The following paper was originally published in the
Proceedings of the USENIX 1996 Annual Technical Conference
San Diego, California, January 1996

Eliminating Receive Livelock in an Interrupt-driven Kernel

Jeffrey Mogul, DEC Western Research Laboratory
K. K. Ramakrishnan, AT&T Bell Laboratories

For more information about USENIX Association contact:

1. Phone: 510 528-8649
2. FAX: 510 548-5738
3. Email: office@usenix.org
4. WWW URL: http://www.usenix.org

Eliminating Receive Livelock in an Interrupt-driven Kernel

Jeffrey C. Mogul
Digital Equipment Corporation Western Research Laboratory
K. K. Ramakrishnan
AT&T Bell Laboratories

Abstract

Most operating systems use interface interrupts to schedule network tasks. Interrupt-driven systems can provide low overhead and good latency at low offered load, but degrade significantly at higher arrival rates unless care is taken to prevent several pathologies. These are various forms of ***receive livelock***, in which the system spends all its time processing interrupts, to the exclusion of other necessary tasks. Under extreme conditions, no packets are delivered to the user application or the output of the system.

To avoid livelock and related problems, an operating system must schedule network interrupt handling as carefully as it schedules process execution. We modified an interrupt-driven networking implementation to do so; this eliminates receive livelock without degrading other aspects of system performance. We present measurements demonstrating the success of our approach.

1. Introduction

Most operating systems use interrupts to internally schedule the performance of tasks related to I/O events, and particularly the invocation of network protocol software. Interrupts are useful because they allow the CPU to spend most of its time doing useful processing, yet respond quickly to events without constantly having to poll for event arrivals.

Polling is expensive, especially when I/O events are relatively rare, as is the case with disks, which seldom interrupt more than a few hundred times per second. Polling can also increase the latency of response to an event. Modern systems can respond to an interrupt in a few tens of microseconds; to achieve the same latency using polling, the system would have to poll tens of thousands of times per second, which would create excessive overhead. For a general-purpose system, an interrupt-driven design works best.

Most extant operating systems were designed to handle I/O devices that interrupt every few milliseconds. Disks tended to issue events on the order of once per revolution; first-generation LAN environments tend to generate a few hundred packets per second for any single end-system. Although people understood the need to reduce the cost of taking an interrupt, in general this cost was low enough that any normal system would spend only a fraction of its CPU time handling interrupts.

The world has changed. Operating systems typically use the same interrupt mechanisms to control both network processing and traditional I/O devices, yet many new applications can generate packets several orders of magnitude more often than a disk can generate seeks. Multimedia and other real-time applications will become widespread. Client-server applications, such as NFS, running on fast clients and servers can generate heavy RPC loads. Multicast and broadcast protocols subject innocent-bystander hosts to loads that do not interest them at all. As a result, network implementations must now deal with significantly higher event rates.

Many multi-media and client-server applications share another unpleasant property: unlike traditional network applications (Telnet, FTP, electronic mail), they are not flow-controlled. Some multi-media applications want constant-rate, low-latency service; RPC-based client-server applications often use datagram-style transports, instead of reliable, flow-controlled protocols. Note that whereas I/O devices such as disks generate interrupts only as a result of requests from the operating system, and so are inherently flow-controlled, network interfaces generate unsolicited receive interrupts.

The shift to higher event rates and non-flow-controlled protocols can subject a host to congestive collapse: once the event rate saturates the system, without a negative feedback loop to control the sources, there is no way to gracefully shed load. If the host runs at full throughput under these conditions, and gives fair service to all sources, this at least preserves the possibility of stability. But if throughput decreases as the offered load increases, the overall system becomes unstable.

Interrupt-driven systems tend to perform badly under overload. Tasks performed at interrupt level,

by definition, have absolute priority over all other tasks. If the event rate is high enough to cause the system to spend all of its time responding to interrupts, then nothing else will happen, and the system throughput will drop to zero. We call this condition *receive livelock*: the system is not deadlocked, but it makes no progress on any of its tasks.

Any purely interrupt-driven system using fixed interrupt priorities will suffer from receive livelock under input overload conditions. Once the input rate exceeds the reciprocal of the CPU cost of processing one input event, any task scheduled at a lower priority will not get a chance to run.

Yet we do not want to lightly discard the obvious benefits of an interrupt-driven design. Instead, we should integrate control of the network interrupt handling sub-system into the operating system's scheduling mechanisms and policies. In this paper, we present a number of simple modifications to the purely interrupt-driven model, and show that they guarantee throughput and improve latency under overload, while preserving the desirable qualities of an interrupt-driven system under light load.

2. Motivating applications

We were led to our investigations by a number of specific applications that can suffer from livelock. Such applications could be built on dedicated single-purpose systems, but are often built using a general-purpose system such as UNIX®, and we wanted to find a general solution to the livelock problem. The applications include:

- *Host-based routing*: Although inter-network routing is traditionally done using special-purpose (usually non-interrupt-driven) router systems, routing is often done using more conventional hosts. Virtually all Internet ''firewall'' products use UNIX or Windows NT™ systems for routing [7, 13]. Much experimentation with new routing algorithms is done on UNIX [2], especially for IP multicasting.
- *Passive network monitoring*: network managers, developers, and researchers commonly use UNIX systems, with their network interfaces in ''promiscuous mode,'' to monitor traffic on a LAN for debugging or statistics gathering [8].
- *Network file service*: servers for protocols such as NFS are commonly built from UNIX systems.

These applications (and others like them, such as Web servers) are all potentially exposed to heavy, non-flow-controlled loads. We have encountered livelock in all three of these applications, have solved or mitigated the problem, and have shipped the solutions to customers. The rest of this paper concentrates on host-based routing, since this simplifies the context of the problem and allows easy performance measurement.

3. Requirements for scheduling network tasks

Performance problems generally arise when a system is subjected to transient or long-term input overload. Ideally, the communication subsystem could handle the worst-case input load without saturating, but cost considerations often prevent us from building such powerful systems. Systems are usually sized to support a specified design-center load, and under overload the best we can ask for is controlled and graceful degradation.

When an end-system is involved in processing considerable network traffic, its performance depends critically on how its tasks are scheduled. The mechanisms and policies that schedule packet processing and other tasks should guarantee acceptable system *throughput*, reasonable *latency* and *jitter* (variance in delay), *fair* allocation of resources, and overall system *stability*, without imposing excessive overheads, especially when the system is overloaded.

We can define throughput as the rate at which the system delivers packets to their ultimate consumers. A consumer could be an application running on the receiving host, or the host could be acting as a router and forwarding packets to consumers on other hosts. We expect the throughput of a well-designed system to keep up with the offered load up to a point called the *Maximum Loss Free Receive Rate* (MLFRR), and at higher loads throughput should not drop below this rate.

