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

Eliminating Receive Livelock in an Interrupt-driven Kernel

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

For more information about USENIX Association contact:

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

以下论文最初发表于 USENIX 1996 年度技术会议圣地亚哥，加利福尼亚州，1996年1月

在中断驱动内核中消除接收死锁

杰弗里·莫格尔, DEC 西部研究实验室 K. K. 拉马库什南, AT&T贝尔实验室

有关USENIX协会的更多信息，请联系：

1. 电话: 510 528-8649
2. 传真: 510 548-5738
3. 电子邮件: office@usenix.org
4. WWW网址: http://www.usenix.org

Eliminating Receive Livelock in an Interrupt-driven Kernel

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

Abstract

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

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

1. Introduction

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

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

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

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

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

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

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

在中断驱动内核中消除接收死锁

杰弗里·C·莫格尔 数字设备公司 西方研究实验室 K.
K. 拉马库什南 AT&T贝尔实验室

摘要

Most operating systems use interface interrupts to
调度网络任务。中断驱动系统可以
提供低开销和良好延迟，在低负载下
的负载，但在更高的到达率下会显著降低性能
比率，除非采取预防措施以防止多种情况
病理。这些是接收的各种形式
死锁，在这种情况下，系统花费所有时间
处理中断，排除了其他必要任务
任务。在极端条件下，没有数据包
交付给用户应用程序或输出
系统。

为了避免死锁和相关问题，一个操作-
系统必须安排网络中断处理
尽可能地安排进程执行。我们
修改了一个中断驱动网络实
实现这样做；这消除了接收死锁，与
out 导致系统性能的其他方面下降。
我们展示了测量结果，证明其成功
我们方法的应用。

1. 简介

大多数操作系统使用中断来互-
迟的服务；最后安排与I/O相关的任务-
事件，尤其是网络调用
协议软件。中断是有用的，因为它们
允许CPU大部分时间做有用
处理中，但能快速响应事件而不
必须不断轮询事件到达。

轮询很昂贵，尤其是当I/O事件
相对稀少，磁盘也是如此，磁盘
很少中断超过几百次每
秒。轮询也会增加的延迟
对事件的响应。现代系统可以响应
中断，只需几十微秒；要实现
使用轮询获得相同的延迟，系统会
必须每秒轮询数万次，
将导致过高的开销。对于
通用系统，中断驱动设计
效果最佳。

大多数现有操作系统都是为
处理每隔几毫秒就会中断的I/O设备
毫秒。磁盘倾向于按顺序发出事件
每转一圈一次；第一代LAN环境
每台终端系统每秒倾向于生成几百个数据包。尽管人们

理解了降低出行成本的需求
中断，通常这种开销足够低，以至于
任何正常系统都只会花费其负载的一小部分
CPU时间处理中断。

世界已经改变。操作系统通常
通常使用相同的中断机制来控制
既进行网络处理又使用传统I/O设备，
然而，许多新应用程序可以生成数据包
比磁盘寻道频繁几个数量级
可以生成寻道。多媒体和其他实时
应用程序将成为主流。客户端-服务器
应用程序，例如NFS，在快速客户端上运行和
服务器可以生成重型RPC负载。多播和
广播协议影响无辜旁观者主机
对它们完全不感兴趣的负载。因此，
网络实现必须现在处理信号，
显著更高的事件速率。

许多多媒体和客户端-服务器应用程序
它们还共享另一个令人不快的特性：与传统网络应用程序
（Telnet、FTP、电子邮件）不同，
它们不是流控的。一些多媒体应-
用；一些应用程序需要恒定速率、低延-
基于RPC的客户端-服务器应用程序通常使用-
数据报式传输，而不是可靠的、流量控制
受控协议。注意，而I/O设备
例如磁盘，仅在中断的结果下产生
接收来自操作系统的请求，因此
天性地流控，网络接口生成
接收非请求中断时。

转向更高的事件速率和非流-
受控协议可能使主机遭受拥塞
崩溃：一旦事件速率使系统饱和，
而无需负反馈循环来控制
多个源，没有办法优雅地卸载负载。如果
在主机在这些条件下以满吞吐量运行
itions，并公平地为所有源提供服务，这
最有可能保持稳定性。但如果
吞吐量随着提供负载的增加而减少，
整体系统变得不稳定。

