Eliminating Receive Livelock in an Interrupt-Driven Kernel

JEFFREY C. MOGUL
Digital Equipment Corporation Western Research Laboratory
and
K. K. RAMAKRISHNAN
AT&T Labs – Research

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 of 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 modification eliminates receive livelock without degrading other aspects of system performance. Our modifications include the use of polling when the system is heavily loaded, while retaining the use of interrupts under lighter load. We present measurements demonstrating the success of our approach.

Categories and Subject Descriptors: C.2 [**Computer Systems Organization**]: Computer-Communication Networks; D.4 [**Software**]: Operating Systems; D.4.1 [**Operating Systems**]: Process Management—*scheduling*; D.4.4 [**Operating Systems**]: Communications Management—*input/output; network communication*

General Terms: Performance

Additional Key Words and Phrases: Interrupt-driven kernel, livelock, polling, scheduling

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

Author's addresses: J. C. Mogul, Digital Equipment Corporation Western Research Laboratory, 250 University Avenue, Palo Alto, CA 94301; email: mogul@wrl.dec.com; K. K. Ramakrishnan: AT&T Labs – Research 180 Park Avenue, Florham Park, NJ 07932; email: kkrama@research.att.com.

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 multimedia and client-server applications share another unpleasant property: unlike traditional network applications (Telnet, FTP, electronic mail), they are not flow-controlled. Some multimedia applications want constant-rate, low-latency service. RPC-based client-server applications often use datagram-style transports, instead of reliable, flow-controlled protocols. For example, the most common UNIX NFS client implementation can generate numerous RPC requests in parallel from one client host. And even when a particular instance of an application is flow-controlled, when the number of potential clients is large or unbounded (e.g., an Internet Web server), the system under load has no way to defer requests from new clients. 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.

In short, temporary overload conditions are a fact of life for many kinds of systems. It may be infeasible to configure such systems to accommodate the peak potential load, yet we certainly would prefer that they respond gracefully to peaks: shedding or deferring load, rather than collapsing.

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 subsystem into the operating system's scheduling mechanisms and policies. In this article, we present a number of simple modifications to the purely interrupt-driven model. We start with a hybrid design in which the system polls only when triggered by an interrupt, and interrupts happen only when polling is suspended; this provides low latency under low loads, and high throughput under high loads. We augment the design with simple feedback control, so that when the system is overloaded and must drop packets, it drops the ones in which it has the least investment. We also create a simple connection between the traditional scheduling system and the network subsystem, in order to guarantee some CPU time to user tasks even during periods of overload.

Later in the article, we describe the results of benchmarks demonstrating that our modifications do indeed guarantee throughput and fairness under overload, while also improving peak throughput and latency, and still 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 internetwork 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 [Mogul 1989; Ranum and Avolio 1994]. Much experimentation with new routing algorithms is done on UNIX [Ferrari et al. 1991], 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 [Mogul 1990].

—*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.

This problem is not simply of theoretical interest. We have encountered livelock in all three of these applications, either in real-life use, or when measuring system performance using standard benchmarking techniques. The potential for livelock is also a security problem, since it leaves a system open to a simple denial-of-service attack. For all three of these applications, our techniques have solved or mitigated the problem, and we have shipped the solutions to customers. For example, an earlier implementation of this work was successfully deployed in the routers used for the NASDAQ financial network.

The rest of this article concentrates on host-based routing and (to a lesser extent) network monitoring, 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.

3.1 Throughput

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; a similar term was first used by Ramakrishnan [1992]), 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.

3.2 Latency and Jitter

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.

3.3 Fair Allocation of Resources

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.

3.4 Overall Stability

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 [Perlman 1983].

3.5 Summary of Requirements

The scheduling of network activity should guarantee:

—High throughput for both input and output, and no loss of throughput during overload conditions.

—Low latency and low jitter, even during overload.

—Fair allocation of CPU and memory resources, both among networking tasks, and to non-networking tasks as well.

The approach described in this article meets these requirements.

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 caused 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 [Leffler et al. 1989] model for our example, but we have observed that other operating systems, such as VMS, MS-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. (The system design avoids lengthy periods at high IPL, in order to reduce latency for handling certain other events, such as lower-priority device interrupts.)

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 article.

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 Ramakrishnan [1992], 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 Ramakrishnan [1992].

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.

To demonstrate this effect, we performed experiments using ULTRIX Version 3.0 running on a DECstation 3100 (approximately 11.3 SPECmarks). ULTRIX, derived from 4.2BSD, closely follows the network design of that system. We used a logic analyzer to measure the time between the generation of an interrupt by the Ethernet device (an AMD 7990 LANCE chip), signalling the complete reception of a packet, and the packet's delivery to an application. We used the kernel's implementation of a simple data-link layer protocol, rather than IP/TCP or a similar protocol stack, but the steps performed by the kernel are substantially the same:

—link-level processing at device IPL, which includes copying the packet into kernel buffers (the interface does not support DMA)

—further processing following a software interrupt, which includes locating the appropriate user process, and queuing the packet for delivery to this process

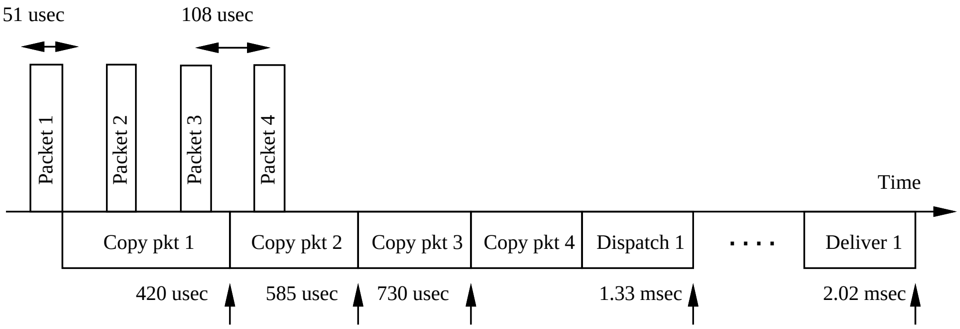

Fig. 1. How interrupt-driven scheduling causes excess latency under overload.

—finally, awakening the user process, which then (in kernel mode) copies the received packet into its own buffer.

Figure 1 shows a time line for the completion of these processing stages, when receiving a burst of four minimum-size packets from the Ethernet. The system starts to copy the first packet into a kernel buffer almost immediately after it arrives, but does not finish copying the third packet until about 1.33 msec. later. Only after finishing this does it schedule a software interrupt to dispatch the packet to the user process, and all of the packets are dispatched before the user process is awakened. It is the use of preemptive interrupt priorities that prevents completion of processing for the first packet until substantial processing has been done on the entire burst.

We generated our bursts of Ethernet packets with an interpacket spacing of 108 μsec. (this is not the minimum theoretical spacing, but we were limited by the packet generator we used). The latency to deliver the first packet to the user application depended on the size of a burst: 1.23 msec. for a single-packet burst, 1.54 msec. for a two-packet burst, 2.02 msec. for a four-packet burst, and 5.03 msec. for a 16-packet burst. A plot of first-packet delivery latency versus burst size (not shown, for reasons of space) reveals that the latency is nearly linear in the burst size, for a wide range of packet sizes.

We will present a more detailed analysis of receive latency in Section 8, in the context of a somewhat different system.

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 [Ramakrishnan 1993].

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-mode tasks 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 from that particular interface. (It is not necessary to disable all system 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 reenabling input interrupts, to prevent unnecessary packet loss. Interrupts may be reenabled 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 [Souza et al. 1994].)

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 reenable 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 nonpreemptable. 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, one could 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. One would still need to use a rate-control mechanism to ensure progress by user-level applications.

We used this first approach in an earlier prototype, where it did provide a sufficient solution to the livelock problem, but using IPL to prevent preemption has several drawbacks. For example, it can interfere with assumptions made elsewhere in the kernel about the IPL.

In a system following the second approach, which we followed for the work described in this article, 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 reenable the device interrupt. The polling thread can be interrupted at most once by each device, and so it progresses at full speed without interference. Note that this does not require fully nonpreemptable threads; we prevent livelock caused by interrupt-driven preemption by disabling the generation of interrupts, not by making the thread nonpreemptable.