Of course, useful throughput depends not just on successful reception of packets; the system must also transmit packets. Because packet reception and packet transmission often compete for the same resources, under input overload conditions the scheduling subsystem must ensure that packet transmission continues at an adequate rate.

Many applications, such as distributed systems and interactive multimedia, often depend more on low-latency, low-jitter communications than on high throughput. Even during overload, we want to avoid long queues, which increases latency, and bursty scheduling, which increases jitter.

When a host is overloaded with incoming network packets, it must also continue to process other tasks, so as to keep the system responsive to management and control requests, and to allow applications to make use of the arriving packets. The scheduling subsystem must fairly allocate CPU resources among packet reception, packet transmission, protocol

processing, other I/O processing, system housekeeping, and application processing.

A host that behaves badly when overloaded can also harm other systems on the network. Livelock in a router, for example, may cause the loss of control messages, or delay their processing. This can lead other routers to incorrectly infer link failure, causing incorrect routing information to propagate over the entire wide-area network. Worse, loss or delay of control messages can lead to network instability, by causing positive feedback in the generation of control traffic [10].

4. Interrupt-driven scheduling and its consequences

Scheduling policies and mechanisms significantly affect the throughput and latency of a system under overload. In an interrupt-driven operating system, the interrupt subsystem must be viewed as a component of the scheduling system, since it has a major role in determining what code runs when. We have observed that interrupt-driven systems have trouble meeting the requirements discussed in section 3.

In this section, we first describe the characteristics of an interrupt-driven system, and then identify three kinds of problems causes by network input overload in interrupt-driven systems:

- *Receive livelocks* under overload: delivered throughput drops to zero while the input overload persists.
- Increased *latency* for packet delivery or forwarding: the system delays the delivery of one packet while it processes the interrupts for subsequent packets, possibly of a burst.
- *Starvation* of packet transmission: even if the CPU keeps up with the input load, strict priority assignments may prevent it from transmitting any packets.

4.1. Description of an interrupt-driven system

An interrupt-driven system performs badly under network input overload because of the way in which it prioritizes the tasks executed as the result of network input. We begin by describing a typical operating system's structure for processing and prioritizing network tasks. We use the 4.2BSD [5] model for our example, but we have observed that other operating systems, such as VMS™, DOS, and Windows NT, and even several Ethernet chips, have similar characteristics and hence similar problems.

When a packet arrives, the network interface signals this event by interrupting the CPU. Device interrupts normally have a fixed Interrupt Priority Level (IPL), and preempt all tasks running at a lower IPL; interrupts do not preempt tasks running at the same IPL. The interrupt causes entry into the associated network device driver, which does some initial processing of the packet. In 4.2BSD, only buffer management and data-link layer processing happens at ''device IPL.'' The device driver then places the packet on a queue, and generates a software interrupt to cause further processing of the packet. The software interrupt is taken at a lower IPL, and so this protocol processing can be preempted by subsequent interrupts. (We avoid lengthy periods at high IPL, to reduce latency for handling certain other events.)

The queues between steps executed at different IPLs provide some insulation against packet losses due to transient overloads, but typically they have fixed length limits. When a packet should be queued but the queue is full, the system must drop the packet. The selection of proper queue limits, and thus the allocation of buffering among layers in the system, is critical to good performance, but beyond the scope of this paper.

Note that the operating system's scheduler does not participate in any of this activity, and in fact is entirely ignorant of it.

As a consequence of this structure, a heavy load of incoming packets could generate a high rate of interrupts at device IPL. Dispatching an interrupt is a costly operation, so to avoid this overhead, the network device driver attempts to *batch* interrupts. That is, if packets arrive in a burst, the interrupt handler attempts to process as many packets as possible before returning from the interrupt. This amortizes the cost of processing an interrupt over several packets.

Even with batching, a system overloaded with input packets will spend most of its time in the code that runs at device IPL. That is, the design gives absolute priority to processing incoming packets. At the time that 4.2BSD was developed, in the early 1980s, the rationale for this was that network adapters had little buffer memory, and so if the system failed to move a received packet promptly into main memory, a subsequent packet might be lost. (This is still a problem with low-cost interfaces.) Thus, systems derived from 4.2BSD do minimal processing at device IPL, and give this processing priority over all other network tasks.

Modern network adapters can receive many back-to-back packets without host intervention, either through the use of copious buffering or highly autonomous DMA engines. This insulates the system from the network, and eliminates much of the rationale for giving absolute priority to the first few steps of processing a received packet.

4.2. Receive livelock

In an interrupt-driven system, receiver interrupts take priority over all other activity. If packets arrive too fast, the system will spend all of its time processing receiver interrupts. It will therefore have no resources left to support delivery of the arriving packets to applications (or, in the case of a router, to forwarding and transmitting these packets). The useful throughput of the system will drop to zero.

Following [11], we refer to this condition as *receive livelock*: a state of the system where no useful progress is being made, because some necessary resource is entirely consumed with processing receiver interrupts. When the input load drops sufficiently, the system leaves this state, and is again able to make forward progress. This is not a deadlock state, from which the system would not recover even when the input rate drops to zero.

A system could behave in one of three ways as the input load increases. In an ideal system, the delivered throughput always matches the offered load. In a realizable system, the delivered throughput keeps up with the offered load up to the *Maximum Loss Free Receive Rate* (MLFRR), and then is relatively constant after that. At loads above the MLFRR, the system is still making progress, but it is dropping some of the offered input; typically, packets are dropped at a queue between processing steps that occur at different priorities.

In a system prone to receive livelock, however, throughput decreases with increasing offered load, for input rates above the MLFRR. Receive livelock occurs at the point where the throughput falls to zero. A livelocked system wastes all of the effort it puts into partially processing received packets, since they are all discarded.

Receiver-interrupt batching complicates the situation slightly. By improving system efficiency under heavy load, batching can increase the MLFRR. Batching can shift the livelock point but cannot, by itself, prevent livelock.

In section 6.2, we present measurements showing how livelock occurs in a practical situation. Additional measurements, and a more detailed discussion of the problem, are given in [11].

4.3. Receive latency under overload

Although interrupt-driven designs are normally thought of as a way to reduce latency, they can actually increase the latency of packet delivery. If a burst of packets arrives too rapidly, the system will do link-level processing of the entire burst before doing any higher-layer processing of the first packet, because link-level processing is done at a higher priority. As a result, the first packet of the burst is not delivered to the user until link-level processing has been completed for all the packets in the burst. The latency to deliver the first packet in a burst is increased almost by the time it takes to receive the entire burst. If the burst is made up of several independent NFS RPC requests, for example, this means that the server's disk sits idle when it could be doing useful work.