中断驱动系统往往表现不佳
在过载情况下。中断级别的任务，

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

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

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

2. Motivating applications

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

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

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

3. Requirements for scheduling network tasks

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

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

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

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

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

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

按定义，具有绝对优先级，高于所有其他
任务。如果事件率足够高以至于导致
系统将花费所有时间来响应中断
如果中断发生，则不会发生其他事情，系统
吞吐量将降至零。我们称这种状态为接收死锁：系统并未死锁，但它
在它的任何任务上都没有进展。

任何纯粹使用固定中断的系统中使用固定中断
中断优先级将遭受接收死锁
der input overload conditions。一旦输入速率
exceeds the reciprocal of the CPU cost of processing
一个输入事件，任何在较低
优先级将不会有机会运行。

然而，我们不想轻易放弃明显的
的中断驱动设计的优势。相反，我们
应该将网络中断控制的控制集成到
将处理子系统集成到操作系统的调度
中断机制和政策。在本文中，我们
提出了一些简单的修改，以改进调度数据包的
纯中断驱动模型，并展示它们
保证吞吐量并提高延迟在
过载，同时保留理想特性
轻负载下的中断驱动系统。

2. 动机应用程序

我们的研究是由一系列
某些应用程序可能会遭受死锁。
此类应用程序可以基于专用单机系统构建。
目的系统，但通常使用通用系统构建。
UNIX系统等目的系统，而且我们想要
找到一个通用的死锁问题解决方案。The
applications include:

- 基于主机的路由：尽管网络间
  路由通常使用特殊-
  目的（通常是非中断驱动）路由器
  系统，路由通常使用更简-
  常规主机。几乎所有互联网
  “”防火墙”“”产品使用UNIX或Windows NT™ 系统用于路由[7, 13]。在输入过载条件下，
  perimentation with new routing algorithms is
  在UNIX [2]，上完成，特别是针对IP多播-
  ing.
- 被动网络监控：网络管理员，
  开发者，研究人员通常使用
  UNIX系统，及其网络接口在
  “混杂模式”来监控局域网流量以进行调试或统计收集 [8]。
- 网络文件服务：NFS协议的服务器通常由UNIX系统构建，
  项过载时，

这些应用程序（以及其他类似的应用程序，例如
Web服务器)都可能暴露于重型、
非流量控制负载。我们遇到了
这三个应用程序中的死锁，已经解决
或缓解了问题，并且已经发布了解决方案

对客户。本文其余部分将
专注于基于主机的路由，因为这样可以简化
在问题上下文中，并允许轻松执行
性能测量。

3. 调度网络任务的要求

性能问题通常在 sys-
tem 是 subjected to transient or long-term input overload。理想情况下，通信子系统可以
handle the worst-case input load without saturating,
但成本考虑通常防止我们从构建-
构建如此强大的系统。系统通常
尺寸以支持指定的设计中心负载，和
在过载情况下，我们所能做到的最好就是受控
以及优雅降级。

当终端系统参与处理时
大量网络流量，其性能取决于
重点关注其任务如何调度。我们
机制和政策，这些机制和政策用于纯中断驱动模型，并表明它们
处理和其他任务应保证接受-
可系统吞吐量，合理延迟和抖动
(延迟方差)，资源公平分配，和
系统整体稳定性，而不会造成过度开销，尤其是在系统过载时。

我们可以将吞吐量定义为系统每秒处理的
系统将数据包传递给最终消费者。
一个消费者可能是一个在应用程序上运行的
接收主机，或者主机可以作为路由器
并将数据包转发到其他主机上的消费者。
我们期望一个设计良好的系统的吞吐量
需要跟上提供的负载，直到一个称为
最大无丢包接收率（MLFRR），和
{v1}在更高的负载下，吞吐量不应低于这个
速率。

当然，有用吞吐量不仅取决于
成功接收数据包；系统还必须
传输数据包。因为数据包接收和
数据包传输经常争夺相同的
资源，在输入过载条件下
scheduling subsystem must ensure that packet transmission continues at an adequate rate.

