpps, after which it begins to suffer from receive live-lock. In contrast, the polling implementations give a more desirable behavior: the forwarding rate increases up to a certain point and then remains flat. In the flat portion of the graph, the router drops more and more packets; however, the router does not waste time on the dropped packets. These results simply reinforce those found in [11].

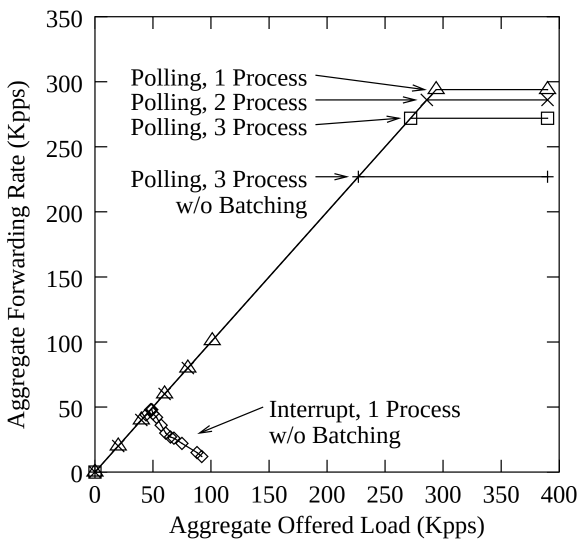

Figure 4: Impact of interrupt handling and context switching on forwarding rate.

Processes	Maximum Batch Size	Max. Forwarding Rate Kpps	Max. Forwarding Rate Normalized
1: I+F+O	16	294	1.00
2: I, F+O	16	286	0.97
3: I, F, O	16	272	0.93
3: I, F, O	1	227	0.77

Table 2: Maximum Forwarding Rates using Polling

Table 2 shows the relative maximum forwarding rates using polling. The forwarding rates are from the plateaus in Figure 4. The last column shows the forwarding rates normalized to the single process with batching case. From this table, we see that each additional process in the forwarding pipeline adds 3 to 4% overhead. The effects of batching are more significant, improving performance by approximately 16%. Further analysis shows that we are batching on the order of 10 packets at each stage. Comparing three processes with batching (i.e., our proposed architecture) to the single process case, we see that the overall difference in performance is only 7%, which seems a tolerable overhead for the increased functionality our architecture provides.

Although raw performance is not the focus of this paper, the total 3.3 μs forwarding time for each packet did not come easily. Of the optimizations we made, the most important to the results presented in this paper was to implement context switches as inexpensive continuations. Having two full-blown context switches on the forwarding path has the potential to add 10 μs to the forwarding time.

3. SCHEDULING DISCIPLINE

The previous section largely ignores the issue of scheduling, and in fact, a simple round-robin scheduler could have been used to produce the performance results in Figure 4. As we consider more complex scenarios, however, we must put more thought into the scheduling decision. In general, the task of the scheduler is to select the next process for execution in a way that leads to desirable behavior along four dimensions: (1) efficient best effort forwarding which makes good use of available resources; (2) different qualities of service to flows that require more than best effort; (3) robust behavior in the presence of overload, including packet flooding denial of service attacks; and (4) support for switching paths of varying computational costs.

3.1 Proportional Share

We use a proportional share (PS) scheduler to meet these four competing goals. PS is a general scheduling discipline that provides a cycle rate to a process; it abstracts the main features of a class of algorithms, such as Weighted Fair Queuing [3]. The essential characteristics of PS are:

- Each process reserves a cycle rate—e.g., 1 million cycles-per-second (Mcps)—and is guaranteed to receive at least this rate when it is not idle.
- Unused and unallocated capacity is fairly distributed to active processes in proportion to each process's reservation. An active process that receives extra cycles beyond its reservation is not charged for them.
- An idle process cannot "save credits" to use when it becomes active. Unused share is simply lost.
- The guarantees made to processes provide *isolation* between them—each process gets its rate no matter what the other processes do.

Proportional sharing maps naturally onto our process architecture, and accomplishes the varying goals we have set for our router. First consider QoS flows, where share assignment is straightforward given some knowledge about processing costs. Every flow traverses a pipeline of three processes (i.e., input, forwarding, and output), and we need to set the process shares so that the pipeline forwards data through the system at the flow's reserved rate. The processing costs for reading a packet in the input process, and sending it in the output process, are fixed and known in advance. Therefore, we can determine the amount that the shares of these processes need to be increased to accommodate the packets of the new QoS flow. If we assume that we know the cost function for the flow's forwarding process (e.g., how many cycles per bit it requires) then the share of the forwarding process is obvious too. We return to the question of how to determine the cycle rate required by forwarding functions in Section 3.5; for now, we assume that forwarding functions have regular costs that can be determined through off-line experimentation, as illustrated in Table 1.

Unlike QoS flows, no hard commitments are made to best effort flows, but shares are still useful for producing good system behavior. We observe that live-lock and poor overload behavior are actually problems of *balance*—one component of the system is receiving more cycles than is desirable, with the result that other components get too little. Our implementation provides good best effort performance by assigning shares to different pipeline stages based on the ideal balance of the system in high load. For example, micro-experiments run on the configuration described in the previous section indicate that in the three-process case, the input process spends 1.6 μs on each packet, the forwarding process spends 0.3 μs on each packet, and the output process spends 1.4 μs on each packet. Thus, the results shown in Figure 4 were achieved with a balanced share assignment of 5:1:5 (I:F:O) for the three-process case. Similarly, for the two-process case—an input process and a combined forwarding/output process—a balanced system has almost exactly a 1:1 ratio.

It is important to keep in mind that a process receives its share only if it has work to do. For example, each forwarding process shown in Figure 3 is allowed to run only when its input queue is not empty and its output queue is not full. We say a process that meets these conditions is *eligible*.

3.2 Input Process Share

Focusing on best effort or QoS flows in isolation makes share assignment easy. Unfortunately, a conflict arises when we consider a router that supports both types of flows. The issue is what share to give the input processes. For the sake of best effort flows, we want to assign input process shares based on the ideal (balanced) cycle distribution in overload, since giving too big of share to the input process potentially leads to live-lock. On the other hand, should this rate be less than is required to read and classify packets at line speed, QoS flows are vulnerable to denial of service attacks. This is because packets belonging to well-behaved QoS flows may be dropped on a line card if the input process does not have enough share to keep up with packets arriving at line speed.