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 segment 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

—Reenabling 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,

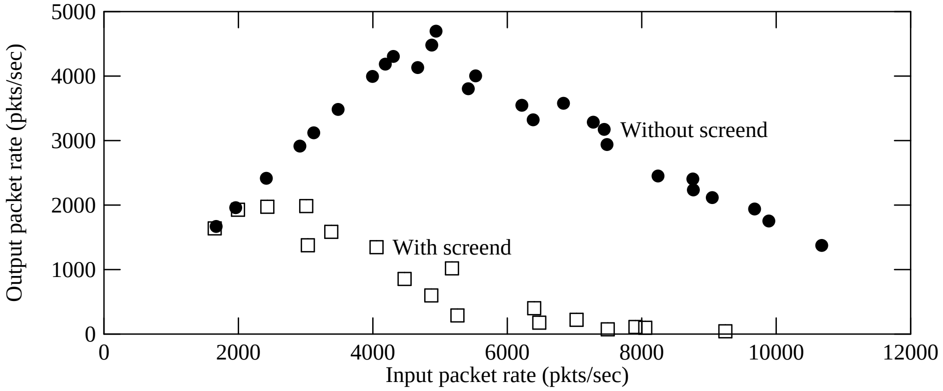

Fig. 2. Forwarding performance of unmodified kernel.

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 10,000 UDP packets carrying four 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 2. Each mark represents one trial. The filled circles show kernel-based forwarding performance, and the open squares show performance using the *screend* program [Mogul 1989], 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.

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.

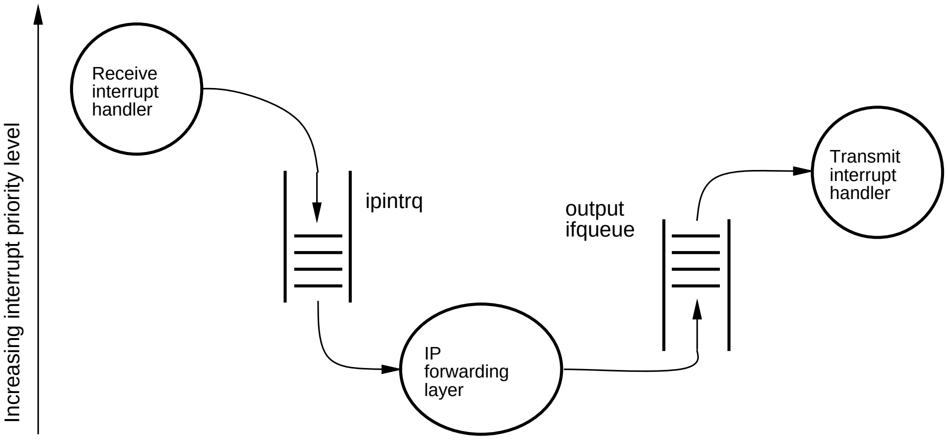
Increasing interrupt priority level
Receive interrupt handler
ipintrq
output ifqueue
Transmit interrupt handler
IP forwarding layer

Fig. 3. IP forwarding path in 4.2BSD.

6.3 Why Livelock Occurs in the 4.2BSD Model

4.2BSD follows the model described in Section 4.1, and depicted in Figure 3. 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]

[1] This is not such a radical idea; Van Jacobson had already used it as a way to improve end-system TCP performance [Jacobson 1990].

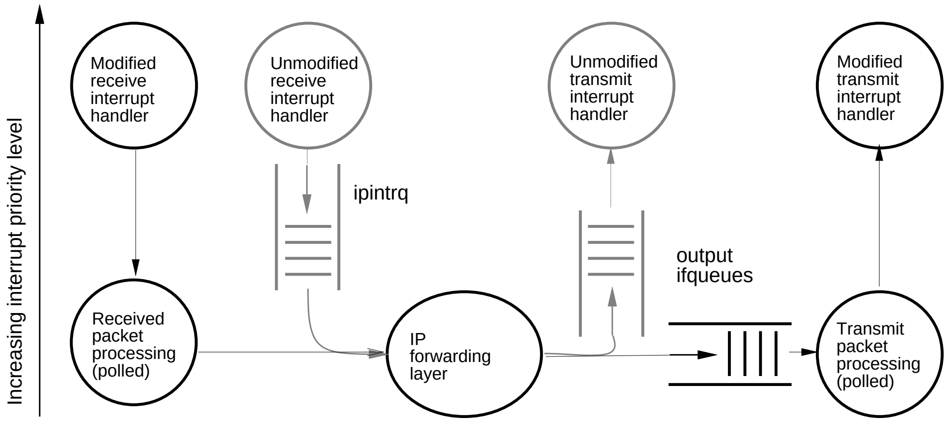
Increasing interrupt priority level
Modified receive interrupt handler
Unmodified receive interrupt handler
Unmodified transmit interrupt handler
Modified transmit interrupt handler
ipintrq
output ifqueues
Received packet processing (polled)
IP forwarding layer
Transmit packet processing (polled)

Fig. 4. Modified IP forwarding path, with polled functions.

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 (see Section 6.6).

In the new system, the interrupt handler for an interface driver does almost no work at all. Instead, it simply 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. The interrupt-enable flag will be set later, once there is no further work pending for this interface.

At boot time, the modified interface drivers register themselves with the polling system, providing 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.

Figure 4 depicts the processing flow, for forwarded packets, in the modified kernel (compare it to the unmodified kernel, depicted in Figure 3.) The original, purely interrupt-driven mechanism is still available for unmodified network interface drivers (shown in gray), but modified drivers use the polling mechanism. Here,

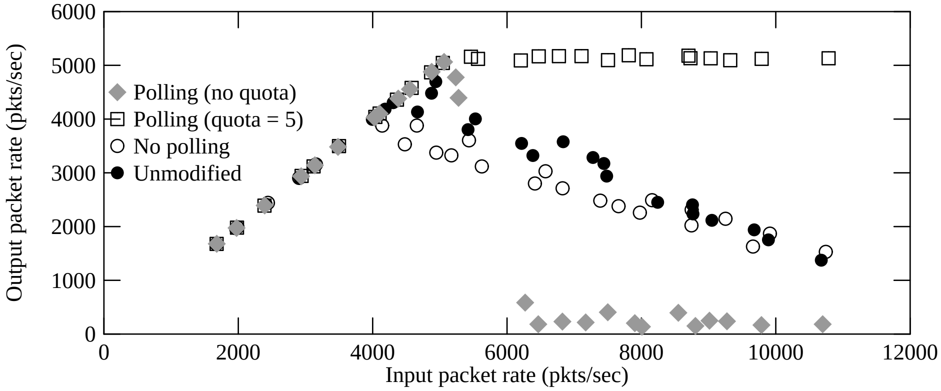

Fig. 5. Forwarding performance of modified kernel, without using *screend*.

the "modified receive interrupt handler" and "modified transmit interrupt handler" simply alert the polling thread to the availability of work. All of the actual packet processing is done in the "received packet processing" and "transmit packet processing" routines, invoked at low IPL by the polling thread. (The polling mechanism itself is not shown in this figure.) The modified path does not use the IP input queue at all, although it remains available for use by unmodified device drivers. We retain the queue between the IP forwarding layer and the transmit packet processing code, since an attempt to transmit a packet might block if the output interface is busy.

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. After 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 5 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 5 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

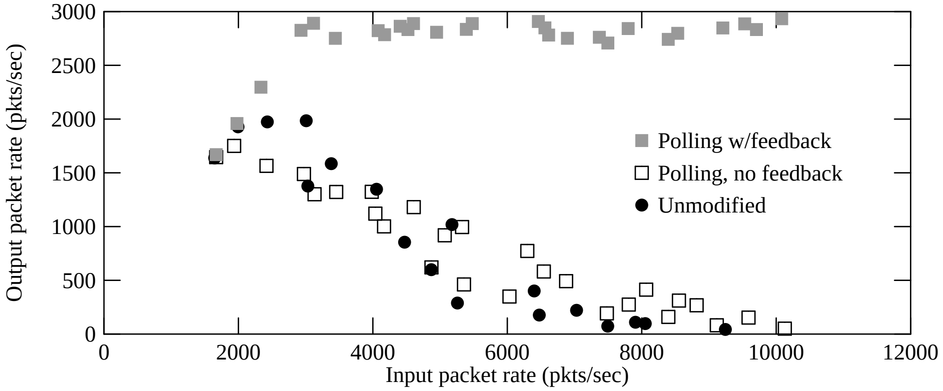

Fig. 6. Forwarding performance of modified kernel, with *screend*.