One of the authors has previously described experiments demonstrating this effect [12].

4.4. Starvation of transmits under overload

In most systems, the packet transmission process consists of selecting packets from an output queue, handing them to the interface, waiting until the interface has sent the packet, and then releasing the associated buffer.

Packet transmission is often done at a lower priority than packet reception. This policy is superficially sound, because it minimizes the probability of packet loss when a burst of arriving packets exceeds the available buffer space. Reasonable operation of higher level protocols and applications, however, requires that transmit processing makes sufficient progress.

When the system is overloaded for long periods, use of a fixed lower priority for transmission leads to reduced throughput, or even complete cessation of packet transmission. Packets may be awaiting transmission, but the transmitting interface is idle. We call this *transmit starvation*.

Transmit starvation may occur if the transmitter interrupts at a lower priority than the receiver; or if they interrupt at the same priority, but the receiver's events are processed first by the driver; or if transmission completions are detected by polling, and the polling is done at a lower priority than receiver event processing.

This effect has also been described previously [12].

5. Avoiding livelock through better scheduling

In this section, we discuss several techniques to avoid receive livelocks. The techniques we discuss in this section include mechanisms to control the rate of incoming interrupts, polling-based mechanisms to ensure fair allocation of resources, and techniques to avoid unnecessary preemption.

5.1. Limiting the interrupt arrival rate

We can avoid or defer receive livelock by limiting the rate at which interrupts are imposed on the system. The system checks to see if interrupt processing is taking more than its share of resources, and if so, disables interrupts temporarily.

The system may infer impending livelock because it is discarding packets due to queue overflow, or because high-layer protocol processing or user code are making no progress, or by measuring the fraction of CPU cycles used for packet processing. Once the system has invested enough work in an incoming packet to the point where it is about to be queued, it makes more sense to process that packet to completion than to drop it and rescue a subsequently-arriving packet from being dropped at the receiving interface, a cycle that could repeat *ad infinitum*.

When the system is about to drop a received packet because an internal queue is full, this strongly suggests that it should disable input interrupts. The host can then make progress on the packets already queued for higher-level processing, which has the side-effect of freeing buffers to use for subsequent received packets. Meanwhile, if the receiving interface has sufficient buffering of its own, additional incoming packets may accumulate there for a while.

We also need a trigger for re-enabling input interrupts, to prevent unnecessary packet loss. Interrupts may be re-enabled when internal buffer space becomes available, or upon expiration of a timer.

We may also want the system to guarantee some progress for user-level code. The system can observe that, over some interval, it has spent too much time processing packet input and output events, and temporarily disable interrupts to give higher protocol layers and user processes time to run. On a processor with a fine-grained clock register, the packet-input code can record the clock value on entry, subtract that from the clock value seen on exit, and keep a sum of the deltas. If this sum (or a running average) exceeds a specified fraction of the total elapsed time, the kernel disables input interrupts. (Digital's GIGAswitch™ system uses a similar mechanism [15].)

On a system without a fine-grained clock, one can crudely simulate this approach by sampling the CPU state on every clock interrupt (clock interrupts typically preempt device interrupt processing). If the system finds itself in the midst of processing interrupts for a series of such samples, it can disable interrupts for a few clock ticks.

5.2. Use of polling

Limiting the interrupt rate prevents system saturation but might not guarantee progress; the system must also fairly allocate packet-handling resources between input and output processing, and between multiple interfaces. We can provide fairness by carefully polling all sources of packet events, using a round-robin schedule.

In a pure polling system, the scheduler would invoke the device driver to ''listen'' for incoming packets and for transmit completion events. This would control the amount of device-level processing, and could also fairly allocate resources among event sources, thus avoiding livelock. Simply polling at fixed intervals, however, adds unacceptable latency to packet reception and transmission.

Polling designs and interrupt-driven designs differ in their placement of policy decisions. When the behavior of tasks cannot be predicted, we rely on the scheduler and the interrupt system to dynamically allocate CPU resources. When tasks can be expected to behave in a predictable manner, the tasks themselves are better able to make the scheduling decisions, and polling depends on voluntary cooperation among the tasks.

Since a purely interrupt-driven system leads to livelock, and a purely polling system adds unnecessary latency, we employ a hybrid design, in which the system polls only when triggered by an interrupt, and interrupts happen only while polling is suspended. During low loads, packet arrivals are unpredictable and we use interrupts to avoid latency. During high loads, we know that packets are arriving at or near the system's saturation rate, so we use polling to ensure progress and fairness, and only re-enable interrupts when no more work is pending.

5.3. Avoiding preemption

As we showed in section 4.2, receive livelock occurs because interrupt processing preempts all other packet processing. We can solve this problem by making higher-level packet processing non-preemptable. We observe that this can be done following one of two general approaches: do (almost) everything at high IPL, or do (almost) nothing at high IPL.

Following the first approach, we can modify the 4.2BSD design (see section 4.1) by eliminating the software interrupt, polling interfaces for events, and processing received packets to completion at device IPL. Because higher-level processing occurs at device IPL, it cannot be preempted by another packet arrival, and so we guarantee that livelock does not occur within the kernel's protocol stack. We still

need to use a rate-control mechanism to ensure progress by user-level applications.

In a system following the second approach, the interrupt handler runs only long enough to set a ‘‘service needed’’ flag, and to schedule the polling thread if it is not already running. The polling thread runs at zero IPL, checking the flags to decide which devices need service. Only when the polling thread is done does it re-enable the device interrupt. The polling thread can be interrupted at most once by each device, and so it progresses at full speed without interference.

Either approach eliminates the need to queue packets between the device driver and the higher-level protocol software, although if the protocol stack must block, the incoming packet must be queued at a later point. (For example, this would happen when the data is ready for delivery to a user process, or when an IP fragment is received and its companion fragments are not yet available.)

5.4. Summary of techniques

In summary, we avoid livelock by:

- Using interrupts only to initiate polling.
- Using round-robin polling to fairly allocate resources among event sources.
- Temporarily disabling input when feedback from a full queue, or a limit on CPU usage, indicates that other important tasks are pending.
- Dropping packets early, rather than late, to avoid wasted work. Once we decide to receive a packet, we try to process it to completion.

We maintain high performance by

- Re-enabling interrupts when no work is pending, to avoid polling overhead and to keep latency low.
- Letting the receiving interface buffer bursts, to avoid dropping packets.
- Eliminating the IP input queue, and associated overhead.

We observe, in passing, that inefficient code tends to exacerbate receive livelock, by lowering the MLFRR of the system and hence increasing the likelihood that livelock will occur. Aggressive optimization, ‘‘fast-path’’ designs, and removal of unnecessary steps all help to postpone arrival of livelock.