It is not clear how to assign a share to the input process to best satisfy both kinds of flows. The system designer must make a fundamental tradeoff when choosing the input share. Our approach is to favor QoS flows by giving the input process a conservative share, but we temper this decision by adding a *queue estimator* mechanism.

Before describing this mechanism, we illustrate the problem faced by QoS flows under unexpected load. Figure 5 shows an experiment in which a QoS flow shares the incoming link with best effort traffic. The QoS flow has reserved a rate of 49 Kpps, and this same amount is dedicated to the best effort packets. The input process therefore receives a share sufficient to read 98 Kpps from the interface. At the start of the test, the QoS flow is sending packets at its reserved rate, while the best effort traffic is within expectations at 39 Kpps.

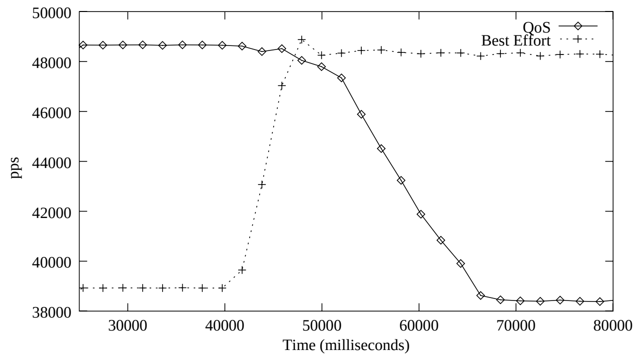

Figure 5: QoS experiencing drops under heavy load

At time 40000 in the experiment, the best effort traffic rate increases at 2000 pps per second up to a peak rate of 84 Kpps. The router only forwards about 49 Kpps of these packets because this consumes all the cycles allocated to best effort. However, the input process was only given a large enough share to read 98 Kpps from the interface. Since packets are arriving at a total rate of 133 Kpps, many are dropped on the line card, including packets belonging to the QoS flow. Figure 5 shows the forwarding rate of the QoS flow dropping down to a rate about 10 Kpps less than it has reserved. Though the QoS flow is well-behaved, the router is not honoring the promise it has made to provide service to the flow. This is a result of not setting aside enough cycles for the input process to handle the worst case.

Returning to the question of how this strategy affects best effort traffic, we observe that there are two risks to running the input process at line speeds. Under heavy load, it is possible that the router will waste cycles reading packets that are dropped later in the pipeline. There is nothing we can do about this if we want to protect QoS flows. Under a light input load, the risk is that the input process—which is essentially a polling thread—runs even though there is no useful work for it to do. This is bad, because the router itself may still be heavily loaded; that is, other interfaces may be servicing high packet volumes even though this one is not. Previous work [5, 11] examines switching between polling and interrupts based on load to handle this situation. The queue estimator addresses the same problem from a different angle.

To understand the queue estimator, consider that the process architecture is designed to ensure that all messages are buffered in explicit message queues, as opposed to hidden thread queues. Our current framework uses these queues to decide process eligibility, where a process is eligible to run only if its input queue is not empty and its output queue is not full. Note that calculating the eligibility of an input process is not straightforward since its input queue resides on the device. It may be possible to exploit a NIC status register that contains the number of packets buffered on the device. However, a process must read this register—we do not want the scheduler itself doing device I/O—and we must still decide when to schedule this process. This does not solve the problem.

The goal of the queue estimator is simply to estimate the device queue length based on previous observations. It does this by keeping a weighted average of the packets read during each execution of the polling thread. Each input process also has a sleep interval and a target range. If the weighted average of packets is less than the target range, the sleep interval is increased (up to some maximum); if it is greater, the interval is decreased. The process then sleeps for that interval before becoming eligible to run again. Note that this mechanism allows the input thread to adjust its rate to read a batch of packets every time it runs; we will discuss this point further in Section 4. The point is that the state of all queues, including the receive queue on the device, is available to be incorporated into the scheduling decision.

3.3 Robustness

We now turn our attention to the performance of our router when forwarding only best effort packets. We show that the system is robust, in that it achieves good best effort forwarding rates even when the system becomes imbalanced (i.e., the shares are not quite right). We test this situation by varying the cost of the forwarding process, where described in Section 2.2, we know that the input requires 1.6 μs to read and classify each packet.

In this experiment, packet flows from the three sources traverse three different switching paths, denoted A, B, and C. Flow A uses a forwarding function that spends 8.0 μs on each packet, meaning that it has an ideal share ratio of 1:5. The forwarding functions for flows B and C delay their packets by an additional 8.0 μs and 16.0 μs, respectively, meaning that their ideal share ratios are 1:10 and 1:15. The fourth column of Table 3.3 shows the forwarding rate that is achievable for these three flows, where the three flows run serially (i.e., they are not competing with each other). This table gives only the peak forwarding rate for each flow, which corresponds to the measured throughput rate at the maximum sending rate of 140 Kpps. The maximum forwarding rate of

Flow	Fwd Cost	Ideal balance	Balanced share	1:10 share w/o estimator	1:10 w/ estimator
A	8 μs	1: 5	101 Kpps	101 Kpps	101 Kpps
B	16 μs	1:10	56 Kpps	56 Kpps	56 Kpps
C	24 μs	1:15	38 Kpps	35 Kpps	38 Kpps

Table 3: Best effort throughput in Kpps

each host's packets are very close to what we would expect. The system is allocating the CPU efficiently.

Next, instead of configuring the router to give each flow its ideal share, we set the ratio to 1:10, meaning that flow B is balanced, while flow A gives too much weight to the forwarding process and flow C gives too much weight to the input process. There are two situations where such an imbalance might arise in practice. One is that the amount of processing required varies from packet to packet, so we can establish only the average CPU allocation for a given flow. The second is that we give the input process a conservative cycle rate—perhaps for the sake of protecting QoS promises—but it can't effectively use this rate. Flow C corresponds to this latter case.