the total forwarding 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 for the modified kernel, when no quota is imposed, is actually worse than that for the unmodified system, because in the modified system packets are being 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 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 unmodified kernel. The problem is that, because *screend* runs in user mode, the kernel must queue packets for delivery to *screend*. The kernel portion of the *screend* implementation includes a special queue for these packets. 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: no livelock, and much improved peak throughput (the MLFRR increases from under 2000 packets/sec. to over 2900 packets/sec.) Feedback from the queue state means that the system properly allocates CPU resources to move packets all the way through

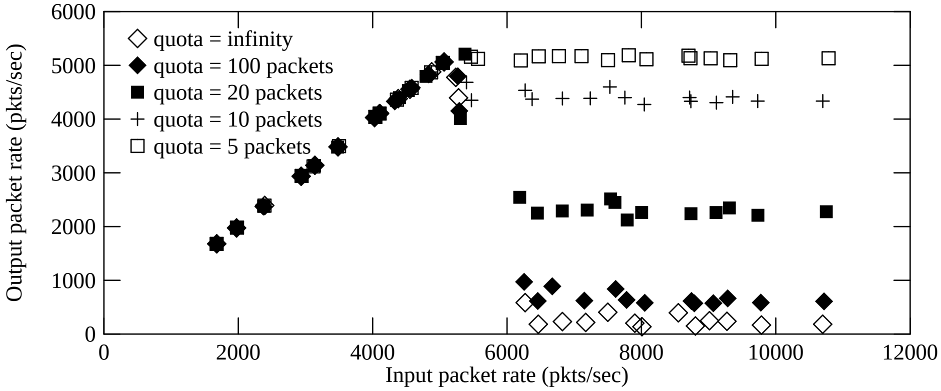

Fig. 7. Effect of packet-count quota on performance, no *screend*.

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 reenabled 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 reenabled, 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) [Mogul 1990; Mogul et al. 1987], 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. Because 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 7 shows the results; smaller quotas work better. As the quota increases, livelock becomes more of a problem.

Since the optimal quota setting probably varies depending on the specific hardware configuration, we took a closer look at the sensitivity of the non-*screend* results to the quota setting. Figure 8 shows that, as we varied the quota from 2 to 20 packets, both the peak forwarding rate (over all input rates tested) and the asymptotic forwarding rate (for the highest input rate tested) reach their maxima at a quota of 8 packets. Setting the quota above this point quickly leads to livelock, as the input stream monopolizes the CPU. Setting the quota below this point avoids livelock, but does reduce peak performance, because the polling overhead is amortized over a smaller number of packets. Figure 8 suggests that, when setting the quota, one should err toward high values if peak performance is the primary goal, but toward low values if avoiding livelock is more important.

When *screend* is used, however, the queue-state feedback mechanism prevents

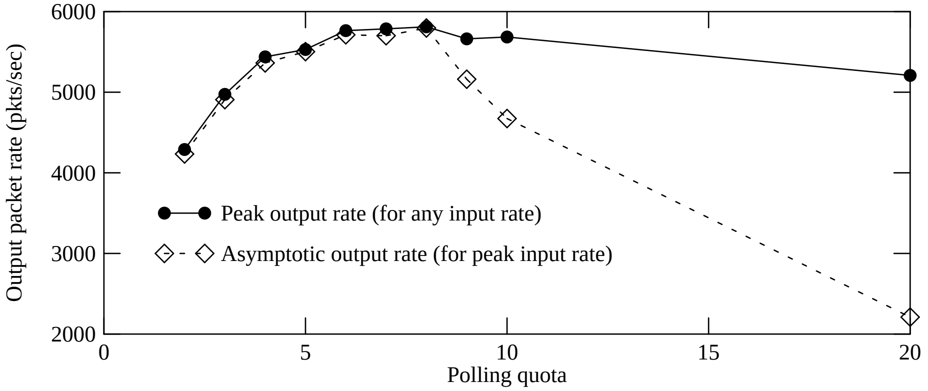

Fig. 8. Sensitivity of packet-count quota, no *screend*.

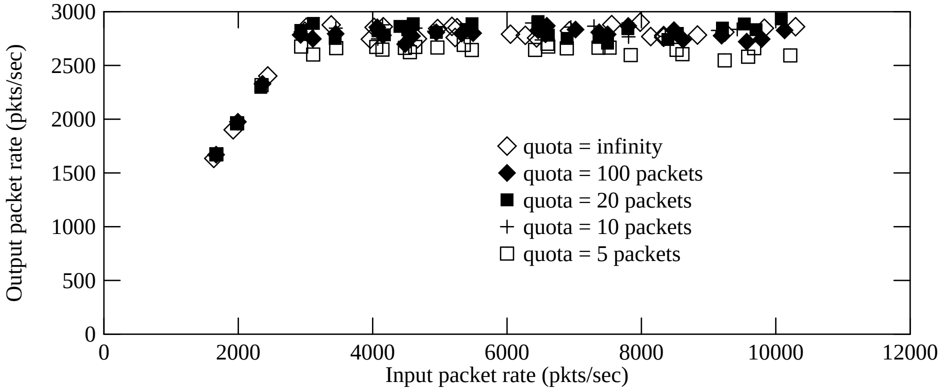

Fig. 9. Effect of packet-count quota on performance, with *screend*.

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 9 shows the results when the *screend* process is in use.

In summary, tests both with and without *screend* suggest that a quota of about seven or eight 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. Alternatively, it might be possible to use a feedback-based control system to dynamically set the quota to a point just below where livelock sets in.

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

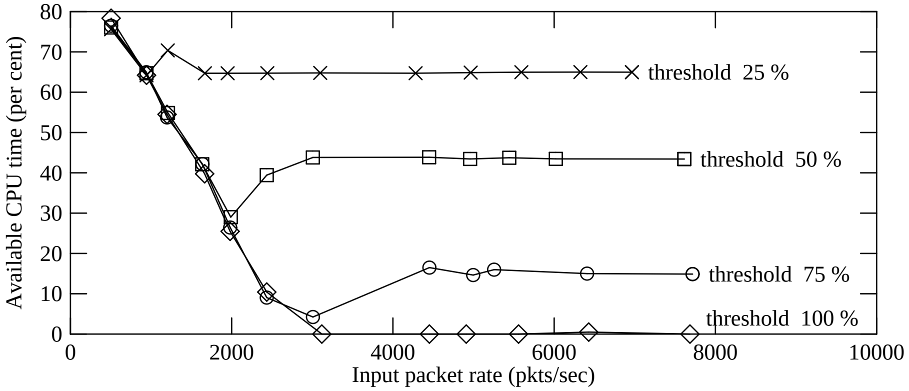

Fig. 10. User-mode CPU time available using cycle-limit mechanism.

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 architectures support similar counters.

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 Alpha's cycle counter register, 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 reenables input handling. Execution of the system's idle thread also reenables 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 10 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; the other 6% probably goes to system background processes and some kernel overheads. 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 five packets). With the system forwarding about 5000 packets/sec., handling a burst of five packets 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 so the usage-limiting mechanism fails to realize that user-level code is being short-changed. At input rates below saturation, each incoming packet may be handled fast enough that no interrupt batching occurs, and so a significant number of cycles are spent in the interrupt-handling path. As the input rate increases, the receiving interface spends less time with interrupts enabled, and so fewer cycles are spent in the interrupt-handling code.

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 Jacobson [1990] specifically to improve TCP performance. 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 acknowledgments and NFS RPCs). While we adopted this technique to help avoid livelock, we also obtain its benefits in improved transport performance.

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 [Macklem 1991].) 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.

Although TCP processing can be done in the same thread as the lower levels of the stack, NFS server implementations require separate threads of control, because a server may block waiting for disk I/O. Traditional NFS implementations, especially on servers, suffered badly from livelock because during overload the NFS threads may never get a chance to run. With queue-state feedback from the NFS server's input queue, however, we should be able to avoid much of this problem. One could also use the CPU cycle-limit mechanism to reserve some resources for the NFS threads, although it might be difficult to find the ideal allocation.

8. MEASUREMENTS USING TRACES OF KERNEL EXECUTION

Although measurements showing the performance of a system under various loads enable one to compare the ultimate benefits of several approaches, these numbers do not provide a deep understanding of the internal behavior of an operating system. We obtained traces of kernel execution to discover how the kernel is spending its time, and to measure the latency for several paths.

These traces expose, in detail, how our modifications affect the delivery latency of single packets (Section 8.1) and of bursts of several packets (Section 8.2). It would be quite difficult to measure these latencies without such traces, since they affect aspects of system behavior that are not directly visible to user-mode software.

