Scheduling Computations on a Programmable Router

Andy Bavier, Scott C. Karlin, Larry Peterson and Xiaohu Qie
Department of Computer Science
Princeton University

Abstract

It is becoming increasingly common for routers to compute on packets in addition to forwarding them, thereby exposing the problem of how the router's CPU cycles are scheduled. To complicate matters, this scheduling decision may be constrained by the desire to differentiate the level of service given different packet flows. This paper addresses the issue of scheduling computations on a programmable router. Its main contributions are to define a process architecture that allows computations to be scheduled in a meaningful way, and to identify the key issues that make this architecture difficult to implement in practice.

1 Introduction

Routers have always formed the backbone of the Internet, but they have recently been asked to do more than envisioned in their original design [4] and early implementations [13]. For example, routers are being asked to differentiate the level of service given to various classes of packets [5, 6], and they are being engineered to achieve high performance through the use of specialized hardware [12, 17]. A third trend is to extend the set of functions that routers support beyond the traditional forwarding service. We see this 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.
- In the limit, the active network research community is designing an architecture that will allow routers to run arbitrary code, thereby enabling the deployment of application-specific virtual networks.

This paper addresses a problem associated with the trend to extend router functionality: how to schedule computations on the router. Like any computing system, a router must schedule its cycles in a meaningful way. For example, it must decide when to apply its cycles to forwarding a vanilla IP packet, as opposed to using its cycles to re-write addresses, run a proxy, process a control message, or execute a router extension. To complicate matters, we allow for the possibility that the router promises to forward certain packets at some sustained bit rate, which clearly makes the issue of scheduling a router's CPU cycles more difficult.

Note that there are other issues with respect to programmable routers that this paper does not address. Three are potential red herrings, and so are worth mentioning. First, while there is significant debate about who should be allowed to install a new function in a router—ranging from the router companies to third party software vendors to end users—the important point is that routers are being asked to compute on packets (i.e., run code) in addition to switching them. These computations must be scheduled. Second, while a recent trend in router design is to improve performance through the use of special-purpose

dedicated logic (e.g., ASIC technology), the fact remains that any functionality not supported by the dedicated logic must be programmed on a reasonably general-purpose processor, and this processor must be scheduled. Third, we are using the term "router" in a generic way to denote any system whose primary responsibility to is forward packets from one network device to another. This does not imply that the fast path through high-end routers at the core of the Internet should be programmable.

Even though active/programmable/extensible routers [1, 7, 15, 18, 20, 23] have received considerable attention recently, it is certainly not fair to say that conventional routers ignore the possibility that there are multiple paths through the router, some involving more computation than others. A conventional commercial router may have as many as half a dozen switching paths. Some Cisco routers, for example, routinely use three paths: Cisco Express Forwarding (the true fast path), Fast Switching (the not quite so fast path) and Process Switching (the slow path). Distributed router architectures increase the number of paths, since some packets may be processed centrally and some on line cards. What is unique to the emerging situation is the diversity of the computations being performed. On most commercial routers, each path is carefully hand-tuned to forward a packet within some predefined time frame, and moving a function from one path to another is a non-trivial endeavor, typically requiring several months of testing and performance tuning. In other words, conventional routers are vertically integrated, realtime, embedded systems, where changes in the software are extremely expensive. Such an environment is stressed by the avalanche of new services people are proposing to add to their routers, especially when those services are added on the fly.

At the other end of the spectrum from embedded systems, one could view a programmable router as no different than a general-purpose computing system. For example, a workstation running Unix is not an uncommon router implementation. The shortcoming of this approach is that a general-purpose OS is not optimized to forward packets. One example that has already received attention is the fact that an I/O-bound workstation can suffer from livelock—spending all of its time servicing interrupts at the expense of running any other functions [14]. In general, such systems do poorly with tasks that must complete within tight cycle budgets or at some prescribed rate.

This paper addresses the issue of scheduling computations on a programmable router, with the goal of easing the constraints under which new software is added to a router without sacrificing the performance of a vertically integrated system. It makes two contributions. The first, modest contribution is to define a process architecture for programmable routers. Section 2 presents this architecture. The second, more important contribution is to identify the subtle implementation issues that arise when this process architecture is stressed under various workloads. Section 3 identifies several such issues and Section 4 reports experiments that measure their impact.