The fifth and sixth columns of Table 3.3 show the results of the imbalance. The difference between the two columns is that the fifth drops packets that demux to a full queue (as we might do if we had QoS flows), while the sixth uses the queue estimator of Section 3.2 to limit the rate at which the input process runs (appropriate with no QoS flows). Not surprisingly, flow B performs exactly as before, since it is given the same share assignment. Though the shares are out of balance for flow A, it is unaffected, since the forwarding process's unused share is distributed upstream to the input process (recall that PS fairly distributes unused share). However, flow C's throughput drops to 35 Kpps in the fifth column. The problem is that the input process is running at a faster packet rate than the forwarding process causing packets to drop off the tail of the forwarding process's input queue. Column six for flow C shows that eligibility and throttling do readjust the rate of the input process to match the forwarding process.

This experiment doesn't really demonstrate the impact of assigning a conservative share to the input process(es) in an effort to protect against a flood of best effort traffic. In another experiment, we configured three input ports, two of which had no traffic arriving and one on which packets arrive at full speed. We gave each input process a large enough CPU share to receive packets at line speed, and we set the processing rate for each packet to 6 μs, which fully utilizes the CPU. Without the queue estimator, roughly 30% of the CPU is wasted polling idle input ports, thereby yielding a forwarding rate of 91 Kpps for the active port. With the estimator enabled, the router was able to forward packets at 130 Kpps, the maximum achievable rate for this configuration.

3.4 Best Effort Policy

Another issue is the extent to which best effort traffic should be aggregated versus isolated. As described in Section 2.1, each unique forwarding function is assigned to a separate switching path (forwarding process). The alternative is that a single switching path services multiple forwarding functions rather than being limited to just one. The latter approach reduces the state the router must maintain, but the former approach makes it possible to assign each switching path a different processor share.

For example, a router might establish a policy that best effort packets that require option processing should be segregated from best effort packets without options, with the former receiving preferential treatment by the scheduler. As another example, one could ensure that forwarding functions that process route updates receive a sufficient share. This is effectively an attempt to differentiate service, just as with QoS flows, but based on functionality rather than bandwidth requirements.

3.5 Determining Cycle Rates

The final issue is how to determine the cycle rate that each forwarding function requires. This rate is needed to reserve the appropriate share and to implement admission control. This section describes our experiences to date.

We have been describing the rate of a flow in terms of packets per second, yet most QoS schemes provide rates in terms of bits per second. We assert that most QoS applications send packets of roughly equal size, and so it is possible to translate a bit rate into a packet rate. Our experience also suggests that the number of cycles required to process each packet (or per byte of data) can be accurately measured, and hence, it is possible to compute a cycle rate. If these assumptions hold, then it should be possible for a router to derive the cycle rate from an RSVP-style bit rate reservation. If not, then it may be necessary for the application to explicitly state the cycle rate it requires, and if the computation is data-dependent, this reservation may need to be conservative.

The next question, then, is whether it is necessary to make a conservative reservation for data-dependent flows. To maximize resource usage, it would be better for each application to reserve the average cycle rate its packets require. The worry is that even though the forwarding process receives its reservation over a long interval, any given packet might arrive at the output queue late, and hence forfeit its share of the link capacity. We note that this is exactly the same problem as the presence of jitter in the arrival rate of packets, the effects of which can be mitigated with sufficient buffering. In other words, as long as a switching path's output queue is large enough, it can buffer packets produced during good times (when processing costs are small), and therefore not go empty when processing cost are large.

4. BATCHING

Our design requires every packet be handled by three threads, which if implemented naïvely, means that the router performs three context switches for every packet it forwards. A context switch in Scout costs 2 μs on the prototype hardware, which implies 6 μs of overhead for each packet. When compared to the 3.3 μs required to process a minimal IP packet—the sum of the time spent in the input, forwarding, and output processes—this overhead is significant. The obvious solution is to batch packets, so that the cost of each context switch can be amortized over multiple packets. Table 2 in Section 2.2 shows that turning batching off for just the forwarding process (both the input and output processes still batch) drops the forwarding rate from 272 Kpps to 227 Kpps. Unfortunately, batching can have a negative impact on other aspects of the system, as discussed in the rest of this section.

4.1 Batching and Granularity

Though batching leads to better performance, we must be careful because it leads to coarser scheduler granularity. A QoS flow with finite buffer size requires its CPU share to be delivered within a certain period, that is, before its queue becomes full. With large batch sizes, a thread may be able to hog the CPU for a long period of time, thereby delaying the execution of other flows. If the delay (or service lag) exceeds the maximum another flow can buffer, future packets belong to this flow will be dropped even if they are within the flow's reservation. Under such a scenario, the contract between the system and the flow is violated.

To verify our conjecture, we performed an experiment that explores the transitions from the router being link-bound to being CPU-bound. We use the same experimental setup as in Section 2.2, and we refer to the packets coming from source x as "flow x". Flows B and C are QoS flows sending at 90 Kpps (they have reserved this rate), while flow A is best effort. Flow A arrives on port 1 of the router, flow B arrives on port 2, and flow C on port 3. The router forwards flow B to output port 1, and both flows A and C are routed to port 2. A and C compete for the same output link; the link is saturated when flow A transmits at 50 Kpps. Running flow A at this rate, we then measure the performance of all three flows as the processing time for flow A's packets increases. The system transitions from being link-bound to being CPU-bound at just before $5\,\mu$s on the x-axis in the following results.

Figure 6 shows the forwarding rate for the flows when we enable simple batching. We allow each thread to run until it has processed up to n packets, where n is the maximum batch size. As flow A's cost increases, it hogs the CPU while it works on a batch of packets in the queue. Because flow A gets the CPU in large bursts while it processes a batch of packets from its queue, the input threads of flows B and C do not get to run often enough and their packets are dropped on the line cards, resulting in loss of share.

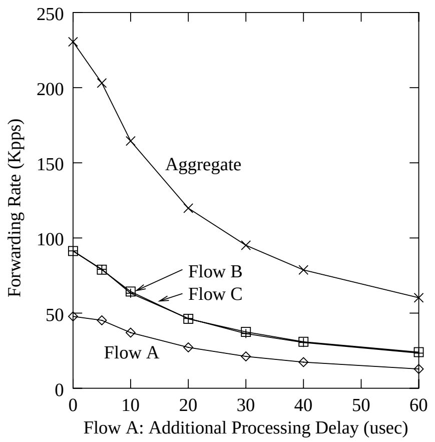

Figure 6: Detrimental Effects of Simple Batching as Processing Costs Increase.