Many applications, such as distributed systems
和交互式多媒体，通常更依赖于
低延迟、低抖动通信，而不是高
吞吐量。即使在过载时，我们也想避免
长队列，这会增加延迟，以及突发调度，这会增加抖动。

当主机因传入网络
数据包，它也必须继续处理其他任务，
以便使系统能够对管理做出响应
控制请求，并允许应用程序
利用到达的数据包。调度
子系统必须公平分配CPU资源给
数据包接收，数据包传输，协议

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

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

4. Interrupt-driven scheduling and its consequences

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

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

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

4.1. Description of an interrupt-driven system

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

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

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

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

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

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

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

处理，其他I/O处理，系统维护
任务，并且应用程序处理。
过载时行为不良的主机可以
也损害网络上的其他系统。死锁在
一个路由器，例如，可能导致控制
消息，或延迟其处理。这可能导致
其他路由器错误地推断链路故障，导致
错误的路由信息在传播
整个广域网。更糟的是，
控制消息可以通过
导致控制生成的正反馈
处理 [10] 的流
量。

4. 中断驱动调度及其后果

调度策略和机制显著
影响系统在下的吞吐量和延迟
过载。在中断驱动操作系统中，
中断子系统必须被视为一个com-
调度系统的组成部分，因为它在决定什么代码何时运行方面起着重要作用。
在确定什么代码运行时起着重要作用。我们
观察到中断驱动系统存在
满足第3节中讨论的要求。
在本节中，我们首先描述了其特性
中断驱动系统，然后确定三个
网络输入过载引起的问题
in 中断 -驱动系统：

- 过载下的接收死锁：交付
  吞吐量降至零，同时输入超-
  负载持续存在。
- 增加数据包传输的延迟或-
  预防：系统延迟一个
  数据包，同时它正在处理子
  数据包过载，可能是一个突发。
- 数据包传输饥饿：即使
  CPU能够跟上输入负载，严格优先级分配可能防止它传输
  任何数据包。

4.1. 中断驱动系统的描述

中断驱动系统在
网络输入过载，因为其方式
它优先处理作为网络-
统输入。我们首先描述一个典型的运
序在处理和优先级排序方面执行最
网络任务。我们使用4.2BSD [5] 模型为我们的
例如，但我们观察到其他操作系统
，例如VMS™、DOS和Windows NT，
甚至几个以太网芯片，具有相似的特
特性，因此存在相似的问题。
当数据包到达时，网络接口发
nal通过中断CPU来处理此事件。设备中断
正常的中断通常具有固定的中断优先级
级别（IPL），并抢占所有低优先级运行的任务
IPL; 中断不会抢占在同一IPL上运行的
相同的IPL。中断导致进入与
关联的网络设备驱动程序，该驱动程序执行一些内
初始处理数据包。在4.2BSD中，仅缓冲
管理和数据链路层处理发生
在 ‘ ‘设备IPL。’ ’ 设备驱动程序然后将
数据包在队列中，并生成一个软件中断
导致对数据包进行进一步处理。
软件中断在较低的IPL被触发，因此这
协议处理可以被后续的
中断。(我们避免在高IPL下长时间停留，以
减少处理某些其他事件的延迟。)
在不同IPL执行的步骤之间的队列提供了一些隔离，
以防止数据包丢失
由于瞬态过载，但通常它们有
固定长度限制。当数据包应该排队
但如果队列已满，系统必须丢弃数据包。
选择合适的队列限制，以及因此的
系统各层之间的缓冲分配，是
对良好性能至关重要，但超出了
本文。
请注意，操作系统的调度器
不参与任何此类活动，实际上
完全不了解它。
由于这种结构，重型负载
传入数据包可能会产生高比率
设备IPL的中断。处理中断是一项
高成本操作，因此为了避免这种开销，网络-
工作设备驱动程序尝试批量处理中断。那
是，如果数据包以突发方式到达，中断处理程序
尝试尽可能多地处理数据包，
在从中断返回之前。这可以分摊
处理中断的成本，分摊到多个数据包上。
即使有批处理，一个系统如果被连续
将数据包放入将花费大部分时间在代码中
在设备IPL上运行。也就是说，该设计赋予
传入数据包绝对优先级。在
4.2BSD 开发的时间，在 20 世纪 80 年代初，其理由是网
络适配器缓冲内存很少，因此如果系统
ers 缓冲内存很少，因此如果系统
未能及时将接收到的数据包移动到主
内存，后续数据包可能会丢失。(这是
仍然存在低成本接口的问题。) 因此，系
作系统从 4.2BSD 派生的设备驱动程
设备IPL，并给予此处理优先级高于所有
其他网络任务。
现代网络适配器可以接收来自多个后端系统
无需主机干预即可将数据包回退，或者
通过大量缓冲或高度
自主DMA引擎。这可以隔离系统
从网络接收，并消除了其中大部分
给前几个中断赋予绝对优先级的理由
处理接收到的数据包的步骤

4.2. Receive livelock

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

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

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

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

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

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

4.3. Receive latency under overload

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

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

4.4. Starvation of transmits under overload

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

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

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

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

This effect has also been described previously [12].