The traces show that our modifications do indeed improve the delivery latency both for single packets, and for the first packet of a burst, and verify that the modifications do not add overhead to the packet delivery path. The traces also reveal a subtle design problem, which slightly delays the delivery of later packets in the same burst. Because the traces can show the kernel behavior in great detail, they also give specific guidance on how to prevent the additional delay.

To obtain these traces, we used ATOM, an extremely flexible mechanism for instrumenting software [Chen and Eustace 1995; Eustace and Srivastava 1995; Srivastava and Eustace 1994]. ATOM takes a fully linked binary program (even a Digital UNIX kernel) as input, and produces an instrumented binary as output. One also supplies to ATOM a module describing which points in the code to instrument, and a module containing analysis routines to execute at run-time.

Because ATOM allows the insertion of instrumentation at carefully chosen points in the kernel, it is possible to trace kernel paths without adding much overhead at all. We did the traces on a DECstation 3000/300, a relatively slow Alpha system. On this system, tracing appeared to add about 1.5 μsec. per call or return. We did some trials in which almost all kernel procedures were traced, and others in which only a few were traced. The former trials provided insight into the precise code paths involved; the latter trials allowed us to obtain relatively accurate timing information.

8.1 Traces of Single-Packet Activity

We started by instrumenting almost all kernel procedures, except for a few low-level procedures that ATOM cannot currently trace and a small set of short but frequently invoked auxiliary procedures. Tables I and II briefly describe the procedures that appear in these traces.

We ran traces while using the system to forward a single minimum-length IP/UDP packet and extracted the relevant sequence of events. We could then plot these as

Table I. Description of Important Procedures Shown in Timeline Traces

Procedure	**Description**
Thread Scheduling	
thread_wakeup_prim	Used by thread scheduler to unblock a waiting thread.
thread_run	Switches between running threads.
assert_wait_mesg_head	Used by a thread to block on an event.
netisr_input	Notifies scheduler that the network software interrupt service routine should be running.
Polling Facility	
lanpoll_isr*	Handler for software interrupt; polls devices with service requirements.
lanpoll_intsched*	Informs polling facility that an interrupt requires service.
LANCE (Ethernet) Driver	
lnintr	Interrupt entry point for LANCE driver.
lnrint, lntint	Original (non-polling) receiver and transmitter interrupt service functions.
lnrintpoll*, lntintpoll*	New (for polling) receiver and transmitter interrupt service functions.
lnintena*	Called to reenable LANCE interrupts.
lnread	Converts received packet buffer to mbuf chain.
lnoutput, lnstart	Initiates packet transmission.
lnput	Converts outgoing mbuf chain to packet buffer.
Ethernet Layer	
ether_input	Parses MAC-level header of received packet.
ether_output	Adds MAC-level header to outgoing packet.
IP Layer	
ipintr	Software interrupt handler for IP packet input.
ipinput	Parses IP header and dispatches received IP packet.
ip_output	Creates IP header for outgoing packet.
ip_forward	Forwards packets when host acts as a router.
Clock Interrupts	
hardclock, clock_tick	Periodic (1024Hz) clock interrupt handler

*New routines added to support polling.

timelines showing how procedure calls and returns nest, using the relative "stack level" to display the nesting. (Where the actual call stack includes uninstrumented procedures, the plotted stack level does not include calls through these procedures.)

Figure 11 shows a timeline for the modified kernel with polling disabled, which should approximate the behavior of an unmodified kernel. Figure 12 shows a timeline for the kernel with polling enabled. Each call is marked with the name of the procedure and the time at which the call was made, in microseconds since the start of the timeline. Returns are not individually marked, but one may deduce them from the decreases in stack level. Interrupts appear as if they were normal procedure calls.

In each case, we ran a rapid series of trials and selected one timeline in which no clock interrupts appear. To reduce the effects of cache misses, we never selected the first trial of a series. Even so, the times shown in these timelines should be treated as illustrative, but not necessarily typical. Also remember that instrumentation overhead adds several hundred microseconds to the total elapsed time (about 1.5 μsec. for each instrumented call or return). Finally, note that these tests were performed on an unusually slow implementation of the Alpha architecture.

In Figure 11, with polling disabled, we see the following interesting events (marked

Table II. Description of Boring Procedures Shown in Timeline Traces

Procedure	Description
thread_setrun, thread_continue, thread_block, switch_context, pmap_activate, get_thread_high	Thread scheduling and memory management
malloc, free	Memory allocation
m_leadingspace, m_freem, m_free, m_copym	Mbuf manipulation
lninitdesc, lnget	LANCE (Ethernet) driver
arpresolve_local	ARP layer
ip_forwardscreen, in_canforward, in_broadcast, gw_forwardscreen	IP layer
bzero, bcopy	Bulk memory operations

with dots on the timelines):

0 μsec. A packet has arrived, and lnintr() is called to begin handling the interrupt from the receiving LANCE Ethernet chip. (Several microseconds have passed between interrupt assertion and the invocation of lnintr().)

19 μsec. lnrint() is called to handle a received-packet interrupt.

29 μsec. lnrint() calls lnread() to begin packet processing, which includes copying the packet to an mbuf structure.

77 μsec. lnread() calls ether_input() to queue the received packet on the ipintr queue; ether_input() then calls netisr_input() to schedule a software interrupt.

142 μsec. lnintr() finishes its execution at device IPL.

191 μsec. After some thread-switching, ipinput() is invoked as a software interrupt.

264 μsec. The IP-layer processing has determined that this packet should be forwarded, has chosen a next-hop destination, and now calls ip_output() to send the packet along.

327 μsec. The LANCE driver has decided to send the packet, and calls lnput() to hand the buffer chain to the device.

444 μsec. IP-layer processing is complete, and the software interrupt handler exits.

522 μsec. The packet has been transmitted and the output interface has interrupted, causing a call to lnintr().

544 μsec. lntint() is called to handle the transmit interrupt.

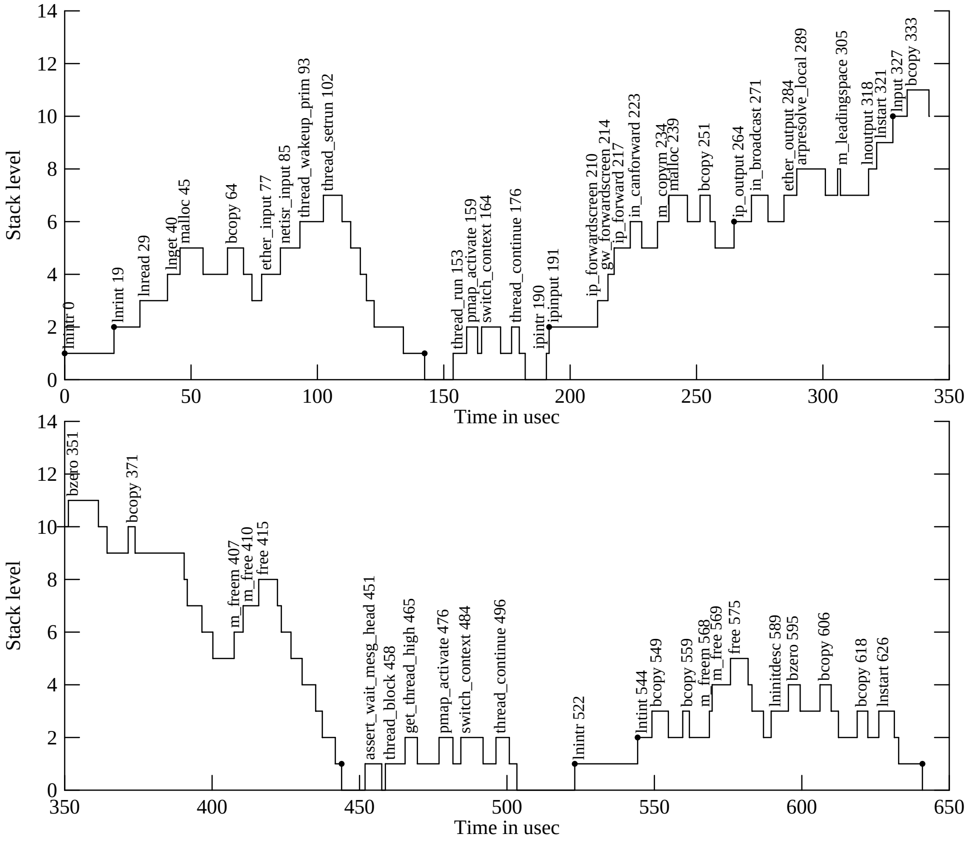

Fig. 11. Timeline forwarding a single packet, polling disabled.

633 μsec. lntint() exits, completing all activity related to this packet.