Realizing the cause of the problem, we turn off batching. In Figure 7, both flows B and C are able to maintain their packet rate. Obviously, A's rate decreases as it spends more time on each packet. The improvement comes from the fact that the proportional share scheduler regains control at a finer granularity. As we have seen, though, turning batching off comes with a cost. To quantify the effect, we use a measure called the *Efficiency Index* (E), which is defined to be the percentage of CPU cycles actually used to process packets (those spent in input, forwarding and output threads) among all cycles consumed (including cycles spent context switching or making scheduling decisions). Conceptually, a higher E means less scheduling overhead. In this experiment, E is 66.2%.

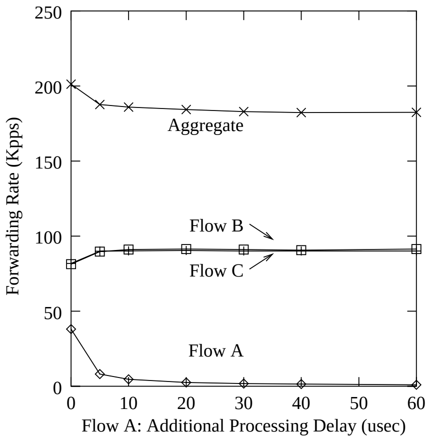

Figure 7: No Batching

In Figure 8, we turn batching back on but we only allow batching to process 16 packets or process for $30\,\mu$s, whichever comes first.[1] We see that QoS flows B and C meet their reservation and that the aggregate forwarding rate is higher than that in Figure 7. In this figure, the measured E is 81.6%.

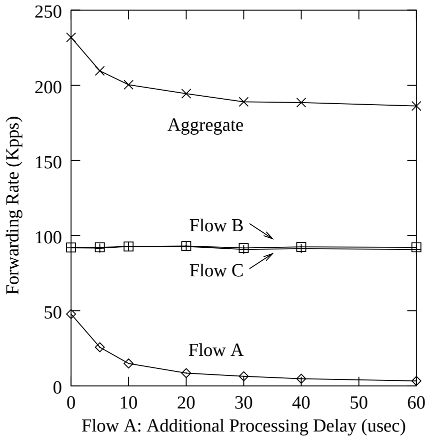

Figure 8: Batching with a $30\,\mu$s limit.

The lesson learned is that the PS scheduler needs to get a frequent enough opportunity to re-schedule the processor in order to preserve the rate requested by QoS flows. Carefully controlled batching provides this opportunity without sacrificing performance.

[1]Note that a switching path is not preempted; whether or not the timeslice has expired is checked once per path execution. For the sake of this paper, we assume that only well-behaved functions are admitted to the system. Untrusted functions could either run in a preemptable process, or the system could kill a packet if it exceeds a certain time limit. Both facilities could easily be added to our architecture.

4.2 Batching Throttle

This section describes a *batching throttle* mechanism that considers each flow's processing cost and queue length, and the system's scheduling overhead. It can dynamically adjust the level of batching to trade off granularity for efficiency, and at the same time meet QoS promises.

G	Scheduler Granularity. A flow can process as many packets as possible within timeslice G but must yield the processor when it expires.
E	Efficiency Index. The percentage of CPU cycles that are actually used to process packets, as opposed to run scheduler, etc.
T	Overhead Threshold. $E = 1/(1+T)$.
Csw	Average Context Switch cost.
Cpp_i	Average per packet processing cost of $Flow_i$.
B_i	Batch Threshold of $Flow_i$, the minimum integer satisfying $\frac{Csw}{B_i \times Cpp_i} \leq T$.
D_i	Delay Threshold of $Flow_i$. Indicating packets of $Flow_i$ should be processed no later than time D_i after being enqueued.

Table 4: Notation used by the batching throttle

Table 4 contains notation used in describing the batching throttle. Context switch cost Csw is a constant that can be measured in advance. G, E and T are system parameters that can be configured by the administrator, while Cpp_i, B_i and D_i are flow-specific variables.

The batching throttle affects the system's behavior in three ways. First, it preserves a specific scheduler granularity by requiring threads to surrender control at least every G time units. This addresses the problem associated with simple batching mentioned in Section 4.1. Second, it tries to schedule threads that can process a full batch when they run, in an attempt to improve the efficiency of the system. If a thread processes B_i packets before it yields the processor, the context switch that follows will add no more than $Csw/(B_i \times Cpp_i)$ overhead, i.e. be bound by T. Third, it allows for flows with latency requirements by including a timeout mechanism: a flow that is not efficient after waiting D_i is allowed to run anyway. We now explain how the batching throttle works in detail.

A thread can be in one of four possible states: Idle, Eligible, Active and Running. The scheduler chooses threads for execution from those that are Active. The Eligible state now signifies that the thread for $Flow_i$ has fewer than B_i packets in its input queue or fewer than B_i empty slots in its output queue, and is therefore considered inefficient to run. (For the time being, we consider the forwarding threads only, and therefore each thread corresponds to a single flow.) A transition from Eligible to Active is triggered by two events. The first is a wakeup call,[2] in which case the thread becomes Active if its input queue length and output queue slots are at least B_i. The second is a timeout, which occurs once the thread has spent D_i in the Eligible state. Figure 9 shows the thread transitions between the four states.

[2]Wakeup happens upon every packet arrival at the input queue and removal from the output queue. However, only one wakeup does the real work of restoring thread states, moving it from Idle to Eligible state. Once the thread is in Eligible state, subsequent wakeups only perform a check on the queue length.

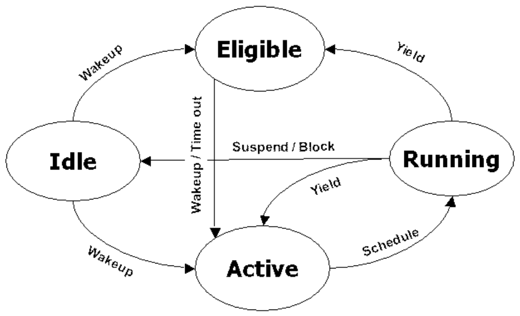

Figure 9: State Transition

The behavior of the batching throttle is mainly characterized by two parameters: Scheduler Granularity G and Overhead Threshold T. They are subject to the following constraint:

$$G \geq \frac{Csw}{T} \qquad (1)$$

This is because under the control of batching throttle, execution of B_i packets of $Flow_i$ should not exceed the system's timeslice, therefore $B_i \times Cpp_i \leq G$ holds for every flow. Substituting in the definition of B_i will give us equation 1.