5. Avoiding livelock through better scheduling

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

4.2. 接收死锁

在中断驱动系统中，接收器中断
优先于所有其他活动。如果数据包到达
太快，系统将花费所有时间处理-
接收器中断。因此，它将没有
资源剩余以支持到达的NFS RPC请求的交付，例如，这意味着
数据包到应用程序（或者，在路由器的情况下，到
转发和传输这些数据包）。使用
系统的吞吐量将降至零。

遵循 [11]，我们将此条件称为
接收死锁：系统的一种状态，其中没有进行有用进
展，因为某些必要资源被完全消耗在处理中

接收器中断。当输入负载下降时
高效地，系统离开这个状态，并且再次
能够实现正向进展。这不是一个死-
锁状态，系统将不会从这个状态恢复
即使输入速率降至零。

系统在输入负载增加时，可能以三种方式之一运行。
输入负载增加。在理想系统中，
交付吞吐量始终与提供负载匹配
负载的概率。在一个可实现的系统中，交付吞吐量
保持与提供负载同步，直到最大
无损接收率 (MLFRR)，然后是相
相对恒定。在负载高于
MLFRR，系统仍然在进步，但它
丢弃部分输入；通常，数据包
在处理步骤之间的队列中丢弃
在不同优先级发生。

在一个易发生接收死锁的系统里，然而，
吞吐量随负载增加而减少，
针对输入速率高于MLFRR的情况。接收死锁
发生在吞吐量降至零的点。
死锁系统浪费了它所付出的所有努力
进入部分处理接收到的数据包，因为它们
全部都被丢弃了。

接收器-中断批处理使情况变得复杂-
情况略有改善。通过提高系统在-
重型负载，批处理可以提高MLFRR。
批处理可以移动死锁点，但无法，通过
它本身，防止死锁。

在6.2节中，我们展示了测量结果，说明死锁如何在实
际情况下发生。Ad-
附加测量，以及更详细的调度
问题，在[11]中给出。

4.3. 过载下的接收延迟

虽然中断驱动设计通常是
被认为是一种减少延迟的方法，但它们可以确
实际上增加数据包传输的延迟。如果
数据包突发到达过于迅速，系统将在对第一个数
据包进行任何高层处理之前，先对整个突发进行
链路级处理，因为链路级处理是在更高的
的突发第一个数据包是
直到链路级处理才交付给用户
对于突发中的所有数据包，处理已经完成。
突发中第一个数据包的交付延迟是
几乎在接收时增加。
整个突发。如果突发由多个独立-
挂起的NFS RPC请求，例如，这意味着
服务器磁盘空闲时，它本可以
有用的工作。

一位作者之前已经描述了
证明此效果的实验 [12]。

4.4. 过载下的传输饥饿

在大多数系统中，数据包传输过程
由从输出队列中选择数据包组成，
将它们交给接口，等待接口已经发送了数据包，然后释放
面已经发送了数据包，然后释放
关联的缓冲区。

数据包传输通常在较低的优先级下进行。
优先级高于数据包接收。这种策略是超级-
官方说法，因为它最大限度地降低了
数据包丢失，当突发到达数据包超过
可用缓冲区空间。合理的操作
高层协议和应用程序，然而，接收
传输处理需要足够
在进步。

当系统长时间过载时，
使用固定低优先级传输会导致
吞吐量减少，甚至完全停止
数据包传输。数据包可能正在等待传-
输，但传输接口处于空闲状态。我们
我们称之为传输饥饿。

如果发送器
中断的优先级低于接收器；或者如果
它们中断的优先级相同，但接收器的
事件首先由驱动程序进行处理；或者如果转换-
任务完成是通过轮询检测的，并且-
轮询的优先级低于接收器事件-
处理。

这种影响也已被描述
之前[12]。

5. 通过更好的讨论

在本节中，我们讨论了避免接收死锁的几种技术。我
们讨论的技术
本节包括控制速率的机制
用于处理传入中断的轮询机制，
保资源公平分配，以及减少延迟的技
避免不必要的抢占。

5.1. Limiting the interrupt arrival rate

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

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

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

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

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

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

5.2. Use of polling

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

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

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

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

5.3. Avoiding preemption

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

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

5.1. 限制中断到达率

我们可以通过限制来避免或延迟接收死锁
即中断对系统施加的速率，但可能无法保证进度；系统
但可能无法保证进度；系统。
正在占用超过其应占的资源份额，如果是这样，
暂时禁用中断。