2 Process Architecture

This section describes the router's process architecture, including an introduction to the scheduling strategy we adopt. Although the architecture itself is fairly straightforward, the discussion does serve to expose the tradeoffs that become an issue in the next two sections. For the purpose of this section, it is sufficient to think of the router as being implemented on a PC with commodity interface cards; we return to the issue of what impact the hardware configuration has on the architecture (and vice versa) at the end of the section.

2.1 Processes

The simplest possible architecture 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 whatever processing the packet requires
> WRITE: write the 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.

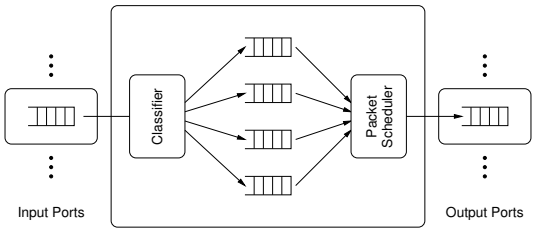

Figure 1: Supporting Differentiated Service

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.

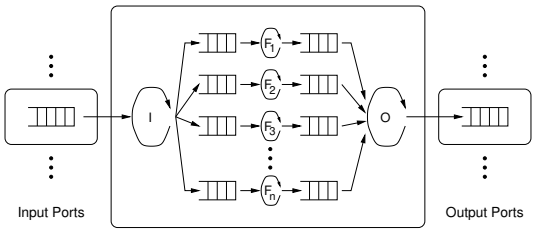

Figure 2: Supporting Differentiated Service and Variable Processing

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 the packet on the 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 the 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—vanilla 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 [23] 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 μsec, 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 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

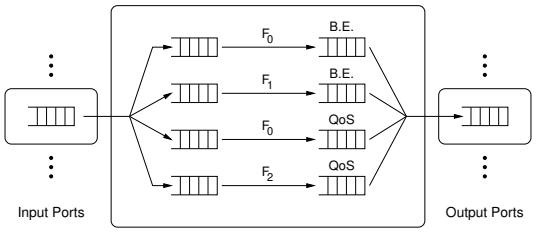

Figure 3: Example Switching Paths

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 vanilla 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/vanilla switching path, a QoS/vanilla path, a best-effort/non-standard path, and a QoS/non-standard path. Note that the two best effort paths aggregates many end-to-end flows, while the QoS paths carry a single end-to-end flow.

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 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 vanilla forwarding function, cycle budget permitting, although our scheduling framework argues for minimizing the work done in the input process.

2.2 Scheduling Discipline

The process model just described exposes all policy decisions to to the CPU scheduler. The task of the scheduler is to choose processes for execution in a manner 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.

We use a proportional share (PS) scheduler to meet these 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 lottery scheduling [22] and Weighted Fair Queueing [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—for example, 1 Mbps network rate times 10 cycles per bit equals 10 Mcps share for the forwarding process. We believe that most standard forwarding functions will have regular costs that can be determined through off-line experimentation, as illustrated in Table 1. More complex functions, and in particular, those with data-dependent costs are problematic; more on this in Section 3.

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 livelock 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 architecture can provide good best effort performance by assigning shares to different pipeline stages based on the ideal balance of the system in high load. That is, if on average it takes 1.5μs to read a packet, 6μs to forward it, and 1.5μs to write it, then our system balance is 16.7% for input, 66.7% for forwarding, and 16.7% for output. Even in overloaded conditions, this share assignment should provide a steady flow of packets through the pipeline without suffering from livelock. Again, the devil is in the details, as reported in Section 3.

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*. Section 3 discusses the interaction between shares and eligibility under various workloads.

2.3 Hardware Issues

The discussion up to this point has focused on a PC-based router. However, we believe the process model outlined in this section is applicable to a much wider set of hardware architectures. We consider three possible configurations.