Equation 1 formulates the conflict between fine-grained scheduling and lower scheduler-related overhead. In particular, improving system throughput by lowering the overhead threshold T must result in coarser scheduling granularity G. Previous fair queuing studies tell us coarser granularity means performance deterioration in terms of fairness and latency.

An administrator can choose any values for the batching throttle parameters as long as they obey this constraint. We would like to stress that with the batching throttle mechanism, it is possible to *dynamically* adjust system behavior in response to workloads. That is, rather than setting values for G and T that are always enforced, the system tries to choose the best values based on observed load. This is done as follows. The system calculates the E required to forward all packets based on the current workload and available cycles, from which it can calculate the overhead threshold T. An appropriate G value can then be derived by Equation 1. When the load is low, E is small so a large T is acceptable. This means that we can run the scheduler at finer granularity to improve the fairness and latency each flow experiences. As the input load increases, there comes a point at which the system will not be able to admit a new flow without improving the efficiency E. If maximizing throughput is the system's goal, it can lower T in order to make room for the new flow. G will have to be adjusted accordingly and the the result is longer delay and coarser control.

We need to address two issues when G is increased to a larger value. First, the service lag introduced by coarser granularity should not lead to the problem of Section 4.1. Since B_i increases with G, B_i approaching $Flow_i$'s queue length can be viewed as a warning that the lag is likely to cause the flow to drop packets. The system then may want to allocate more buffers to counter the effect, thereby trading memory for CPU efficiency. Second, for a QoS flow with tighter delay requirements, waiting for its B_ith packet to arrive before considering it Active is not an appropriate thing to do. The batching throttle includes a delay parameter D_i with each flow. If a thread has spent D_i in the Eligible state, it times out and becomes Active. Since the scheduler now has to run an "inefficient" thread, the system may not be able to achieve the efficiency it planned for. Therefore, such timeouts must not be uncontrolled. We can imagine that we only grant this privilege to a few QoS flows.

As an aside, although the batching throttle focuses on forwarding threads (whose processing costs have the greatest variance), it can be applied to input and output threads as well. In fact, the throttle fits quite naturally with the queue estimator described in Section 3.2. Recall that since the queue on the line card is not visible, the queue estimator adjusts the sleep interval of the thread based on previous observations. In the language of the batching throttle, the system cannot see when the B_ith packet arrives, so it guesses how long until it arrives and sets D_i accordingly. This dynamically adjusts the actual execution rate of the input thread in response to the workload, helping performance by avoiding unnecessary polling.

4.3 Evaluation

In this section we present experimental results designed to demonstrate the effectiveness of the batching throttle. We have implemented and tested the mechanism on our prototype system. We also developed a simulator which is used to produce results presented in this section. The simulator captures all scheduler related events. It allows us to log detailed tracing information without incurring overhead which on a real system would interfere with scheduling decisions. We configured 13 flows, with the flow specifications shown in Table 5.

Flow	Pps	Cpp	Type	$D_i\,(\mu s)$	Share
1	5000	$1\,\mu s$	BE	N/A	0.5%
2	10000	$1\,\mu s$	BE	N/A	1.0%
3	20000	$1\,\mu s$	BE	N/A	2.0%
4	5000	$3\,\mu s$	QoS	N/A	1.5%
5	10000	$3\,\mu s$	QoS	N/A	3.0%
6	20000	$3\,\mu s$	QoS	N/A	6.0%
7	5000	$3\,\mu s$	QoS	2000	1.5%
8	10000	$3\,\mu s$	QoS	1000	3.0%
9	20000	$3\,\mu s$	QoS	500	6.0%
10	5000	$10\,\mu s$	BE	N/A	5.0%
11	10000	$10\,\mu s$	BE	N/A	10.0%
12	20000	$10\,\mu s$	BE	N/A	20.0%
13	100000	varies	BE	N/A	3.0%

Table 5: Flow Specifications

Intuitively, Flows 1-12 can be viewed as separate end-to-end flows executing different forwarding functions. Some of them (type QoS) make a reservation. Despite the flow type, the CPU share each flow gets is equal to its packet-per-second $\times$ cost-per-packet product. Flow 13 represents an aggregate of Best-Effort traffic. The system administrator allocates 3% of the CPU to it, regardless of its per-packet-cost. We leave the per-packet-cost of Flow 13 to be a parameter that varies for this series of experiments, thereby varying the pressure exerted on the CPU.

During the test, each flow is generating evenly spaced packets according to its packet rate. The results are based on data collected during a 0.1 second period. Within this period, the router receives 500 packets from Flow 1, 1000 packets from Flow 2, ..., and 10000 packets from Flow 13. The number of buffers allocated to each flow is set to 128, unless stated otherwise. The average context switch cost is modeled to be $3\,\mu s$. We evaluate three mechanisms that control the level of batching: (1) simple batching using a predefined limit (Figure 6); (2) timeslice (Figure 8); and (3) batching throttle.

4.3.1 *Fixed Batching*

In the first series of experiments we set the batch limit to be $1, 8, 16, 32$ and 64, respectively. For each batch limit, we repeat the test with $3\,\mu s$, $10\,\mu s$, and $30\,\mu s$ as Flow 13's per-packet-cost. We measured the system's efficiency, packets dropped from Flow 13 and maximum queue size seen by each flow. The results are summarized in Figures 10, 11 and 12.

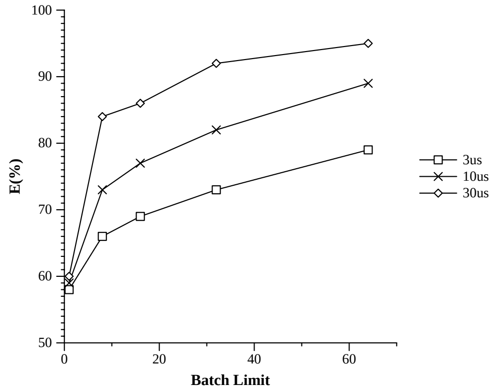

Figure 10: Efficiency Index for simple batching.