In Figure 12, with polling enabled, we see a slightly different sequence of events:

0 μsec. A packet has arrived, and again lnintr() is called to begin handling the interrupt from the receiving LANCE chip.

21 μsec. lanpoll_intsched() is called to schedule a poll for this event.

53 μsec. lnintr() finishes its execution at device IPL. At this point, interrupts from this interface are still disabled, and the CPU is entirely under the control of the polling mechanism.

97 μsec. After some thread-switching, lanpoll_isr() is called as a software interrupt handler, and begins its polling loop.

112 μsec. lnread() is called from lnrintpoll().

160 μsec. ether_input() determines that this is an IP packet, and does *not* place it on a queue.

166 μsec. ipinput() is called directly from ether_input().

235 μsec. The IP-layer processing calls ip_output() to send the packet along.

294 μsec. The LANCE driver calls lnput() to hand the buffer chain to the device.

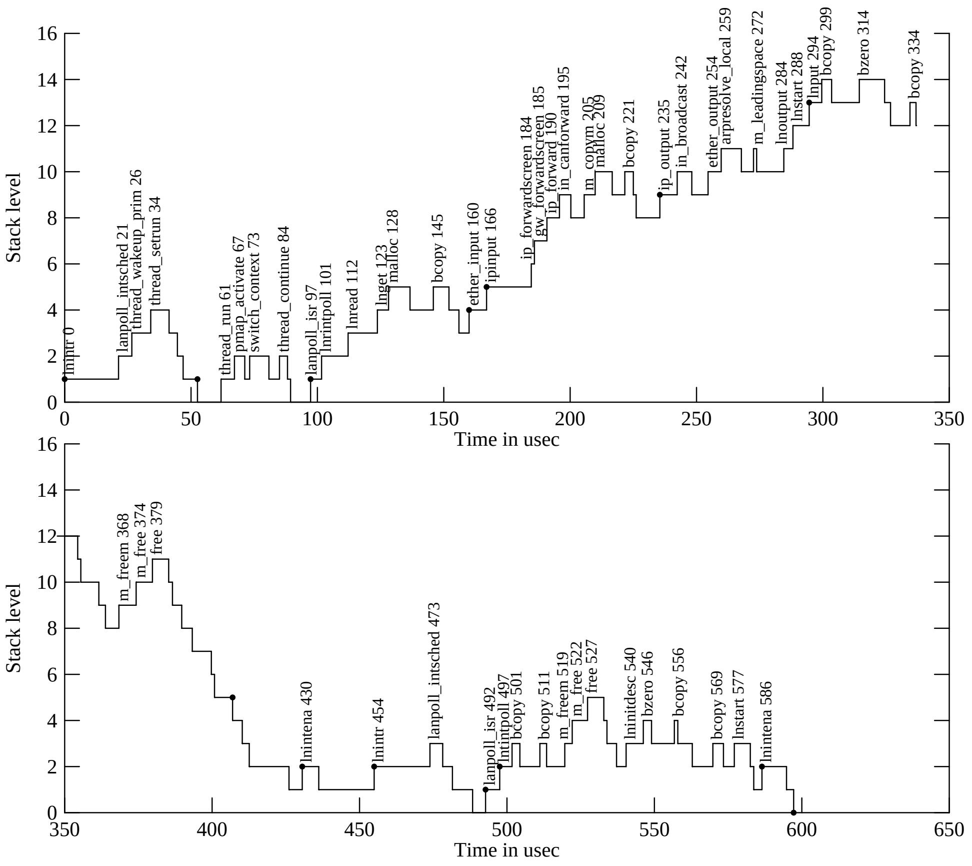

Fig. 12. Timeline forwarding a single packet, polling enabled.

407 μsec. IP-layer processing is complete, and control returns to the polling loop.

430 μsec. lanpoll_isr() calls lnintena() to reenable interrupts from this device.

454 μsec. The packet has been transmitted and the output interface has interrupted, causing a call to lnintr(), which requests service for this event.

492 μsec. lanpoll_isr() is called without any thread-switching overhead, since this is still the current thread.

544 μsec. lntintpoll() is called to handle the transmit event.

586 μsec. lanpoll_isr() calls lnintena() to reenable interrupts from this device.

597 μsec. lanpoll_isr() exits, completing all activity related to this packet.

From Figures 11 and 12, one might conclude that with polling enabled, the kernel saves about 30 μsec., mostly between the initial interrupt and the invocation of ipinput(). It is dangerous to base timing comparisons on a single pair of traces, and the instrumentation overhead confuses the situation somewhat. Therefore, we built another kernel just instrumenting the calls to lnintr() and ipinput(), and then ran a series of trials in order to obtain a statistically useful sample of the latency

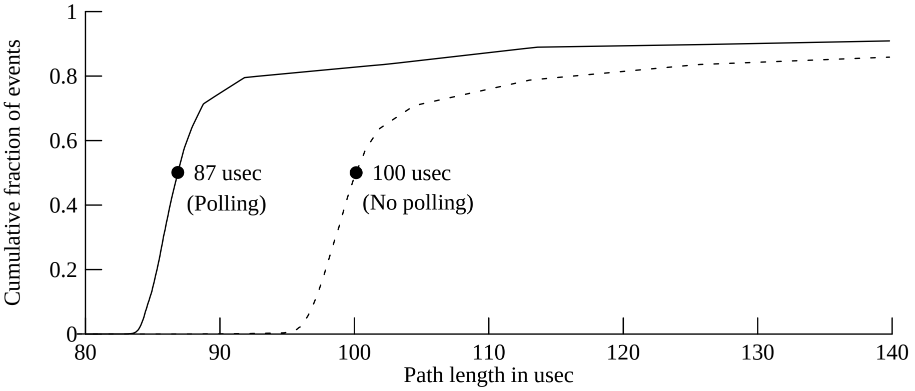

Fig. 13. Distribution of latencies from lnintr() to ipinput().

between these two points in the code. Each trial resulted in at least 10,000 packet receptions, almost all of which were short ICMP Echo packets.

The resulting distributions are shown in Figure 13. The medians are marked with dots; the use of polling seems to reduce the median latency by about 13 μsec. Polling also seems to reduce the latency variance somewhat. Note that the non-polling case includes the instrumentation overhead for one procedure return (from lnintr()) that is not included in the other case. The non-polling kernel also includes an extra "if" statement and an extra procedure call that were not present in the unmodified kernel, but these should not account for much time.

In summary, we believe that the polling kernel, on this hardware, avoids about 10 μsec. of work per packet, probably because it does not move each packet onto and off of the ipintr queue. This is reassuring, since it relieves the concern that the polling mechanism could actually add overhead to the packet reception path.

8.2 Traces of Packet Bursts

In Section 4.3, we discussed how the unmodified kernel added extra latency to the processing of packets received in bursts. Figure 1 showed measurements of this effect on an ULTRIX kernel. With the ATOM tools, we can repeat this kind of measurement using our current test system. However, since ATOM cannot directly measure the assertion of the hardware interrupt signal, we do not include the kernel's initial interrupt latency. We believe this missing time amounts to less than 10 microseconds.

Figure 14 shows traced timelines for the both the polling and non-polling kernels forwarding a burst of three short packets. To reduce the clutter in this trace, we instrumented only the most interesting procedures, and omitted most of the procedure labels.

Both traces start with a call to lnintr(). In the non-polling trace (dotted line), the first packet is delivered to the output interface by lnput() 462 μsec. later (marked with a gray square). In the polling trace (solid line), the first call to lnput() occurs after just 366 μsec. (solid square), or almost 100 μsec. sooner. Some part of this (perhaps 30 microseconds) comes from additional instrumentation overhead for the non-polling kernel. The difference may also be affected, in either direction, by some

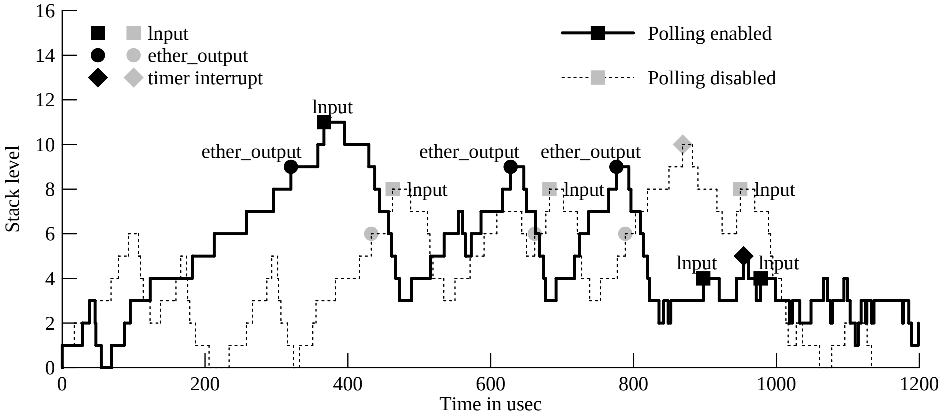

Fig. 14. Three-packet burst latency.

variation in the number of cache misses, although the trials were run repeatedly in order to warm up the caches. We also carefully selected traces that included as few clock interrupts as possible; each trace in Figure 14 contains one clock interrupt, marked with a diamond. After accounting for these sources of potential error, we still see a significantly lower initial forwarding latency in the polling kernel.