系统可能会因为即将发生的死锁而推断出来，
由于队列溢出而丢弃数据包，或
因为高层协议处理或用户代码
没有任何进展，或通过测量分数
的CPU周期用于数据包处理。一旦
系统已经在一个传入
数据包到即将排队的程度，它
处理该数据包更有意义以完成
它而不是丢弃它并救援一个随后
到达的数据包在接收端被丢弃
接口，一个可以无限重复的循环。

当系统即将丢弃一个接收到的
数据包时，因为内部队列已满，这强烈
建议它应该禁用输入中断。这
主机可以在已定位的CPU资源上
排队进行高层处理时，它具有
释放缓冲区以供后续使用的副作用
接收的数据包。同时，如果接收接口
方面有足够的缓冲，额外的
传入数据包可能会在那里积累一段时间。

我们也需要一个触发器来重新启用输入间-
中断，以防止不必要的数据包丢失。中断
可能会在内部缓冲区空间可用时重新启用，
在内部缓冲区空间变得可用，或计时器到期时。

我们可能还希望系统保证某些
用户级代码的进度。系统可以观察
那，在某个时间间隔内，它花费了太多时间
处理数据包输入和输出事件，和 tem-
时禁用中断以提供更高的协议
用层和用户进程时间来运行。在处理器
具有细粒度时钟寄存器时，数据包输入代码可以记录进入
时的时钟值，减去
从退出时看到的时钟值，并保持一个
差值的总和。如果这个总和（或一个运行平均值）
超过指定的分数的总经过时间，
内核禁用输入中断。(Digital's
GIGAswitch™ 系统使用类似的
机制 [15]。）

在一个没有细粒度时钟的系统中，可以
通过采样CPU状态来粗略模拟这种方法
在每个时钟中断时（时钟中断通常
调用抢占设备中断处理）。如果
系统发现自己正处于处理软件中断、轮询接口以获取事件，以及
处理，它可以为一系列此类样本禁用中断
中断持续几个时钟滴答。

5.2. 轮询的使用

限制中断率可以防止系统饱和，
会检查中断处理是否正常，
也必须公平分配数据包处理资源
输入和输出处理之间，以及
多个接口。我们可以通过注意来提供公平性，
完全轮询所有数据包事件源，使用一个
轮转调度。

在纯投票系统中，调度器会
调用设备驱动程序来“‘监听’’传入
数据包和传输完成事件。这
将控制设备级处理的数量，
并且也可以在事件之间公平分配资源
源，从而避免死锁。简单地轮询
以固定间隔，然而，增加了不可接受的延迟
用于数据包接收和传输。

轮询设计和中断驱动设计有所不同
在它们放置策略决策的位置上。当
任务的特性无法预测，我们依赖
调度器和中断系统动态地调
进行进度。当任务可以预期
以可预测的方式行为，任务它们
自身能够更好地做出调度决定
之间，轮询依赖于自愿合作
在任务中。

由于纯中断驱动系统会导致
死锁，而纯轮询系统会增加不必要的
sary 延迟，我们采用混合设计，其中
系统仅在触发中断时进行轮询，并且
中断仅在轮询被暂停时发生。
在低负载下，数据包到达是不可预测的
并我们使用中断来避免延迟。在高
负载下，我们知道数据包正在或接近
系统的饱和率，因此我们使用轮询来临
确保进度和公平性，并且仅重新启
中断在没有更多工作待处理时发生。

5.3. 避免抢占

如我们在第4.2节所示，接收死锁发生时，
因为中断处理抢占所有其他
数据包处理。我们可以通过
使高层数据包处理不可-
可抢占的。我们观察到这可以做到 fol-
以下两种一般方法之一：做（几乎）
在高位IPL下做所有事情，或在高位
IPL.