6. Livelock in BSD-based routers

In this section, we consider the specific example of an IP packet router built using Digital UNIX (formerly DEC OSF/1). We chose this application because routing performance is easily measured. Also, since firewalls typically use UNIX-based routers, they must be livelock-proof in order to prevent denial-of-service attacks.

Our goals were to (1) obtain the highest possible maximum throughput; (2) maintain high throughput even when overloaded; (3) allocate sufficient CPU cycles to user-mode tasks; (4) minimize latency; and (5) avoid degrading performance in other applications.

6.1. Measurement methodology

Our test configuration consisted of a router-under-test connecting two otherwise unloaded Ethernets. A source host generated IP/UDP packets at a variety of rates, and sent them via the router to a destination address. (The destination host did not exist; we fooled the router by inserting a phantom entry into its ARP table.) We measured router performance by counting the number of packets successfully forwarded in a given period, yielding an average forwarding rate.

The router-under-test was a DECstation™ 3000/300 Alpha-based system running Digital UNIX V3.2, with a SPECint92 rating of 66.2. We chose the slowest available Alpha host, to make the livelock problem more evident. The source host was a DECstation 3000/400, with a SPECint92 rating of 74.7. We slightly modified its kernel to allow more efficient generation of output packets, so that we could stress the router-under-test as much as possible.

In all the trials reported on here, the packet generator sent 10000 UDP packets carrying 4 bytes of data. This system does not generate a precisely paced stream of packets; the packet rates reported are averaged over several seconds, and the short-term rates varied somewhat from the mean. We calculated the delivered packet rate by using the ‘‘netstat’’ program (on the router machine) to sample the output interface count (‘‘Opkts’’) before and after each trial. We checked, using a network analyzer on the stub Ethernet, that this count exactly reports the number of packets transmitted on the output interface.

6.2. Measurements of an unmodified kernel

We started by measuring the performance of the unmodified operating system, as shown in figure 6-1. Each mark represents one trial. The filled circles show kernel-based forwarding performance, and the open squares show performance using the *screend* program [7], used in some firewalls to screen out unwanted packets. This user-mode program does one system call per packet; the packet-forwarding path includes both kernel and user-mode code. In this case, *screend* was configured to accept all packets.

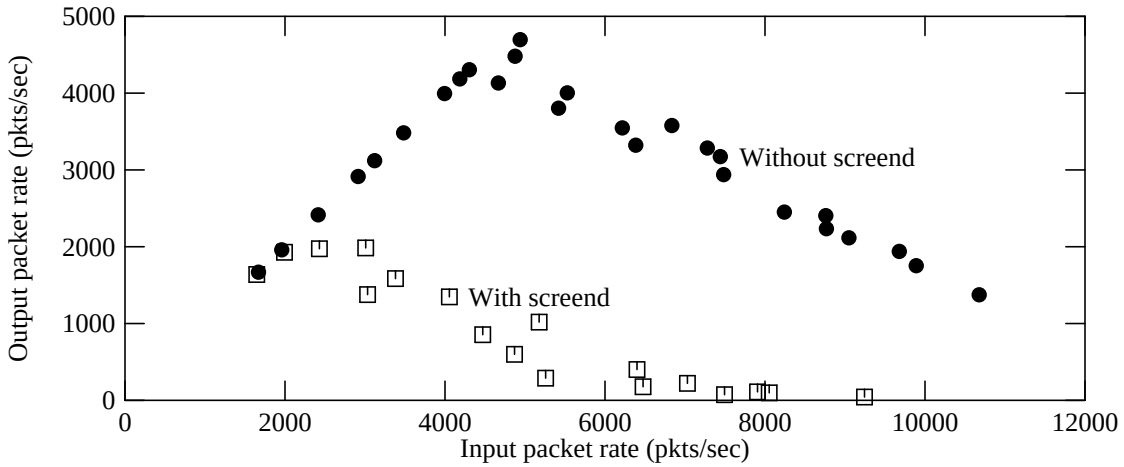

Figure 6-1: Forwarding performance of unmodified kernel

From these tests, it was clear that with *screend* running, the router suffered from poor overload behavior at rates above 2000 packets/sec., and complete livelock set in at about 6000 packets/sec. Even without *screend*, the router peaked at 4700 packets/sec., and would probably livelock somewhat below the maximum Ethernet packet rate of about 14,880 packets/second.

6.3. Why livelock occurs in the 4.2BSD model

4.2BSD follows the model described in section 4.1, and depicted in figure 6-2. The device driver runs at interrupt priority level (IPL) = SPLIMP, and the IP layer runs via a software interrupt at IPL = SPLNET, which is lower than SPLIMP. The queue between the driver and the IP code is named ‘‘ipintrq,’’ and each output interface is buffered by a queue of its own. All queues have length limits; excess packets are dropped. Device drivers in this system implement interrupt batching, so at high input rates very few interrupts are actually taken.

Digital UNIX follows a similar model, with the IP layer running as a separately scheduled thread at IPL = 0, instead of as a software interrupt handler.

It is now quite obvious why the system suffers from receive livelock. Once the input rate exceeds the rate at which the device driver can pull new packets out of the interface and add them to the IP input queue, the IP code never runs. Thus, it never removes packets from its queue (ipintrq), which fills up, and all subsequent received packets are dropped.

The system’s CPU resources are saturated because it discards each packet after a lot of CPU time has been invested in it at elevated IPL. This is foolish; once a packet has made its way through the device driver, it represents an investment and should be processed to completion if at all possible. In a router, this means that the packet should be transmitted on the output interface. When the system is overloaded, it should discard packets as early as possible (i.e., in the receiving interface), so that discarded packets do not waste any resources.

6.4. Fixing the livelock problem

We solved the livelock problem by doing as much work as possible in a kernel thread, rather than in the interrupt handler, and by eliminating the IP input queue and its associated queue manipulations and software interrupt (or thread dispatch)[1]. Once we decide to take a packet from the receiving interface, we try not to discard it later on, since this would represent wasted effort.

We also try to carefully ‘‘schedule’’ the work done in this thread. It is probably not possible to use the system’s real scheduler to control the handling of each packet, so we instead had this thread use a polling technique to efficiently simulate round-robin scheduling of packet processing. The polling thread uses additional heuristics to help meet our performance goals.

In the new system, the interrupt handler for an interface driver does almost no work at all. Instead, it simple schedules the polling thread (if it has not already been scheduled), recording its need for packet processing, and then returns from the interrupt. It does not set the device’s interrupt-enable flag, so the system will not be distracted with additional interrupts until the polling thread has processed all of the pending packets.

At boot time, the modified interface drivers register themselves with the polling system, provid-

[1]This is not such a radical idea; Van Jacobson had already used it as a way to improve end-system TCP performance [4].