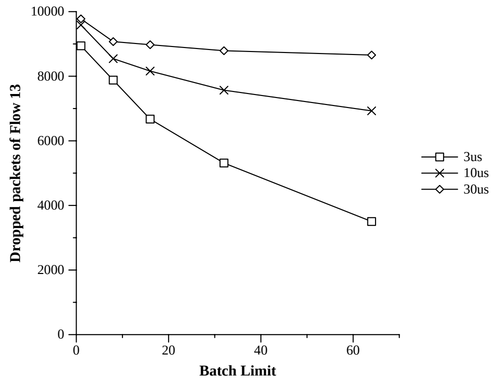

Figure 11: Flow 13's packets dropped for simple batching.

We see the number of drops decreases as the batch limit increases. For a fixed batch limit, E also increases as we increase Flow 13's per-packet-cost. These results are not surprising as the number of context-switches are reduced.

The results in Figure 12 deserves more explanation (We use Flow 6 as an example, the results for other flow are all similar). When we set the batch limit to 1 (no batching), Flow 6 drops packets (the dot-dashed line in Figure 12 indicates the number of buffers allocated to each flow). However, as we increase Flow 13's per-packet-cost, the dropping stops. This result can be explained by the increase in E, as shown in Figure 10. A more detailed explanation is that when the processing of every packet has to be followed by a context switch, the system does not have enough cycles to service flows at their reserved rate. The significant scheduling overhead overloads the system, even though the allocated shares only add up to 62.5% of total capacity. Since the context switch cost isn't charged to anyone, the shares each flow actually receives is less than what it should get. In all the experiments, the actual cycles consumed by

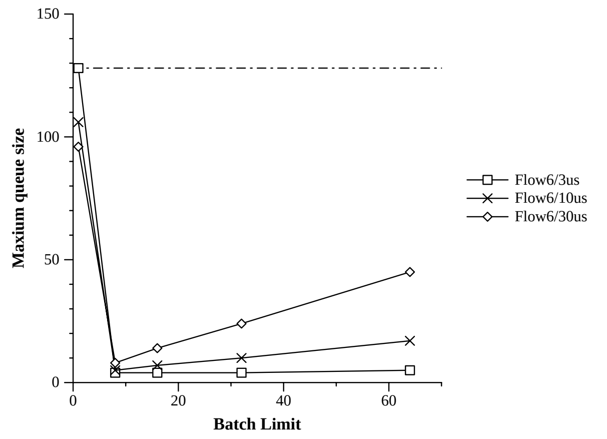

Figure 12: Maximum queue length seen by Flow 6

Flow 13 are approximately the same, but the scheduling overhead associated with Flow 13 is much higher for the less expensive packets. As Flow 13's packets becomes more computationally expensive, the system delivers the same cycle rate to it with less context switches, meaning more cycles available for the other 12 flows. The scenario suggests it is important to keep track of the scheduling overhead because it can substantially inflate actual per-packet-cost, thereby over-subscribing the system. Fortunately, with the batching throttle, such functionality can be easily integrated into the admission control mechanism.

As we increase the batch limit, the maximum queue length (and therefore the latency) each flow experiences goes up. However, the latency is also sensitive to the load imposed by Flow 13, which is illustrated by the gap between the 3μs, 10μs, and 30μs curves in Figure 12. This is a drawback of simple batching: it does not provide an upper bound on the scheduler's granularity. As a consequence, a change in the input workload can have a great impact on both E and latency each flow experiences, which makes it hard to reason about the system's behavior. Suppose Flow 6 only has 32 buffers, it will drop packets when Flow 13 per-packet-cost increases to $30\mu s$. This confirms the problem we saw earlier in Figure 6.

4.3.2 Enforcing Timeslice

Recognizing the problem of setting a meaningful batch limit, we rerun the experiments with a timeslice enforced on every flow, as we did in Figure 8. The timeslices we chose are $30\mu s$, $150\mu s$, $300\mu s$, $500\mu s$ and $1000\mu s$, respectively. The results are shown in Figures 13, 14 and 15.

Comparing these graphs with the ones presented in the previous section, the most notable change is that the curves in Figures 13 and 15 stay much closer as the load imposed by Flow 13 varies. This means both E and the latency a flow experiences are now dominated by the timeslice, rather than the load from other flows. This is a good property as the system behavior becomes more predictable. Figure 13 exhibits the tradeoff between efficiency and fine-grained control: in order to reach a good E we have to use a very large timeslice. For example, to make the system 84% efficient when Flow 13's per-packet-cost is $30\mu s$, we have to use a timeslice of $500\mu s$, whereas in Figure 10 setting batch limit to 8 will accomplish this goal. In the latter case, the maximum timeslice is only $8 \times 30 = 240\mu s$, and the maximum queue length seen by Flows 6 is 8. In contrast, enforcing a $500\mu s$ timeslice nearly triples the latency. Even with a large

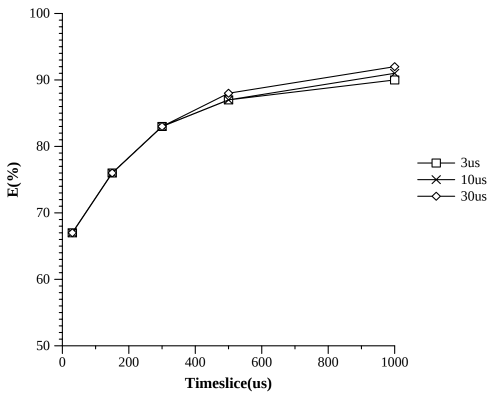

Figure 13: Efficiency Index for timeslice.

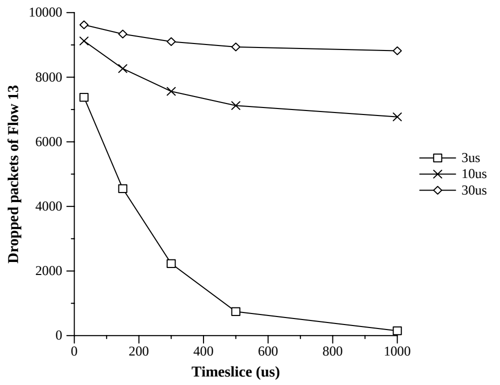

Figure 14: Flow 13's packets dropped when using timeslice.

timeslice, only those flows with high packet arrival rate enjoy the benefit. In this set of experiments only Flow 13 is able to use the full timeslice. Flow 6 never uses more than 16% of its timeslice.