First, many high-speed routers interconnect the ports with a high-speed switching fabric, with the PC functioning as a control processor rather than being directly in the forwarding path. They also employ special-purpose hardware and offload the input and output processes to that hardware. Like the input process discussed above, however, these front-end devices have a limited number of cycles that they can apply to any given packet. While it is likely that classification can be downloaded to such a device, as well as the IP fast path, most other computations must be performed on the router's control processor. A second factor that limits what functions can be offloaded is the programmability of these devices, which are often implemented with custom ASICs. Even if a function fits within the per-packet cycle budget, it may not be possible to program the front-end device to perform it. In any case, the situation on the control processor is exactly the same as described above, the only difference being that the processor may be able to reduce the classification time by taking advantage of work al-

ready done on the front-end device.

Second, a PC-based router that has programmable line cards (e.g., those based on an Intel IXP1200 chip[11] or a RAMiX PMC694 module[19]) behaves in much the same way: the programmable line card performs classification and the subset of functions F that fit within its cycle budget, and the PC handles all other functions. Unlike the ASIC-based configuration, however, the interface card (which has to deal with a collection of MAC chips) faces much the same scheduling problem as the PC. As a consequence, our process model applies to both programmable line cards and the PC that controls them. A related issue that arises in this setting in how processes on the line cards interact with the processes on the main processor. One possibility is that the input and output processes described in this section run on the line cards, and only the forwarding functions run on the main processor. This requires the intelligent line card to directly insert packets into the first set of queues on the main processor. Alternatively, a trivial input process still needs to read packets from the input device, classify them (perhaps based on a simple index already produced by the line card), and insert them into the appropriate queue for processing.

Third, an alternative to loading up the router proper with additional functionality (e.g., proxies, web caches, application-specific services) is to couple a conventional router with a server. In other words, the router forwards most packets by itself, but it diverts packets that require special treatment to a directly connected server. Again, this is just a variation on the previous two configurations, where the process model described above runs on the server. This raises an interesting point, however. The issue we are addressing is broader than just IP routers—our process model applies to any network device that perform I/O-centric tasks, that is, reads a packet from an input device, perform some transformation on the packet, and writes the result to an output device. For example, a web server/cache/proxy receives packets on an input port, parses the HTTP request and constructs a reply, and sends a packet on an output port (usually the same as the input port).

One can argue that in the limit the distinction between what's the router and what's the server will become blurred, and that a router will simply consist of multiple computing elements and multiple switching elements [18]. The process model described in this section applies to those computing elements that can be programmed to support new functionality.

3 Implementation Issues

We have implemented the architecture described in the previous section in the Scout operating system [16]. Scout is designed around a communication-oriented abstraction called a path. Our router uses Scout paths to implement switching paths. This section discusses our experiences implementing and tuning the architecture.

3.1 Implementation Details

Scout supports early demultiplexing, so separating the input process from the forwarding process was straightforward. The only complication was that Scout normally reads and classifies packets at interrupt time. We modified Scout so this work is done in a polling thread. On the output side, we added an output process since Scout paths normally write directly to the NIC's transmit queue.

Each Scout path implements a single forwarding function. Normally a thread pool is associated with each path. In this case, the pool was limited to a single thread, which corresponds to the forwarding process in our architecture. The function implemented by each path is constructed from a sequence of modules. For example, the module-sequence ETH/IP/ETH implements a general IP forwarding path, one that is able to handle IP options, dissimilar network devices, and so on. We also implemented a second, single-module path that is optimized for a specific source/sink device pair. It changes the addresses in the frame header, decrements IP's TTL field, and incrementally recomputes the header checksum. These two paths (forwarding functions) correspond to the second and first rows, respectively, in Table 1. We primarily use the fast path in the measurements reported in the next section.

One of the optimizations exploited by the fast path is to bypass the Scout message library. Scout messages facilitate standard packet processing such as stripping and adding packet headers. Since the fast path assumes a fixed-length IP header and the same frame header on both input and output, the general-purpose library is unnecessary.

The implementation uses the WF^{2}Q+ [3] fair queuing scheduler. There are actually two instances of this scheduler. The first selects a process to execute based on the cycle reservation made for each. The second scheduler runs only when an output process executes, and selects a packet to transmit on a link. It maintains a queue of Scout paths, ordered by the WF^{2}Q+ as-

signed virtual timestamp. The path with the smallest timestamp is selected, and a packet from that path's output queue is selected for transmission.