We expect, as in the experiments described in Section 4.3, that the first-packet delivery latency for the non-polling kernel will increase with burst size; the first-packet latency for the polling kernel should be independent of burst size.

While the polling kernel delivers the first packet much sooner, the non-polling kernel manages to deliver the second and third packets at 682 μsec. and 949 μsec., respectively, while the polling kernel waits to deliver its second and third packets until 898 and 978 μsec.

The cause for this discrepancy comes from an assumption in the 4.2BSD design that until the transmitter interrupts, no new packets should be added to its output buffer. Until the interrupt is serviced, the interface is marked "active," which causes the upper-layer code to leave it alone. The non-polling kernel services the first transmitter interrupt (at 590 μsec.) immediately, which allows it to restart the transmitter as soon as the second packet is ready for output. The polling kernel receives the transmit interrupt sooner (at 534 μsec.) but because the polling thread is still busy with the pending input packets, it fails to service the interrupt and so leaves the interface marked "active."

If we were to change the code to queue additional output packets even while the interface was active, the polling kernel would deliver those packets shortly after where it now calls ether_output(), at 628 μsec. and 775 μsec., respectively (solid circles). The corresponding points on the non-polling trace are marked with gray circles; they come at essentially the same relative times, after correcting for tracing overhead.

We believe that this simple change would therefore eliminate the excess latency, although it would add some per-packet overhead. The primary cause of this overhead is that starting packet transmissions more often could reduce the mean number of packets sent in one invocation of the driver's output code, which decreases the chances for amortizing the costs of device interaction over several packets. Device interaction is expensive, in part because it requires non-cachable loads and stores. Another result of starting transmissions more often is that, when the output link is not saturated, this could increase the rate of transmitter interrupts (it should not affect the number of interrupts when the output link is saturated). These overheads could reduce the peak forwarding rate of the system, although without leading to livelock. It may be necessary to choose between optimizing throughput and optimizing latency.

The polling kernel also spends a little more time handling transmitter interrupts than the non-polling kernel, because after all three packets of the burst have been fully processed, the polling kernel still schedules the polling mechanism to see if anything else needs to be done. This does not add extra overhead during conditions of input overload, because then the polling mechanism would have useful work to do. However, at lower input rates it does rob cycles from other system tasks. We believe that there are several possible solutions to this problem, including different interrupt-generation schemes in the interface hardware, or some form of "clocked interrupts" [Smith and Traw 1993; Traw and Smith 1993].

8.2.1 Implications for Other Applications. Although our changes improve end-to-end latency for the first packet in a burst of forwarded packets, packet forwarding is an unusual application because almost all of the work can be done without blocking. Most other applications, whether in-kernel (such as NFS service), or user-mode, require received packets to be queued for processing by another thread. Do our changes improve latency for these applications?

We note that as long as the polling thread has complete control of the CPU resources, nothing else can happen. (Kernel threads in Digital UNIX are preemptable with fixed priorities, and the polling thread runs at a priority above those of all consumers of network packets.) In particular, no other thread can start processing the first packet of a burst. We can see two ways to avoid this problem:

—Multiprocessing: if the number of polling threads is smaller than the number of CPUs, at least some CPU resources will be available to finish processing early packets while the polling thread (or threads) continues to receive later packets.

—CPU-time limits: in Section 7 we described how our polling mechanism can set a limit on the fraction of CPU time spent in the polling thread. We implemented this by disabling polling after the thread has used m msec. out of an n-msec. period. This has the side-effect of limiting first-packet latency (as seen by the next consumer after the polling thread) to approximately m msec., unless the burst starts while polling is inhibited because of overload.

Neither of these completely solves the problem, but at least the polling mechanism provides these partial solutions; in a purely interrupt-driven kernel, user-mode application might have to wait for an unbounded interval to receive the first packet.

9. AVOIDING LIVELOCK IN A PROMISCUOUS NETWORK MONITOR

LAN-monitoring applications typically require the host computer to place its network interface(s) into "promiscuous mode," receiving all packets on the LAN, not just those addressed to the host itself. While a modern workstation can easily handle the full large-packet data rate of a high-speed LAN, if the LAN is flooded with small packets, even fast hosts might not keep up. For example, an FDDI LAN can carry up to 227,000 packets per second. At that rate, a host has about 4.4 μsec. to process each packet.

The *tcpdump* application obtains packets from the kernel using the packet filter pseudo-device driver [Mogul et al. 1987]. Packet filtering (the selection of which received packets to hand to *tcpdump*) is done in the received-packet interrupt handler thread, and the resulting packets are put on a queue for delivery to the application. During overload conditions, the kernel discards packets because the application has no chance to drain this queue.

While the behavior of a system running *tcpdump* is fairly similar to that of one running *screend*, we include a brief discussion of how our modifications affect *tcpdump* because we have found that livelock is more likely to arise in this application than for packet routing. Many sources of packet flows, especially TCP senders, do react to packets lost at a congested router by reducing their transmission rate. But a network monitor is passive by definition, and so we cannot expect packet sources to slow down when a passive monitor is overloaded. In fact, one of the most important uses of a network monitor is to locate the cause of an anomalous network overload; a livelocked monitor would be worthless here.

Note that *tcpdump* normally only looks at the packet headers, and so requests just the first 68 bytes of each packet. Since the kernel does not touch the remaining bytes of the packet, *tcpdump* throughput is nearly independent of packet size; its speed depends on per-packet, not per-byte, overheads.

We modified our polling kernel to implement queue-state feedback (see Section 6.4) from the packet filter queue. For the measurements described here, the queue can contain at most 32 packets. Whenever the queue has room for fewer than eight additional packets, we disable polling. We also set a one-millisecond timeout, after which polling is reenabled. There is no direct mechanism to reenable polling when more queue space has been made available, although perhaps there should be.

We then tested the network-monitoring performance of the modified kernel. We set up a relatively slow system (a DECstation 3000/300) as the network monitor, and a faster system (a DECstation 3000/400) as a packet generator. The generator sent streams of packets to a third host on the same Ethernet, and *tcpdump* on the network monitor attempted to capture all of them, filtering on the UDP port number. For each trial, we sent between 10,000 and 30,000 packets at various rates, and measured the number of packets received by the monitor.

In all of these trials, we found that very few packets were dropped at the receiving interface. This means that almost all losses happened because the packet filter queue was full, not because the kernel failed to service the interface rapidly enough.

Figure 15 shows the results for all of our *tcpdump* trials.

In our first set of trials, we configured *tcpdump* to simply copy the packet headers to the null device, `/dev/null`, instead of regular disk file. This should reduce the per-packet overhead and so increase the MLFRR.

The use of queue-state feedback clearly results in much better peak packet capture performance. It is tempting to infer that the use of feedback also improves overload behavior, but the rates measured in the `/dev/null` trials do not quite reach the saturation point, and so provide no direct evidence about performance above that

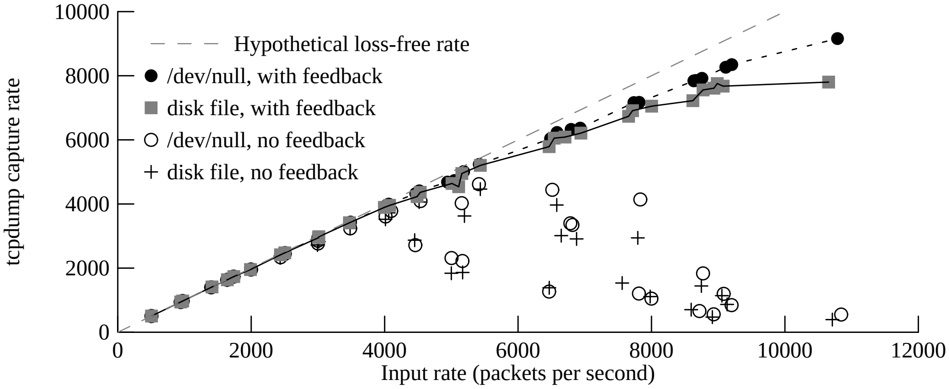

Fig. 15. *tcpdump* capture rate, all trials shown.

rate.