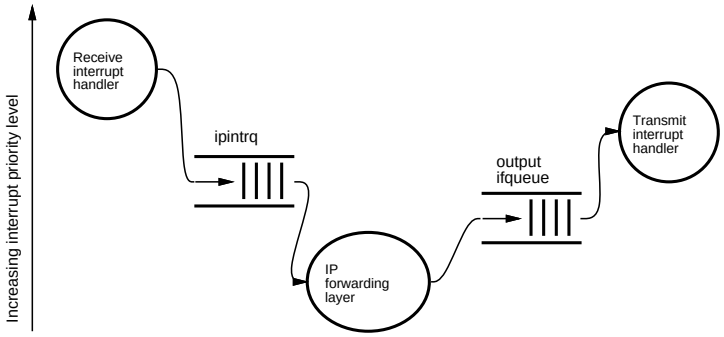

Figure 6-2: IP forwarding path in 4.2BSD

ing callback procedures for handling received and transmitted packets, and for enabling interrupts. When the polling thread is scheduled, it checks all of the registered devices to see if they have requested processing, and invokes the appropriate callback procedures to do what the interrupt handler would have done in the unmodified kernel.

The received-packet callback procedures call the IP input processing routine directly, rather than placing received packets on a queue for later processing; this means that any packet accepted from the interface is processed as far as possible (e.g., to the output interface queue for forwarding, or to a queue for delivery to a process). If the system falls behind, the interface's input buffer will soak up packets for a while, and any excess packets will be dropped by the interface before the system has wasted any resources on it.

The polling thread passes the callback procedures a quota on the number of packets they are allowed to handle. Once a callback has used up its quota, it must return to the polling thread. This allows the thread to round-robin between multiple interfaces, and between input and output handling on any given interface, to prevent a single input stream from monopolizing the CPU.

Once all the packets pending at an interface have been handled, the polling thread also invokes the driver's interrupt-enable callback so that a subsequent packet event will cause an interrupt.

6.5. Results and analysis

Figures 6-3 summarizes the results of our changes, when *screend* is not used. Several different kernel configurations are shown, using different mark symbols on the graph. The modified kernel (shown with square marks) slightly improves the MLFRR, and avoids livelock at higher input rates.

The modified kernel can be configured to act as if it were an unmodified system (shown with open circles), although this seems to perform slightly worse than an actual unmodified system (filled circles). The reasons are not clear, but may involve slightly longer code paths, different compilers, or unfortunate changes in instruction cache conflicts.

6.6. Scheduling heuristics

Figure 6-3 shows that if the polling thread places no quota on the number of packets that a callback procedure can handle, when the input rate exceeds the MLFRR the total throughput drops almost to zero (shown with diamonds in the figure). This livelock occurs because although the packets are no longer discarded at the IP input queue, they are still piling up (and being discarded) at the queue for the output interface. This queue is unavoidable, since there is no guarantee that the output interface runs as fast as the input interface.

Why does the system fail to drain the output queue? If packets arrive too fast, the input-handling callback never finishes its job. This means that the polling thread never gets to call the output-handling callback for the transmitting interface, which prevents the release of transmitter buffer descriptors for use in further packet transmissions. This is similar to the transmit starvation condition identified in section 4.4.

The result is actually worse in the no-quota modified kernel, because in that system, packets are discarded for lack of space on the output queue, rather than on the IP input queue. The unmodified kernel does less work per discarded packet, and therefore occasionally discards them fast enough to catch up with a burst of input packets.

6.6.1. Feedback from full queues

How does the modified system perform when the *screend* program is used? Figure 6-4 compares the performance of the unmodified kernel (filled circles) and several modified kernels.

With the kernel modified as described so far (squares), the system performs about as badly as the

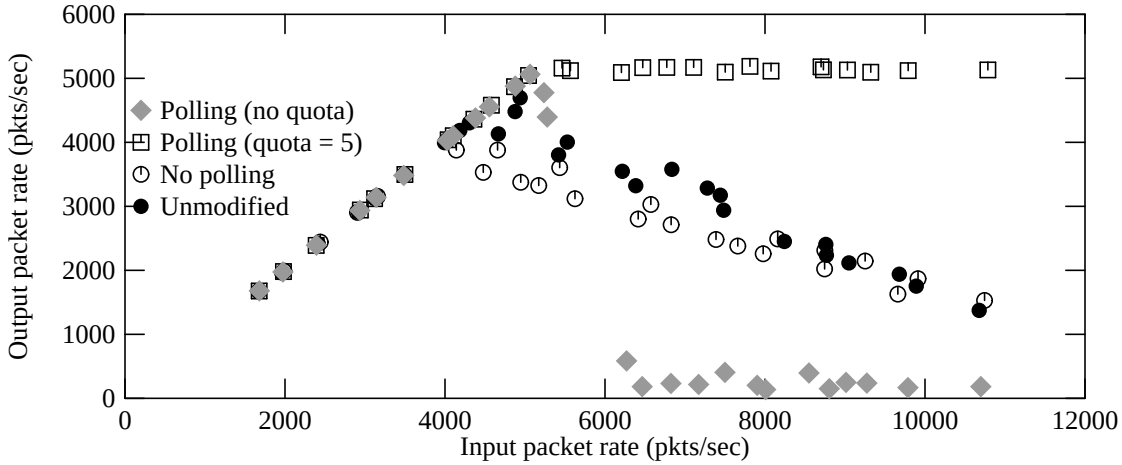

Figure 6-3: Forwarding performance of modified kernel, without using *screend*

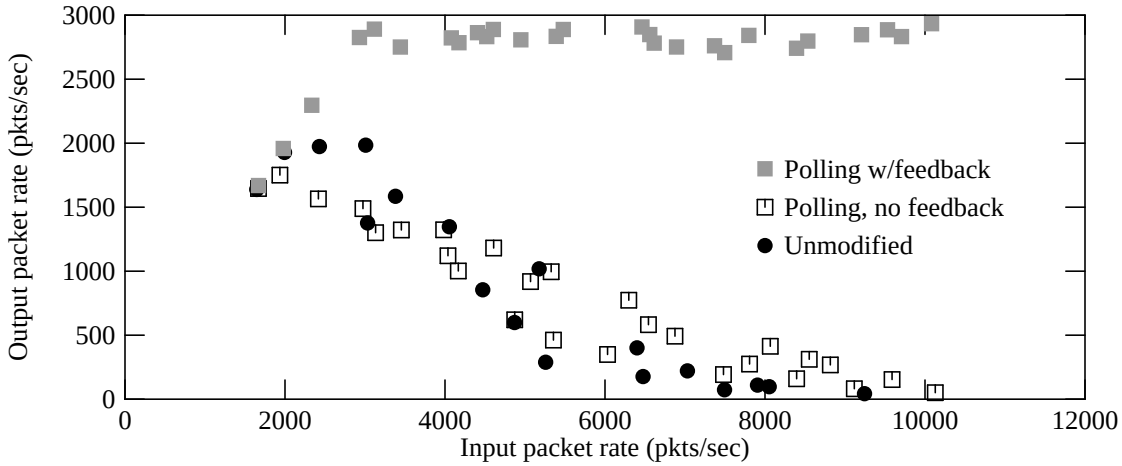

Figure 6-4: Forwarding performance of modified kernel, with *screend*

unmodified kernel. The problem is that, because *screend* runs in user mode, the kernel must queue packets for delivery to *screend*. When the system is overloaded, this queue fills up and packets are dropped. *screend* never gets a chance to run to drain this queue, because the system devotes its cycles to handling input packets.