Because Scout threads are non-preemptive, and since they perform a simple action in our architecture (i.e., move packets from one queue to another), we can use thread continuations whenever a thread yields or suspends. A thread that yields maintains no state; it always resumes execution with the next packet in the process's input queue. This allows us to avoid saving registers between runs of the same thread.

We use a Tulip driver from MIT that was tuned for their Click router [15]. This driver places the card in auto transmit mode; in this mode, the transmit process on the device polls the DMA queue, saving I/O commands. The driver also performs specialized buffer management and prefetching.

3.2 Policy Issues

There are several policy issues regarding how a programmable router allocates its resources, including what fraction of the router's CPU cycles and link bandwidth it is willing to set aside for QoS flows. We highlight two additional issues.

The first issue is what share to give the input processes, where the requirements of best effort and QoS flows are in conflict. For the sake of best effort flows, we want to assign input process shares based on the ideal cycle distribution in overload. We do this to avoid livelock. 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 the input process must run at line speeds; otherwise, packets belonging to well-behaved QoS flows may be dropped on the line card, thereby violating the promises the router has made to the flow. 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. QoS flows want conservative input shares to resist denial of service attacks; best effort flows do not, since it means the system will waste cycles reading in packets that will be dropped later in the pipeline. Our choice is to favor QoS flows by giving the input process a conservative share. We describe how to temper this decision in Section 3.3.

The second policy issue is the extent to which best effort traffic should be aggregated versus isolated. As described in Section 2, 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.

3.3 Queue Estimator

The process architecture is designed to ensure that all messages are buffered in explicit message queues, as opposed to hidden thread queues. We expose these message queues to allow for future scheduling algorithms that may take queue lengths into account. Our current framework uses them 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. However, calculating the eligibility of an input process is not straightforward since its input queue resides on the device. 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 especially troublesome if we give the input process a conservative share so as to ensure that it can keep up with packets arriving at line speeds.

One option is 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. Instead, our prototype *estimates* the device queue length based on previous observations. It does this by keeping a simple weighted average of the packets read during each execution of the polling thread. This estimate is currently used as feedback for a batching mechanism that will be discussed in the next section. 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.

3.4 Batching

Batching allows the system to spread the cost of a context switch across multiple packets, resulting in higher forwarding rates. Our system tries to batch packets at all three stages of the process pipeline. The forwarding process is given a timeslice and allowed to send as many packets as possible during that slice.[1] The input and output processes use a simple mechanism which we call a *throttle* to achieve good batching behavior. Below we explain how batching works in our router.

Bigger batches of packets become available for processing if we reduce the period at which the packet consumer runs. For example, assume that packets arrive on a particular interface at a rate of 10 Kpps. If the input process on that interface runs every 100 μs, it can read only one packet; if it runs every millisecond, it can read ten. Since reading ten packets at a time reduces the cost per packet of the context switch, the input process can read packets at the same rate using less share, leading to a more efficient system.

Proportional share algorithms already decouple share and period. When a thread runs, its virtual time is advanced by an amount proportional to its execution time. The more its virtual time advances, the longer the amount of time before it will be allowed to run again. One option would be to vary the period of the input thread using virtual time. We could advance its virtual time by a minimum amount (i.e., the amount of time it would take to read a target-sized batch of packets) to stretch its period so that it would find multiple packets on the device when it runs.

We take another approach. Since we give an input process a conservative share, we want both to adjust its period for better batching, and to give back cycles it does not need to the system. We wrap both of these functions into the throttle mechanism. The throttle puts an input or output process to sleep after it runs, varying the sleep time using feedback from the estimator in Section 3.3. Each process has a batch target range, $[x, y]$, representing the number of packets it wants to batch when it runs. When the estimate falls below the minimum threshold x, we increase the sleep interval of the input process; when it goes above the maximum threshold y, we decrease the sleep time. We cap the maximum sleep time of an input process so that it will always poll the device at an acceptable rate. We can imagine needing to tweak this algorithm as we understand more about actual workloads (e.g., rapidly increasing the polling rate in response to a single large input queue), but this simple strategy has worked well in the experiments we have run.