Note that at rates above the MLFRR of the no-feedback system, even with queue-state feedback the system does lose some packets. (The gray dashed line shows the performance of a hypothetical loss-free system.) We attribute this to the necessarily finite size of the packet filter queue: even though queue-state feedback inhibits input processing when the queue becomes full, the one-millisecond timeout happens before *tcpdump* has drained much of the queue, and so the kernel has no place to put the next batch of packets.

The results for the no-feedback kernel show a noisy relationship between input and output rates, above the MLFRR. This is because the packet generator is a relatively bursty source, and the mean burst size changes for different long-term generated rates. When the mean burst size is large, the network monitor processes more packets per interrupt, thus using fewer CPU cycles and leaving more for the *tcpdump* application.

We also ran trials with *tcpdump* writing the received packet headers to a disk file. This added just enough per-packet overhead to allow us to saturate the system, even with queue-state feedback, at an input rate of about 9000 packets/sec. (and a capture rate of about 7700 packets/sec.). Below the saturation point, the extra overhead of writing headers to the disk has only a small effect on capture rate.

10. 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 [Smith and Traw 1993; Traw and Smith 1993] 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 [Smith and Traw 1993] 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 [Leffler et al. 1989] 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 [Floyd and Jacobson 1993].

Fall [1994] discusses the problem of overload in non-flow-controlled systems such as routers. His approach improves system behavior by reducing per-packet and per-byte overheads, and thus increases the MLFRR, but does not directly improve overload behavior. His approach complements ours, but does not solve the livelock problem.

Druschel and Banga's "Lazy Receiver Processing" (LRP) [Druschel and Banga 1996] work is a more radical approach, which includes among its goals a partial solution to the livelock problem. LRP depends on early demultiplexing of received packets, which can either be done in software (SOFT-LRP), or in a special network interface (NI-LRP). NI-LRP fully solves the livelock problem, by explicitly shedding excess load in the network interface, whereas our approach implicitly sheds excess load by simply not servicing the interface. NI-LRP therefore can select which packets to drop, based on higher-level considerations; our approach is oblivious to the value of any given packet. SOFT-LRP postpones livelock, by improving system efficiency, but does not prevent it. It may be possible to combine the SOFT-LRP design with some of our techniques, thereby achieving both high performance and preventing livelock, without requiring special network interface implementations.

Mosberger and Peterson [1996] describe an even more radical approach in the Scout operating system, which uses the concept of a "path" as its basic design principle. Paths, as used in Scout, are a more formal version of the LRP approach, relying on early packet classification to assign high-level priorities to packets, immediately upon reception. This gives Scout the ability to shed load selectively; it is not clear if Scout currently prevents livelock in all situations. Scout, LRP, and our work all share the premise that if a packet is going to be dropped, it is better to drop it as early as possible.

Massalin and Pu [1990] describe a feedback-based scheduling system, used in the Synthesis Kernel, which adjusts the amount of CPU time allocated to a thread, based on queue lengths. This scheduler attempts to avoid congestion by allocating more time to threads with full input queues, and less time to threads with full output queues. Our approach is vaguely similar, although cruder: we simply shut off the input to the polling system when any of its output queues becomes too full, which implicitly stops it from consuming CPU time.

Some of our initial work on improved interface driver algorithms is described in Chang et al. [1993].

Some of the work presented in this article was first done in the context of Calaveras, an advanced development project at Digital [Ramakrishnan et al. 1995; Vahalia et al. 1995]. The Calaveras kernel was designed to provide a foundation for building a high-bandwidth distributed file server capable of supporting traditional data as well as continuous media; in particular, it was used for a video-on-demand server. The Calaveras scheduler supported three classes of tasks, including isochronous tasks (primarily for video and audio streams), real-time tasks with weights for allocating CPU resources, and general-purpose tasks. Interrupt handlers did no work except to set a flag indicating that a device needed service; the rest of the job of servicing an I/O event was then performed by a real-time task, invoked by a polling thread. Livelock was thus avoided, as long as the interrupt interarrival time was longer than the time required to handle each interrupt and set the flag. This lightweight-handler design also reduced interrupt overhead, because very little CPU state had to be saved on an interrupt. Measurements of the Calaveras prototype showed that it did succeed in avoiding livelock.

11. FUTURE WORK

Although the implementation described in this article is straightforward and robust, and earlier versions have been deployed to customer installations, we see several areas that may require additional research.

11.1 Faster Implementations

The experiments reported on in this article were done on a relatively slow LAN (Ethernet, at 10 Mbits/sec.) and on the slowest available CPU that would run the Digital UNIX operating system. This allowed us to investigate the performance regime at and near overload, but the results cannot be extrapolated to predict the performance of state-of-the-art LANs and CPUs.

Use of faster CPUs in itself would be no problem; our kernel will run without modification on the fastest Alpha system. However, we would be unable to saturate such a system with our existing Ethernet-based test setup. To do high-speed experiments, we would have to obtain or construct a much faster packet generator. We would also have to apply our modifications to a driver for a faster LAN technology, such as FDDI or ATM. Unfortunately, such drivers seem to be far more complex than those for Ethernet interfaces. (Fast Ethernet would provide a suitable LAN speed, but no Fast Ethernet interface is available for the ancient Alpha workstation we used in these experiments.)

Memory-system performance has not usually improved quite as rapidly as the peak CPU instruction issue rate. This implies that future processors will be less sensitive to instruction counts, and more sensitive to memory locality. We expect that this will change the relative performance advantages of our kernel modifications, but we cannot predict which direction these changes will go.

11.2 Extension to Multiprocessor Kernels

Most computer vendors now sell some form of multiprocessor, to increase the performance of high-end systems beyond what is possible with a single CPU. Symmetric Multiprocessing (SMP) systems have been quite successful in many applications, and typically run a traditional operating system kernel that has been modified to support multiple kernel threads.

Although Digital UNIX is an SMP kernel, we have not yet extended our work to a true SMP environment. Our current kernel could run on SMP hardware, but it would do all of the interface driver and network processing on just one of the processors.

We believe that our polling approach allows better parallelization of interface and network processing, and so should improve performance on SMP systems. In order to demonstrate this, we would have to extend our modifications to include multiple polling threads, and we may have to modify the interface drivers to support concurrency in what were originally interrupt service routines (which are not run concurrently in the original system).

The use of multiple polling threads, while providing the opportunity for parallelization, also presents some challenges. How many threads should be active at once? If CPU cycle limits are enabled, should they apply individually to each CPU, or to the entire system? And will the extra overhead of locking defeat the purpose of parallelization?

11.3 Selective Packet Dropping

Our approach to input overload is to drop packets as early as possible, to avoid spending resources on packets that would be dropped later on. This is a policy about when to drop packets, not about which packets to drop. In many cases, some packets may be much more important than others, and the system would be more effective if it preserved those when possible.

For example, when video streams are sent using Motion-JPEG compression, individual frames are independent of each other. A frame may be made up of several packets, all of which must be received for successful decompression. If one must discard several packets within a short interval, it is better to discard all of the packets of one frame rather than spreading the discarded packets among multiple frames, which then might all be impossible to decompress. Similarly, Romanow and Floyd [1995] have pointed out that if an ATM switch must drop multiple cells, it is better to drop all the cells of one packet rather than to spread the cell loss evenly over many packets.

Fall et al. [1995] have proposed *early discard load shedding* for Motion-JPEG streams, a technique in which frames are discarded as early in the processing pipeline as possible. Their goal is to shed load before the system becomes overloaded, while we have focused on responding to overload, but in either case it seems useful to be able to identify specific classes of packets to drop.

When received packets are dropped late, and therefore at upper levels in the network stack, it is fairly easy to determine which ones are useful and which ones are expendable. In our approach, because we have tried to drop packets as early as possible, perhaps before the software even sees them, it could be quite hard to drop all of the packets from one Motion-JPEG frame instead of spreading the losses at random.

Note that labeling techniques such as virtual circuits, “flows,” and packet priorities do not help here: from the sender’s point of view, each Motion-JPEG frame has the same priority. One cannot *a priori* say that a given packet is more important than another; only when one is forced to drop a packet does it then become possible to define other packets as good targets for dropping.