To resolve this problem, we detect when the screening queue becomes full and inhibit further input processing (and input interrupts) until more queue space is available. The result is shown with the gray square marks in figure 6-4: no livelock, and much improved peak throughput. Feedback from the queue state means that the system properly allocates CPU resources to move packets all the way through the system, instead of dropping them at an intermediate point.

In these experiments, the polling quota was 10 packets, the screening queue was limited to 32 packets, and we inhibited input processing when the queue was 75% full. Input processing is re-enabled when the screening queue becomes 25% full. We chose these high and low water marks arbitrarily, and some tuning might help. We also set a timeout (arbitrarily chosen as one clock tick, or about 1 msec) after which input is re-enabled, in case the *screend* program is hung, so that packets for other consumers are not dropped indefinitely.

The same queue-state feedback technique could be applied to other queues in the system, such as interface output queues, packet filter queues (for use in network monitoring) [9, 8], etc. The feedback policies for these queues would be more complex, since it might be difficult to determine if input processing load was actually preventing progress at these queues. Since the *screend* program is typically run as the only application on a system, however, a full screening queue is an unequivocal signal that too many packets are arriving.

6.6.2. Choice of packet-count quota

To avoid livelock in the non-*screend* configuration, we had to set a quota on the number of packets processed per callback, so we investigated how system throughput changes as the quota is varied. Figure 6-5 shows the results; smaller quotas work

better. As the quota increases, livelock becomes more of a problem.

When *screend* is used, however, the queue-state feedback mechanism prevents livelock, and small quotas slightly reduce maximum throughput (by about 5%). We believe that by processing more packets per callback, the system amortizes the cost of polling more effectively, but increasing the quota could also increase worst-case per-packet latency. Once the quota is large enough to fill the screening queue with a burst of packets, the feedback mechanism probably hides any potential for improvement.

Figure 6-6 shows the results when the *screend* process is in use.

In summary, tests both with and without *screend* suggest that a quota of between 10 and 20 packets yields stable and near-optimum behavior, for the hardware configuration tested. For other CPUs and network interfaces, the proper value may differ, so this parameter should be tunable.

7. Guaranteeing progress for user-level processes

The polling and queue-state feedback mechanisms described in section 6.4 can ensure that all necessary phases of packet processing make progress, even during input overload. They are indifferent to the needs of other activities, however, so user-level processes could still be starved for CPU cycles. This makes the system's user interface unresponsive and interferes with housekeeping tasks (such as routing table maintenance).

We verified this effect by running a compute-bound process on our modified router, and then flooding the router with minimum-sized packets to be forwarded. The router forwarded the packets at the full rate (i.e., as if no user-mode process were consuming resources), but the user process made no measurable progress.

Since the root problem is that the packet-input handling subsystem takes too much of the CPU, we should be able to ameliorate that by simply measuring the amount of CPU time spent handling received packets, and disabling input handling if this exceeds a threshold.

The Alpha architecture, on which we did these experiments, includes a high-resolution low-overhead counter register. This register counts every instruction cycle (in current implementations) and can be read in one instruction, without any data cache misses. Other modern RISC architectures support similar counters; Intel's Pentium is known to have one as an unsupported feature.

We measure the CPU usage over a period defined as several clock ticks (10 msec, in our current implementation, chosen arbitrarily to match the scheduler's quantum). Once each period, a timer function clears a running total of CPU cycles used in the packet-processing code.

Each time our modified kernel begins its polling loop, it reads the cycle counter, and reads it again at the end of the loop, to measure the number of cycles spent handling input and output packets during the loop. (The quota mechanism ensures that this interval is relatively short.) This number is then added to the running total, and if this total is above a threshold, input handling is immediately inhibited. At the end of the current period, a timer re-enables input handling. Execution of the system's idle thread also re-enables input interrupts and clears the running total.

By adjusting the threshold to be a fraction of the total number of cycles in a period, one can control fairly precisely the amount of CPU time spent processing packets. We have not yet implemented a programming interface for this control; for our tests, we simply patched a kernel global variable representing the percentage allocated to network processing, and the kernel automatically translates this to a number of cycles.

Figure 7-1 shows how much CPU time is available to a compute-bound user process, for several settings of the cycle threshold and various input rates. The curves show fairly stable behavior as the input rate increases, but the user process does not get as much CPU time as the threshold setting would imply.

Part of the discrepancy comes from system overhead; even with no input load, the user process gets about 94% of the CPU cycles. Also, the cycle-limit mechanism inhibits packet input processing but not output processing. At higher input rates, before input is inhibited, the output queue fills enough to soak up additional CPU cycles.

Measurement error could cause some additional discrepancy. The cycle threshold is checked only after handling a burst of input packets (for these experiments, the callback quota was 5 packets). With the system forwarding about 5000 packets/second, handling such a burst takes about 1 msec, or about 10% of the threshold-checking period.

The initial dips in the curves for the 50% and 75% thresholds probably reflect the cost of handling the actual interrupts; these cycles are not counted against the threshold, and at input rates below saturation, each incoming packet may be handled fast enough that no interrupt batching occurs.