This throttle mechanism accomplishes two things. First, it stretches out the period at which the process runs for better batching. Second, it dynamically adjusts the actual execution rate of the process in response to the workload. The throttle slows down the input process to a rate commensurate with the packet arrival rate, and its unused share is distributed fairly throughout the system. So the throttle helps performance through amortizing context switch overhead for both input and output processes, as well as preventing an input process from polling unnecessarily.

Our system batches at all stages of the process pipeline. Though batching leads to better performance, we must be careful because it can interact with process eligibility in undesirable ways. With large batch sizes, processes may be able to drain their input queues when they run, thus becoming ineligible. The result can be that, at any one time, only a few processes are actually eligible to be scheduled; process eligibility, rather than share, may then dominate the system behavior. We are not aware of a general solution to this problem, other than experimentally trying different batch sizes to see which works best.

3.5 Cycle and Link Rates

Our discussion describes 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.

An interesting question is whether we can do better than make a conservative reservation for data-dependent flows. For example, can an application just reserve the average cycle rate its packets require, even

[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.

when the cycle requirements are highly variable from packet to packet. 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. In the end, 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 a switching path's output queue is large enough, it can buffer packet produced during good times (when processing costs are small), and therefore not go empty when processing cost are large.

4 Experiments

This section reports on a series of experiments designed to evaluate the effectiveness of our process architecture. The experiments were run on the configuration shown in Figure 4. It consists of our router, a $100\,\mathrm{Mbps}$ switch, and three PCs. The machine running our prototype router has a $450\,\mathrm{MHz}$ Pentium-II processor with a $512\,\mathrm{KB}$ L2 cache, and three Tulip (21143 chip) $100\,\mathrm{Mbps}$ network interface cards. We use the three PCs, labelled A, B, and C, as packet sources.

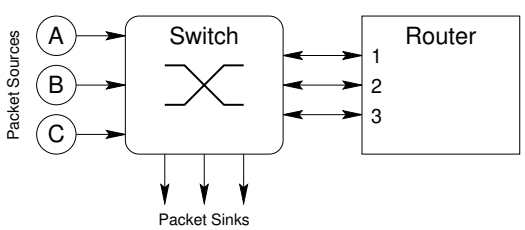

Figure 4: Experimental Setup

The reason for the switch, as opposed to directly connecting the sources to the router, is that we divert packets destined for the three PCs to a non-existent sink. Otherwise, we found that packet reception interfered with packet transmission on the PCs (which used 3COM cards), making our sources unpredictable.

During the tests, each source PC generates a stream of 64-byte IP packets at rates up to $140\,\mathrm{Kpps}$. Using all three sources, we can generate an aggregate maximum offered load of $420\,\mathrm{Kpps}$. 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.

4.1 Evaluating Overhead

We begin by evaluating the overhead introduced by our process architecture, to determine whether or not it is prohibitively expensive in practice. We measure the relative overhead by plotting the offered load to the router versus the resulting 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. We assign an equal share of the processor to each process.

Batching To reduce the overhead of context switches, we can enable batching. With batching enabled, each 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.

Figure 5 summarizes the results of five experiments. In each experiment, the source PCs in Figure 4 gradually increase their aggregate offered load from 0 to $420\,\mathrm{Kpps}$, as 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 $48\,\mathrm{Kpps}$, after which it begins to suffer from receive livelock. In contrast, the polling implementations give a more desirable behavior: the forwarding rate increases up to a certain point and them 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 [14].

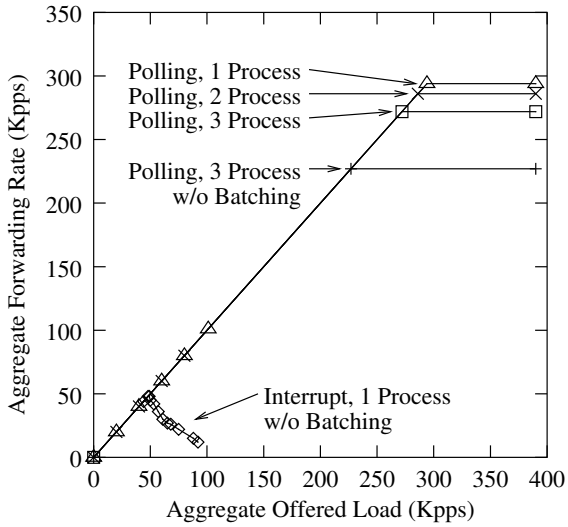

Figure 5: Impact of interrupt handling and context switching on forwarding rate.