There is one problem common to both the batching limit and the timeslice: the system does not have *active* control on the level of batching. Both mechanisms respond to workload passively in that they provide only an *upper bound*. The actual level of batching achieved depends mainly on the flow's packet rate and current load. Another drawback associated with these two mechanisms is that QoS and BE flows are treated equally. Ideally, we want to process BE packets in a large batch so to achieve higher efficiency, and at the same time schedule QoS flows at a finer granularity. This is enabled by the batching throttle.

4.3.3 Batching Throttle

We test the batching throttle with five (T, G) tuples: $(30\%, 10\mu s)$, $(20\%, 15\mu s)$, $(10\%, 30\mu s)$, $(5\%, 60\mu s)$ and $(2\%, 150\mu s)$. For each T, G is the minimum value satisfying Formula 1. Figures 16, 17 and 18 summarize the results. In Figure 18 we also plot Flow 3 and 9. Compared to Flow 6, Flow 3 has the same packet rate but less expensive packets; Flow 9 has the same specification, except it is allowed to timeout. These figures clearly demonstrate the three advantages of the batching throttle:

1. It achieves high efficiency by actively delaying processing of inefficient flows. It can produce better throughput even with a small granularity. With the same timeslice, the E in Figure 16 is much higher than those in Figure 13. In figure 17, when we use $(10\%, 30\mu s)$ or

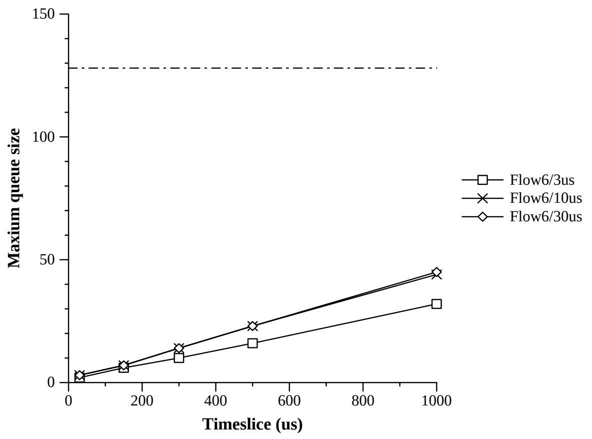

Figure 15: Maximum queue length seen by Flow 6 with timeslice.

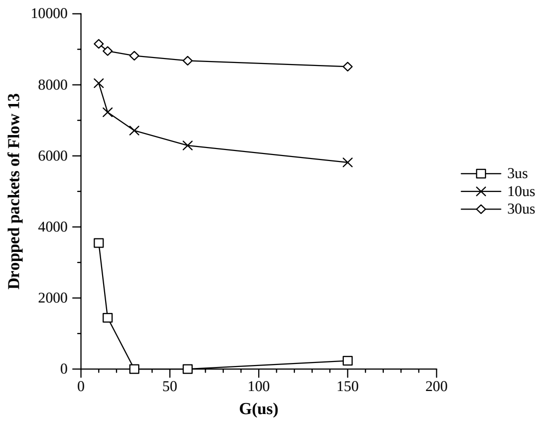

Figure 17: Flow 13's packets dropped when using batching throttle.

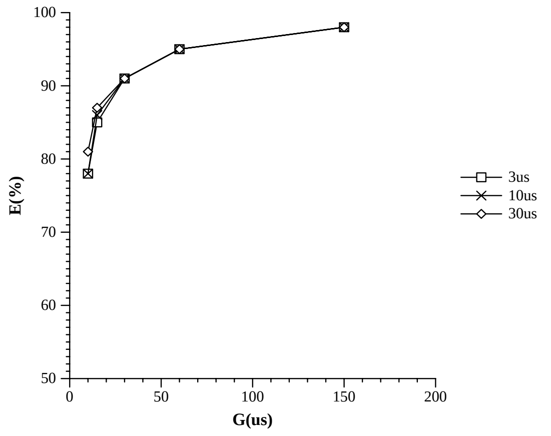

Figure 16: Efficiency Index for batching throttle.

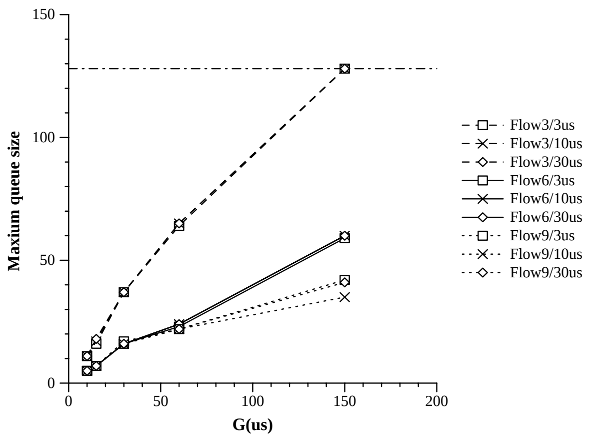

Figure 18: Maximum queue length seen by Flow 3, 6, 9 when using the batching throttle.

$(5\%, 60\mu s)$ as the parameters, the system successfully forwards all of Flow 13's packets with a $3\mu s$ per-packet-cost. There are no packet dropped from other flows, either. Note when T is further lowered to 2%, we again start to see drops even though the E is higher. The reason is such a low T makes the system non-work-conserving. The calculation of E does not include idle time.

2. It explicitly controls the scheduler's granularity, therefore it can provide stable delay bound and throughput. In Figure 18, the curves corresponding to one flow nearly overlap (the same in Figure 16), meaning that change in workload has little effect on the measured efficiency or the delay experienced by flows.

3. It considers the requirements of latency-sensitive QoS flows. In Figure 18, when $T = 2\%, G = 150\mu s$, both Flow 3 and Flow 6 suffer long delay. Flow 3 even experiences packet loss. In contrast, Flow 9 (with a $500\mu s$ timeout) enjoys better latency because it is scheduled more frequently.