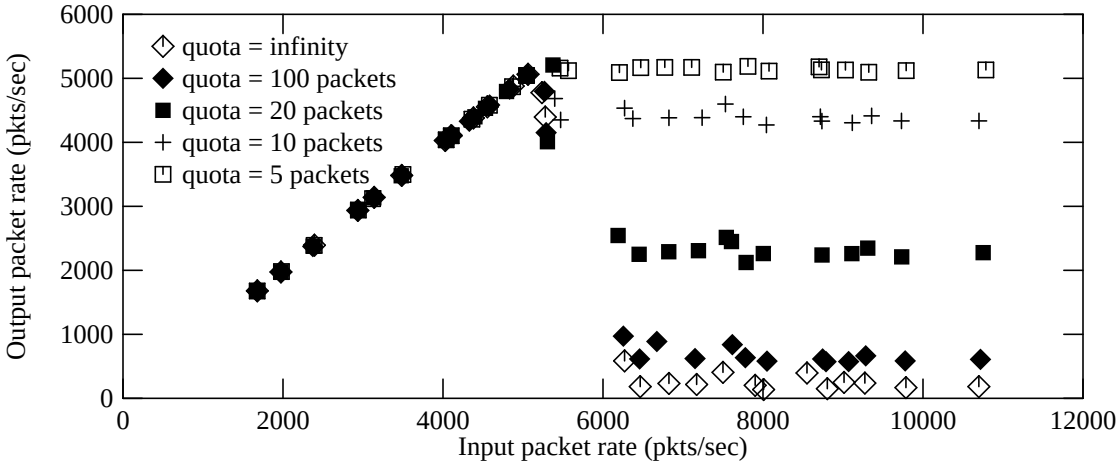

Figure 6-5: Effect of packet-count quota on performance, no *screend*

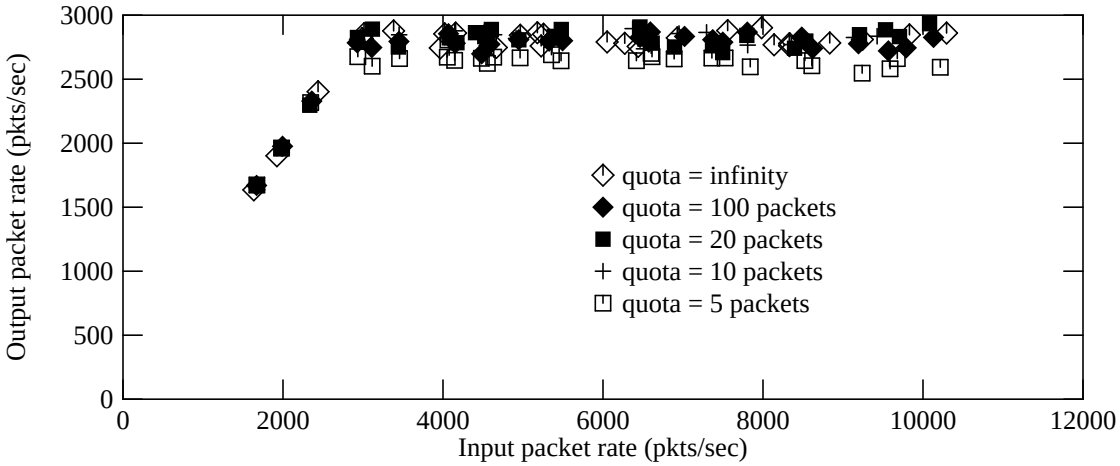

Figure 6-6: Effect of packet-count quota on performance, with *screend*

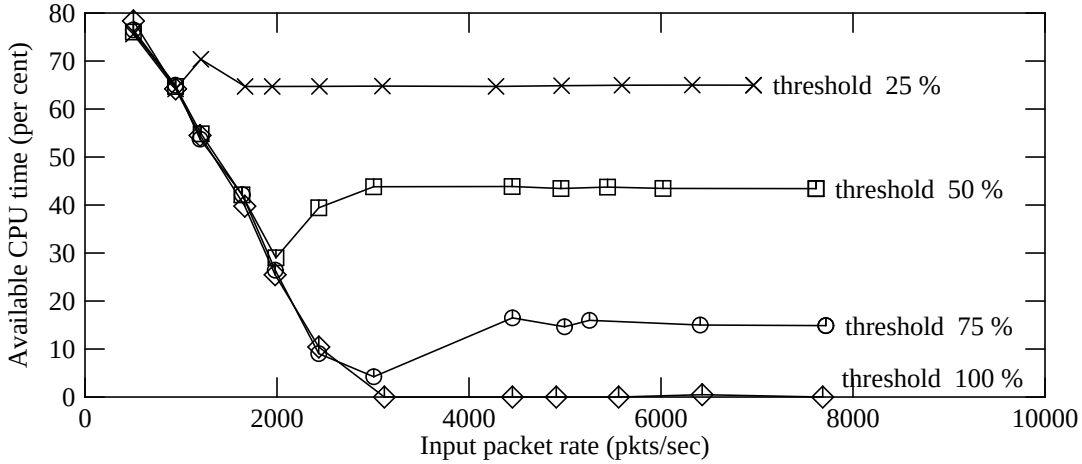

Figure 7-1: User-mode CPU time available using cycle-limit mechanism

With a cycle-limit imposed on packet processing, the system is subjectively far more responsive, even during heavy input overload. This improvement, however, is mostly apparent for local users; any network-based interaction, such as Telnet, still suffers because many packets are being dropped.

7.1. Performance of end-system transport protocols

The changes we made to the kernel potentially affect the performance of end-system transport protocols, such as TCP and the UDP/RPC/XDR/NFS stack. Since we have not yet applied our modifications to a high-speed network interface driver, such

as one for FDDI, we cannot yet measure this effect. (The test system can easily saturate an Ethernet, so measuring TCP throughput over Ethernet shows no effect.)

The technique of processing a received packet directly from the device driver to the TCP layer, without placing the packet on an IP-level queue, was used by Van Jacobson specifically to improve TCP performance [4]. It should reduce the cost of receiving a packet, by avoiding the queue operations and any associated locking; it also should improve the latency of kernel-to-kernel interactions (such as TCP acknowledgements and NFS RPCs).

The technique of polling the interfaces should not reduce end-system performance, because it is done primarily during input overload. (Some implementations use polling to avoid transmit interrupts altogether [6].) During overload, the unmodified system would not make any progress on applications or transport protocols; the use of polling, queue-state feedback, and CPU cycle limits should give the modified system a chance to make at least some progress.

8. Related work

Polling mechanisms have been used before in UNIX-based systems, both in network code and in other contexts. Whereas we have used polling to provide fairness and guaranteed progress, the previous applications of polling were intended to reduce the overhead associated with interrupt service. This does reduce the chances of system overload (for a given input rate), but does not prevent livelock.

Traw and Smith [14, 16] describe the use of ‘‘clocked interrupts,’’ periodic polling to learn of arriving packets without the overhead of per-packet interrupts. They point out that it is hard to choose the proper polling frequency: too high, and the system spends all its time polling; too low, and the receive latency soars. Their analysis [14] seems to ignore the use of interrupt batching to reduce the interrupt-service overhead; however, they do allude to the possibility of using a scheme in which an interrupt prompts polling for other events.