Processes	Maximum Batch Size	Max. Forwarding Rate	
		Kpps	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 4.1 shows the relative maximum forwarding rates using polling. The forwarding rates are from the plateaus in Figure 5. 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% of 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.

Finally, micro-experiments indicate that in the three process case, the input process spends $1.6\,\mu$s on each packet, the forwarding process spends $0.3\,\mu$s on each packet, and the output process spends $1.4\,\mu$s on each packet. Note that if we assume two process stages—an input process and a combined forwarding/output process—a balanced system has almost exactly a 1:1 ratio. We use the two-process configuration in the next subsection to focus on best effort flows, since the output process is only required when we need to implement link scheduling.

Note that although absolute performance is not the focus of this paper, the total 3.3μs forwarding time for each packet did not come easily. We started with a per-packet cost of over 11.7μs. The specialized IP forwarding path shaved 2.7μs (this is the difference between the first two rows in Table 1), the driver optimizations shaved 2.8μs, and bypassing Scout's general-purpose message library saved 2.9μs. Perhaps most important of all, however, is that fact that our architecture allowed us to implement context switches as very inexpensive continuations. Having two full-blown context switches on the forwarding path has the potential to add 10μs or more to the forwarding time.

4.2 Best Effort Forwarding

We now focus on the performance of our prototype when forwarding only best effort packets. We show that our architecture 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 as just described, we know that the input requires $1.6\,\mu$s 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 $8.0\,\mu$s 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 $8.0\,\mu$s and $16.0\,\mu$s, respectively, meaning that their ideal share ratios are 1:10 and 1:15. The fourth column of Table 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 $140\,\mathrm{Kpps}$. The maximum forwarding rate of 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

Table 3: Best effort throughput in Kpps

Flow	Fwd Cost	Ideal balance	Balanced share	1:10 share w/o throttle	1:10 w/ throttle
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

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 show the results of the imbalance, the difference between the two columns being that the fifth drops packets that demux to a full queue (as we might do if we had QoS flows), while the sixth uses a combination of eligibility and the batching throttle 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. 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 and packets are dropped off the tail of the forwarding process's input queue. Column six for flow C shows that eligibility and throttling 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. To see the full effect, we configured an experiment with 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 6μs, which fully utilizes the CPU. Without the queue estimator, roughly a quarter of the CPU is wasted polling idle input ports, thereby yielding a forwarding rate of 91Kpps for the active port. With the estimator enabled, the router was able to forward packets at 130Kpps, the maximum achievable rate for this configuration.

4.3 Mixing Best Effort and QoS

We now describe an experiment in which we mix two QoS flows with a best effort flow. We again us the three sources depicted in Figure 4, with the flow from source A arriving on port 1, the flow from B arriving on port 2, and the flow from C arriving on port 3. In this experiment flow B is routed to output port 1, while both flows A and C are routed to port 2. Thus, A and C compete for this output link. Also, flows B and C are QoS flows with a 90 Kpps reservation. They are using their entire reservation.

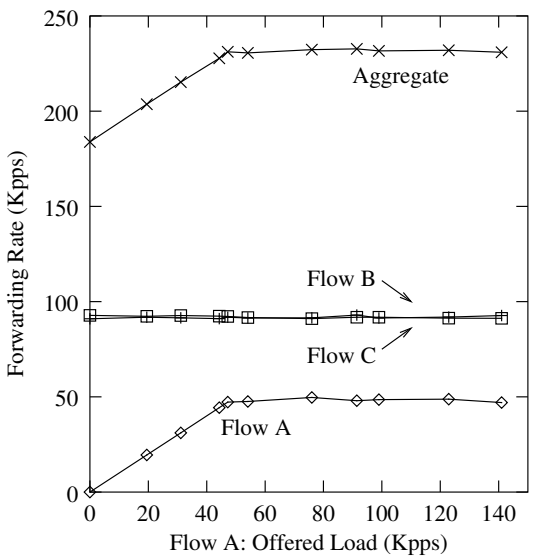

Figure 6: Mix of Best Effort (A) and QoS (B,C) Flows.