In summary, the batching throttle exposes more "control knobs" to the system administrator. How to configure these parameters depends on local policy. The throttle mechanism also provides valuable information that can be used by the admission control module to adjust the system's behavior and/or to counter the side effects of the throttle itself. Taking Flow 3 as an example, when $T = 2\%$, the number of packets required to make it efficient is 150, which exceeds the number of its buffers, so it can be foreseen that Flow 3 will drop packets. As a compensation, the system could either allocate more buffers to the flow, thereby trading memory for CPU efficiency, or allow the flow to timeout. We reran the experiments with these two modifications. In the first case, Flow 3's buffer is doubled to 256, and all of its packets got through. In the second case, we gave Flow 3 a delay threshold of $500\mu s$. As a result, packet drops are also eliminated without any noticeable effects on other aspects of the system.

5. RELATED WORK

There have recently been several efforts to define extensible architectures for network routers [1, 4, 12, 15, 16, 18], although none have directly addressed the issue how the router should schedule these computations. Of these, our approach to scheduling would apply most naturally to router plugins [4] and active flows [16], both of which segregate work early. With router plugins, for example, it would be very straightforward to run all the plugins that implement a particular flow in a single forwarding process. In contrast, it is not clear how Click [12] modules would be efficiently broken into processes since Click has no notion of a per-flow switching path through the router. Support for flow isolation seems to be the critical requirement for our approach.

As mentioned in the introduction, the only work that has addressed the issue of scheduling a router's CPU cycles is a study of livelock conducted by Mogul and Ramakrishnan [11]; Druschel and Banga make similar observations about network servers [5] and Smith and Traw [17] discuss techniques for reducing the overhead of receiving interrupts. Our work goes beyond the issue of livelock by also considering the implications of meeting QoS obligations. Our use of a proportional share scheduler goes directly to this point. Both our work and Mogul-Ramakrishnan cite the importance of keeping the input-forwarding-output pipeline balanced, but we offer a general approach that combines proportional share and the batching throttle.

There has been considerable work on packet scheduling [3, 7], and some of the algorithms developed for this purpose have also been applied to CPU scheduling [2, 6]. However, none of these efforts demonstrate how a programmable router might exploit these algorithms to balance concerns about guarantees versus efficiency when one has to worry about scheduling cycles and bandwidth simultaneously. We leverage this algorithmic work, and in fact, we use an implementation of WF^{2}Q+ [3] in our prototype.

6. CONCLUSIONS

This paper explores the design space for scheduling the CPU on a software-based router. The router has three overriding goals: (1) maximize the throughput of best effort packets while providing different levels of service to QoS packets; (2) exhibit robust behavior in the presence of varying workloads, including packet flooding denial-of-service attacks; and (3) support switching paths of varying computational costs. The strategy we propose first divides the forwarding path into a processing pipeline (thereby exposing the critical scheduling decisions), and then applies a combination of two mechanisms: a proportional share scheduler and the batching throttle. Experiments with a prototype implementation verify the effectiveness of the resulting framework.

Although we have established a sound starting point, much work remains to be done. For example, we need to either verify our assertion that the cycle rate required by QoS flows can be derived empirically from a specified bit rate, or else develop a signalling protocol by which an application reserves a particular cycle rate. We also need to experiment with the router under a wider range of workloads, particularly those involving data-dependent costs. We plan to integrate the scheduling framework on a router architecture that includes both PCs and programmable line cards.

7. REFERENCES

[1] D. S. Alexander, M. Shaw, S. M. Nettles, and J. M. Smith. Active Bridging. In *Proceedings of the ACM SIGCOMM '97 Conference*, pages 101–111, September 1997.

[2] A. Bavier, L. Peterson, and D. Mosberger. BERT: A Scheduler for Best-Effort and Realtime Paths. Technical Report TR–602–99, Department of Computer Science, Princeton University, March 1999.

[3] J. C. R. Bennett and H. Zhang. Hierarchical Packet Fair Queueing Algorithms. In *Proceedings of the ACM SIGCOMM '96 Conference*, pages 143–156, August 1996.

[4] D. Decasper, Z. Dittia, G. Parulkar, and B. Plattner. Router Plugins: A Software Architecture for Next Generation Routers. In *Proceedings of the ACM SIGCOMM '98 Conference*, pages 229–240, September 1998.

[5] P. Druschel and G. Banga. Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems. In *Proceedings of the Second USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 261–275, October 1996.

[6] P. Goyal, X. Guo, and H. Vin. A Hierarchial CPU Scheduler for Multimedia Operating Systems. In *Proceedings of the Second USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 107–122, October 1996.

[7] P. Goyal, H. M. Vin, and H. Cheng. Start-time Fair Queueing: A Scheduling Algorithm for Integrated Services Packet Switching Networks. In *Proceedings of the ACM SIGCOMM '96 Conference*, pages 157–168, August 1996.

[8] S. Karlin and L. Peterson. VERA: An Extensible Router Architecture. In *Proceedings of the 4th International Conference on Open Architectures and Network Programming (OPENARCH)*, April 2001.

[9] N. McKeown. A Fast Switched Backplane for a Gigabit Switched Router. *Business Communications Review*, 27(12), December 1997.

[10] D. L. Mills. The Fuzzball. In *Proceedings of the SIGCOMM '88 Symposium*, pages 115–122, August 1988.

[11] J. C. Mogul and K. K. Ramakrishnan. Eliminating Receive Livelock in an Interrupt-Driven Kernel. *ACM Transactions on Computer Systems*, 15(3):217–252, August 1997.

[12] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 217–231, December 1999.

[13] D. Mosberger and L. L. Peterson. Making Paths Explicit in the Scout Operating System. In *Proceedings of the Second USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 153–167, October 1996.

[14] C. Partridge et al. A 50-Gb/s IP Router. *IEEE/ACM Transactions on Networking*, 6(3):237–247, June 1998.

[15] L. L. Peterson, S. C. Karlin, and K. Li. OS Support for General-Purpose Routers. In *Proceedings of the 7th Workshop on Hot Topics in Operating Systems (HotOS–VII)*, March 1999.

[16] J. M. Smith, K. L. Calvert, S. L. Murphy, H. K. Orman, and L. L. Peterson. Activating Networks: A Progress Report. *IEEE Computer*, 32(4):32–41, April 1999.

[17] J. M. Smith and C. B. S. Traw. Giving Applications Access to Gb/s Networking. *IEEE Network*, 7(4):44–52, July 1993.

[18] D. Wetherall. Active network vision and reality: lessons from a capsule-based system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 64–79, December 1999.