The 4.3BSD operating system [5] apparently used a periodic polling technique to process received characters from an eight-port terminal interface, if the recent input rate increased above a certain threshold. The intent seems to have been to avoid losing input characters (the device had little buffering available) but one could view this as a sort of livelock-avoidance strategy. Several router implementations use polling as their primary way to schedule packet processing.

When a congested router must drop a packet, its choice of which packet to drop can have significant effects. Our modifications do not affect *which* packets are dropped; we only change *when* they are dropped. The policy was and remains ‘‘drop-tail’’; other policies might provide better results [3].

Some of our initial work on improved interface driver algorithms is described in [1].

9. Summary and conclusions

Systems that behave poorly under receive overload fail to provide consistent performance and good interactive behavior. Livelock is never the best response to overload. In this paper, we have shown how to understand system overload behavior and how to improve it, by carefully scheduling when packet processing is done.

We have shown, using measurements of a UNIX system, that traditional interrupt-driven systems perform badly under overload, resulting in receive livelock and starvation of transmits. Because such systems progressively reduce the priority of processing a packet as it goes further into the system, when overloaded they exhibit excessive packet loss and wasted work. Such pathologies may be caused not only by long-term receive overload, but also by transient overload from short-term bursty arrivals.

We described a set of scheduling improvements that help solve the problem of poor overload behavior. These include:

- Limiting interrupt arrival rates, to shed overload
- Polling to provide fairness
- Processing received packets to completion
- Explicitly regulating CPU usage for packet processing

Our experiments showed that these scheduling mechanisms provide good overload behavior and eliminate receive livelock. They should help both special-purpose and general-purpose systems.

Acknowledgements

We had help both in making measurements and in understanding system performance from many people, including Bill Hawe, Tony Lauck, John Poulin, Uttam Shikarpur, and John Dustin. Venkata Padmanabhan, David Cherkus, and Jeffry Yaplee helped during manuscript preparation.

Most of K. K. Ramakrishnan’s work on this paper was done while he was an employee of Digital Equipment Corporation.

References

[1] Chran-Ham Chang, R. Flower, J. Forecast, H. Gray, W. R. Hawe, A. P Nadkarni, K. K. Ramakrishnan, U. N. Shikarpur, and K. M. Wilde. High-performance TCP/IP and UDP/IP Networking in DEC OSF/1 for Alpha AXP. *Digital Technical Journal* 5(1):44-61, Winter, 1993.

[2] Domenico Ferrari, Joseph Pasquale, and George C. Polyzos. *Network Issues for Sequoia 2000*. Sequoia 2000 Technical Report 91/6, University of California, Berkeley, December, 1991.

[3] Sally Floyd and Van Jacobson. Random Early Detection gateways for Congestion Avoidance. *Trans. Networking* 1(4):397-413, August, 1993.

[4] Van Jacobson. Efficient Protocol Implementation. Notes from SIGCOMM '90 Tutorial on ''Protocols for High-Speed Networks''. 1990.

[5] Samuel J. Leffler, Marshall Kirk McCusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System.* Addison-Wesley, Reading, MA, 1989.

[6] Rick Macklem. Lessons Learned Tuning The 4.3BSD Reno Implementation of the NFS Protocol. In *Proc. Winter 1991 USENIX Conference*, pages 53-64. Dallas, TX, January, 1991.

[7] Jeffrey C. Mogul. Simple and Flexible Datagram Access Controls for Unix-based Gateways. In *Proc. Summer 1989 USENIX Conference*, pages 203-221. Baltimore, MD, June, 1989.

[8] Jeffrey C. Mogul. Efficient Use Of Workstations for Passive Monitoring of Local Area Networks. In *Proc. SIGCOMM '90 Symposium on Communications Architectures and Protocols*, pages 253-263. ACM SIGCOMM, Philadelphia, PA, September, 1990.

[9] Jeffrey C. Mogul, Richard F. Rashid, Michael J. Accetta. The Packet Filter: An Efficient Mechanism for User-Level Network Code. In *SOSP11*, pages 39-51. Austin, Texas, November, 1987.

[10] Radia Perlman. Fault-Tolerant Broadcast of Routing Information. *Computer Networks* 7(6):395-405, December, 1983.

[11] K. K. Ramakrishnan. Scheduling Issues for Interfacing to High Speed Networks. In *Proc. Globecom '92 IEEE Global Telecommunications Conf.*, pages 622-626. Orlando, FL, December, 1992.

[12] K. K. Ramakrishnan. Performance Considerations in Designing Network Interfaces. *IEEE Journal on Selected Areas in Communications* 11(2):203-219, February, 1993.

[13] Marcus J. Ranum and Frederick M. Avolio. A Toolkit and Methods for Internet Firewalls. In *Proc. Summer 1994 USENIX Conference*, pages 37-44. Boston, June, 1994.

[14] Jonathan M. Smith and C. Brendan S. Traw. Giving Applications Access to Gb/s Networking. *IEEE Network* 7(4):44-52, July, 1993.

[15] Robert J. Souza, P. G. Krishnakumar, Cüneyt M. Özveren, Robert J.Simcoe, Barry A. Spinney, Robert E. Thomas, and Robert J. Walsh. GIGAswitch: A High-Performance Packet Switching Platform. *Digital Technical Journal* 6(1):9-22, Winter, 1994.

[16] C. Brendan S. Traw and Jonathan M. Smith. Hardware/Software Organization of a High-Performance ATM Host Interface. *IEEE Journal on Selected Areas in Communications* 11(2):240-253, February, 1993.

Jeffrey Mogul received an S.B. from the Massachusetts Institute of Technology in 1979, and his M.S. and Ph.D. degrees from Stanford University in 1980 and 1986. Since 1986, he has been a researcher at Digital's Western Research Laboratory, working on network and operating systems issues for high-performance computer systems. He is the author or co-author of several Internet Standards, an associate editor of *Internetworking: Research and Experience*, and was Program Chair for the Winter 1994 USENIX Conference.

Address for correspondence: Digital Equipment Corp. Western Research Lab, 250 University Ave., Palo Alto, CA, 94301 (mogul@wrl.dec.com)

K. K. Ramakrishnan is a Member of Technical Staff at AT&T Bell Laboratories. He holds a B.S. from Bangalore University in India in 1976, an M.S. from the Indian Institute of Science in 1978, and a Ph.D. from the University of Maryland in 1983. Until 1994, he was a Consulting Engineer at Digital. Ramakrishnan's research interests are in performance analysis and design of algorithms for computer networks and distributed systems. He is a technical editor for *IEEE Network Magazine* and is a member of the Internet Research Task Force's End-End Research Group.

Address for correspondence: AT&T Bell Laboratories, 600 Mountain Ave., Murray Hill, NJ, 07974 (kkrama@research.att.com)