Figure 6 shows the forwarding rate for flows A, B, and C, as the offered load on flow A increases. We see that flows B and C meet their reservation despite the increased load. We also see that as long as flow A is under approximately 50 Kpps, none of its packets are dropped. After this point, no more packets in flow A can get through because we have saturated port 2 (140 Kpps total), the outgoing link shared by flows A and C.

4.4 CPU versus Link

Using same setup of hardware and flows as Section 4.3, we explore the situation where the router transitions from being link-bound to being CPU-bound. We do this by fixing the arrival rate of best effort flow A at $50\,\mathrm{Kpps}$, which saturates the link that A and C share. 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 5μs on the x-axis.

Figure 7 shows the forwarding rate for the flows when we enable simple batching. 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 output queues associated with QoS flows B and C empty, causing them to become idle.

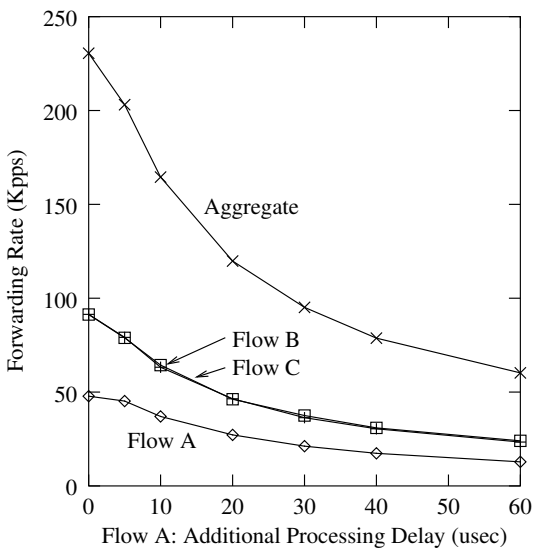

Figure 7: Detrimental Effects of Simple Batching as Processing Costs Increase.

In Figure 8, we turn off batching and do better: 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 PS scheduler regains control at a finer granularity. As we have seen, though, turning batching off comes with a cost.

In Figure 9, we turn batching back on but we only allow batching to process 16 packets or process for $30\,\mu$s, whichever comes first. We see that QoS flows B and C meet their reservation and that the aggregate forwarding rate is higher than that in Figure 8.

In summary, the router makes a smooth transition

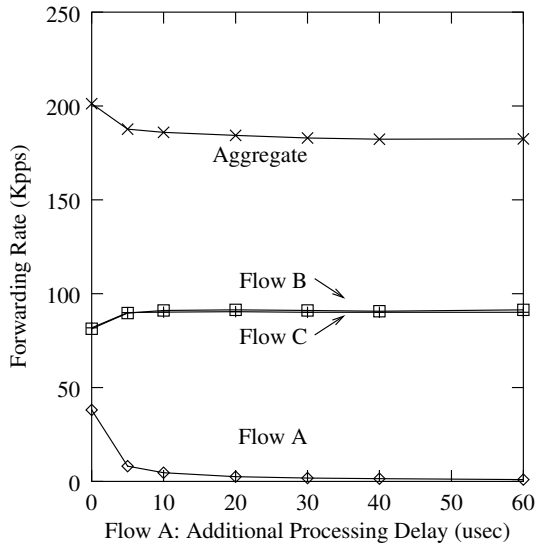

Figure 8: No Batching

from being link-bound to being CPU-bound, while preserving the rate requested by QoS flow, as long as the PS scheduler gets a frequent enough opportunity to re-schedule the processor. The batching throttle provides this opportunity without sacrificing all the performance advantages of batching.

4.5 Data-Dependent Processing

Finally, we experimented with the effect of data-dependent processing costs on the router's ability to meet QoS promises. Specifically, we ran a QoS flow at 50Kpps, first with a fixed processing cost of 8μs per packet, and then with a processing cost distributed uniformly between 4μs and 12μs per packet. In the second case we reserved the average rate of 8μs per packet. In both cases, the flow was able to maintain its 50Kpps rate, but other best effort traffic was affected. When the QoS flow used a fixed 8μs per packet, a full-speed best effort flow was able to achieve 127Kpps, while competing with the variable cost flow slowed the best effort flow down to 120Kpps. In effect, the expensive packets stole cycles from the best effort flow.