11.4 Interactions with Application-Layer Scheduling

We see several connections between scheduling the processing of received packets, and scheduling of application (user-mode) processes. At input rates below the ML-FRR, it may be appropriate to modify the order in which packets are processed so as to reduce latency for packets destined to currently running processes, or to provide batching across the kernel-user boundary (and so reduce context switching). At higher input rates, where some packets must be discarded, the packet-discard policy might favor packets for currently scheduled processes; this could avoid thrashing. In either case, the process scheduler might prefer to give time to a process that has a large queue of pending packets.

Waldspurger has developed a proportional-share framework for expressing and implementing application scheduling policies [Waldspurger 1995; Waldspurger and Weihl 1994; 1995]. In this framework, resource shares can be expressed in a kind of currency. He suggests (personal communication, 1995) that if arriving packets, by the use of some simple labeling scheme, carry tokens in such a currency, it might be possible to integrate the packet-level scheduling decisions with process-level scheduling.

For example, suppose that packets were simply marked with the ID of the receiving process. As packets arrive for various processes, the kernel could track the number of unreceived packets per blocked process, and unblock the process with the largest batch of work to perform. If the system becomes overloaded, the network thread might start discarding all packets except for those destined to the current process.

Or suppose we want to drop excess packets according to some predetermined allocation of resources to processes. The network processing thread could keep track of how many packets have been received for each process during a recent interval, and discard packets in such a way as to maintain packet-consumption rates in proportion to the predetermined resource allocations.

We believe that such mechanisms could be implemented with minimal overhead, but we do not yet know how useful they would be.

12. 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 article, 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
—Carefully switching between polling and interrupts
—Processing received packets to completion
—Explicitly regulating CPU usage for packet processing
—Using feedback to inhibit input that would be discarded

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.

ACKNOWLEDGMENTS

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, Kevin Fall, Hal Murray, Carl Waldspurger, and Jeffry Yaplee helped during manuscript preparation. Marc Viredaz helped us work around a bug in the ACM TOPLAS Latex style file.

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

REFERENCES

Chang, C.-H., Flower, R., Forecast, J., Gray, H., Hawe, W. R., Nadkarni, A. P., Ramakrishnan, K. K., Shikarpur, U. N., and Wilde, K. M. 1993. High-performance TCP/IP and UDP/IP networking in DEC OSF/1 for Alpha AXP. *Digital Tech. J. 5,* 1 (Winter), 44–61.

Chen, J. B. and Eustace, A. 1995. Kernel instrumentati¡on tools and techniques. Tech. Rep. TR-26-95, Harvard Univ. Center for Research in Computing Technology, Cambridge, Mass. Nov.

Druschel, P. and Banga, G. 1996. Lazy Receiver Processing (LRP): A network subsystem architecture for server systems. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation.* USENIX Assoc., Berkeley, Calif., 261–275.

Eustace, A. and Srivastava, A. 1995. ATOM: A flexible interface for building high performance program analysis tools. In *Proceedings of the 1995 USENIX Conference.* USENIX Assoc., Berkeley, Calif., 303–313.

Fall, K. 1994. A peer-to-peer I/O system in support of I/O intensive workloads. Ph.D. thesis, Univ. of California, San Diego.

Fall, K., Pasquale, J., and McCanne, S. 1995. Workstation video playback performance with competitive process load. In *Proceedings of the 5th International Workshop on Network and Operating Systems Support for Digital Audio and Video.* IEEE Communications Society, New York, 179–182.

Ferrari, D., Pasquale, J., and Polyzos, G. C. 1991. Network issues for Sequoia 2000. Sequoia 2000 Tech. Rep. 91/6, Univ. of California, Berkeley. Dec.

Floyd, S. and Jacobson, V. 1993. Random early detection gateways for congestion avoidance. *Trans. Networking 1,* 4 (Aug.), 397–413.

Jacobson, V. 1990. Efficient protocol implementation. In bound notes provided at ACM SIGCOMM '90 Tutorial on "Protocols for High-Speed Networks".

Leffler, S. J., McCusick, M. K., Karels, M. J., and Quarterman, J. S. 1989. *The Design and Implementation of the 4.3BSD UNIX Operating System.* Addison-Wesley, Reading, Mass.

Macklem, R. 1991. Lessons learned tuning the 4.3BSD Reno implementation of the NFS protocol. In *Proceedings of the Winter 1991 USENIX Conference.* USENIX Assoc., Berkeley, Calif., 53–64.

MASSALIN, H. AND PU, C. 1990. Fine-grain adaptive scheduling using feedback. *Comput. Syst. 3,* 1 (Winter), 139–174.

MOGUL, J. C. 1989. Simple and flexible datagram access controls for UNIX-based gateways. In *Proceedings of the Summer 1989 USENIX Conference.* USENIX Assoc., Berkeley, Calif., 203–221.

MOGUL, J. C. 1990. Efficient use of workstations for passive monitoring of local area networks. In *Proceedings of the SIGCOMM '90 Symposium on Communications Architectures and Protocols.* ACM, New York, 253–263.

MOGUL, J. C., RASHID, R. F., AND ACCETTA, M. J. 1987. The Packet Filter: An efficient mechanism for user-level network code. In *Proceedings of the 11th Symposium on Operating Systems Principles.* ACM, Austin, Texas, 39–51.

MOSBERGER, D. AND PETERSON, L. L. 1996. Making paths explicit in the scout operating system. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation.* USENIX Assoc., Berkeley, Calif., 153–167.

PERLMAN, R. 1983. Fault-tolerant broadcast of routing information. *Comput. Networks 7,* 6 (Dec.), 395–405.

RAMAKRISHNAN, K. K. 1992. Scheduling issues for interfacing to high speed networks. In *Proceedings of the Globecom '92 IEEE Global Telecommunications Conference.* IEEE, New York, 622–626.

RAMAKRISHNAN, K. K. 1993. Performance considerations in designing network interfaces. *IEEE J. Sel. Areas Commun. 11,* 2 (Feb.), 203–219.

RAMAKRISHNAN, K. K., VAITZBLIT, L., GRAY, C., VAHALIA, U., TING, D., TZELNIC, P., GLASER, S., AND DUSO, W. 1995. Operating system support for a video-on-demand file service. *Multimedia Syst. 3*, 53–65.

RANUM, M. J. AND AVOLIO, F. M. 1994. A toolkit and methods for Internet firewalls. In *Proceedings of the Summer 1994 USENIX Conference.* USENIX Assoc., Berkeley, Calif., 37–44.

ROMANOW, A. AND FLOYD, S. 1995. Dynamics of TCP traffic over ATM networks. *IEEE J. Sel. Areas Commun. 13,* 4 (May), 633–641.

SMITH, J. M. AND TRAW, C. B. S. 1993. Giving applications access to Gb/s networking. *IEEE Network 7,* 4 (July), 44–52.

SOUZA, R. J., KRISHNAKUMAR, P. G., ÖZVEREN, C. M., J.SIMCOE, R., SPINNEY, B. A., THOMAS, R. E., AND WALSH, R. J. 1994. GIGAswitch: A high-performance packet switching platform. *Digital Tech. J. 6,* 1 (Winter), 9–22.

SRIVASTAVA, A. AND EUSTACE, A. 1994. ATOM: A system for building customized program analysis tools. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation.* ACM, New York, 196–205.

TRAW, C. B. S. AND SMITH, J. M. 1993. Hardware/software organization of a high-performance ATM host interface. *IEEE J. Sel. Areas Commun. 11,* 2 (Feb.), 240–253.

VAHALIA, U., GRAY, C. G., AND TING, D. 1995. Metadata logging in an NFS server. In *Proceedings of the 1995 USENIX Conference.* USENIX Assoc., Berkeley, Calif., 265–276.

WALDSPURGER, C. A. 1995. Lottery and stride scheduling: Flexible proportional-share resource management. Tech. Rep. MIT/LCS/TR-667, Massachusetts Institute of Technology Laboratory for Computer Science, Cambridge, Mass. Sept.

WALDSPURGER, C. A. AND WEIHL, W. E. 1994. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation (OSDI).* USENIX Assoc., Berkeley, Calif., 1–11.

WALDSPURGER, C. A. AND WEIHL, W. E. 1995. Stride scheduling: Deterministic proportional-share resource management. Tech. Memo. MIT/LCS/TM-528, Massachusetts Institute of Technology Laboratory for Computer Science, Cambridge, Mass. June.

Received May 1996; revised May 1997; accepted June 1997