A second set of experiments evaluated the impact of the output queue size. With two flows competing for the same output link, one a QoS flow running at 50Kpps and the other a best effort flow running as fast as possible, the best effort flow is able to achieve 90Kpps when the QoS flow requires a fixed 8μs of computation for each packet. This is because 50+90Kpps=140Kpps, the maximum output rate of the link. When the QoS flow's cost is variable, again uniformly distributed between 4μs and 12μs per

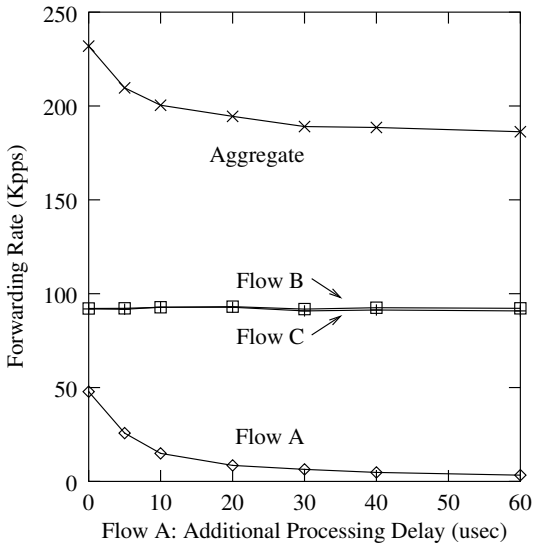

Figure 9: *Batching Throttle*. Batching with a 30 μs limit.

packet, both flows achieved the same rate. This is with an output queue size of 128 packets. If the queue size is set to 1 packet, then the best effort flow is able to achieve 110Kpps and the QoS flow dips to 30Kpps. This is because the best effort flow is able to send extra packets when the QoS flow's queue is empty due to a packet that requires extra processing.

Clearly we need to do additional experiments under differing circumstances, but ultimately the required queue size is a function of the behavior of the forwarding process. In general, we need to better understand what applications can tell us about their forwarding functions, and define a relationship to queue sizes based on this information.

5 Related Work

There have recently been several efforts to define extensible architectures for network routers [1, 7, 15, 18, 20, 23], 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 [7] and active flows [20], 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 [15] 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 [14]; Druschel and Banga make similar observations about network servers [8] and Smith and Traw [21] 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 eligibility.

There has been considerable work on packet scheduling [3, 10], and some of the algorithms developed for this purpose have also been applied to CPU scheduling [2, 9, 22]. 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^2Q+ [3] in our prototype.

6 Conclusions

This paper explores the design space for scheduling the CPU on a programmable router. The router has three overriding goals: (1) to 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 a 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 involve data-dependent costs. Finally, we plan to integrate the scheduling framework on a router architecture that includes both PCs and programmable line cards.

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. D. Clark. The Design Philosophy of the DARPA Internet Protocols. In *Proceedings of the SIGCOMM '88 Symposium*, pages 106–114, August 1988.

[5] D. D. Clark and W. Fang. Explicit Allocation of Best-Effort Packet Delivery Service. *IEEE/ACM Transactions on Networking*, 6(4):362–373, August 1998.

[6] D. D. Clark, S. Shenker, and L. Zhang. Supporting Real-Time Applications in an Integrated Services Packet Network: Architecture and Mechanism. In *Proceedings of the SIGCOMM '92 Conference*, pages 14–26, August 1992.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] Level One Communications, Inc., An Intel Company, Sacramento, California. *Level One™ IXP1200 Network Processor Advance Datasheet*, September 1999.

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

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

[14] 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.

[15] 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.

[16] 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.

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

[18] 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.

[19] RAMiX Incorporated, Ventura, California. *PMC/CompactPCI Ethernet Controllers Product Family Data Sheet*, 1999.

[20] 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.

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

[22] C. A. Waldspurger and W. E. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *Proceedings of the First USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 1–11, November 1994.

[23] 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.