按照第一种方法，我们可以修改
4.2BSD设计（参见第4.1节）通过消除
中断以处理一系列此类样本，它可以禁用中断处理接收到的数据包以完成设备
处理接收到的数据包以完成设备
IPL。因为高层处理发生在设备IPL，所以它不能
被另一个数据包到达抢占，因此我们保证死锁不
会在内核协议栈内发生。我们仍然

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

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

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

5.4. Summary of techniques

In summary, we avoid livelock by:

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

We maintain high performance by

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

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

6. Livelock in BSD-based routers

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

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

6.1. Measurement methodology

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

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

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

6.2. Measurements of an unmodified kernel

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

需要使用一个速率控制机制来确保
通过用户级应用程序进行进展。

在一个采用第二种方法的系统中，
中断处理程序仅运行足够长的时间来设置一个 ‘ ‘ser-
需要 ‘’ 标志，并调度轮询线程
如果它尚未运行。轮询线程在
零 IPL，检查标志以决定哪些设备
需要服务。只有当轮询线程完成
设备中断一次，因此它以全速运行而不中断。

设备，所以它以全速运行而不中断。
干扰。

任何一种方法都消除了在设备驱动程序和更高-
源主机在设备驱动程序和更高-
级别协议软件，尽管如果协议栈
必须阻止，传入数据包必须在某个
稍后的点。（例如，当
数据准备好交付给用户进程，或
当IP碎片被接收时及其伴随
碎片尚未可用。）

5.4. 技术总结

总之，我们通过以下方式避免死锁：

- 仅使用中断来启动轮询。
- 使用轮转轮询在事件源之间公平分配资源。

- 在满队列的反馈或CPU使用率限制时暂时禁用输入，
表示有其他重要任务正在等待。
- 尽早丢弃数据包，而不是晚些时候丢弃，以
避免浪费工作。一旦我们决定接收
一个数据包，我们尝试将其处理完成。

我们通过

- 在没有工作待处理时重新启用中断
ing，以避免轮询开销并保持
延迟较低。
- 让接收接口缓冲突发，以
避免丢弃数据包。
- 消除IP输入队列，以及相关的
开销。

我们注意到，低效的代码倾向于
加剧接收死锁，通过降低系统的MLFRR并因此增加

死锁发生的可能性。激进的op-
优化，‘ ‘快速路径’ ’ 设计，以及移除未修改的
必要的步骤都有助于推迟
死锁。

6. 基于BSD的路由器中的死锁

在本节中，我们考虑一个具体的例子
使用Digital UNIX构建的IP数据包路由器的一个系统调用
(曾用名 DEC OSF/1)。我们选择了这个应用程序
情况下，因为路由性能很容易测量。
此外，由于防火墙通常使用基于UNIX的

路由器，它们必须避免死锁，以便
防止拒绝服务攻击。

我们的目标是 (1) 获得尽可能高的
最大吞吐量； (2) 保持高吞吐量
即使过载； (3) 分配足够的 CPU
周期以用户模式任务； (4) 最小化延迟；和
(5) 避免降低其他应用的性能。
条款。

6.1. 测量方法

我们的测试配置包括一个路由器-under-
测试连接两个未加载的以太网。一个
源主机在设备驱动程序和更高-
比率，并通过路由器发送到目标
地址。（目标主机不存在；我们
通过向其插入幻影条目来欺骗路由器
ARP表。）我们通过
计算数据包数量成功
在给定时期内获得奖励，产生平均值
转发速率。

待测路由器是一个DECstation™
3000/300 基于Alpha的系统运行Digital UNIX
V3.2，SPECint92评分为66.2。我们选择了
最慢的可用Alpha主机，以使死锁问题更加明显。源主机
是一个
DECstation 3000/400，其SPECint92评分为74.7。我们
稍微修改了它的内核以允许更多
有效地生成输出数据包，以便我们
可能会对待测路由器施加最大压力。

在所有这里报告的试验中，数据包生成
发送器发送了10000个UDP数据包，每个数据包携带4个字节的
数据。该系统不会生成精确的
以固定速率的数据包流；报告的数据包比率是
在几秒钟内进行平均，并且短期
比率与平均值有所不同。我们计算了
通过使用 ‘ ‘netstat’ ’ 来计算交付的数据包速率
程序（在路由器上）来采样输出
每次试验前后的接口计数（ ‘ ‘Opkts’ ’ ）。
我们使用stub上的网络分析器进行了检查
以太网，该计数确实报告了
在输出接口上传输数据包。

6.2. 未修改内核的测量

我们首先测量了
未修改的操作系统，如图6-1所示。
每个标记代表一次试验。实心圆圈
显示基于内核的转发性能，而开口方块显示使用screend
的性能
程序 [7]，用于某些防火墙以筛选出不需要的
的包。这个用户模式的程序对每个包执行一个
；数据包转发路径
它既包含内核代码也包含用户模式代码。在这个
这个案例中，screend 被配置为接受所有数据包。

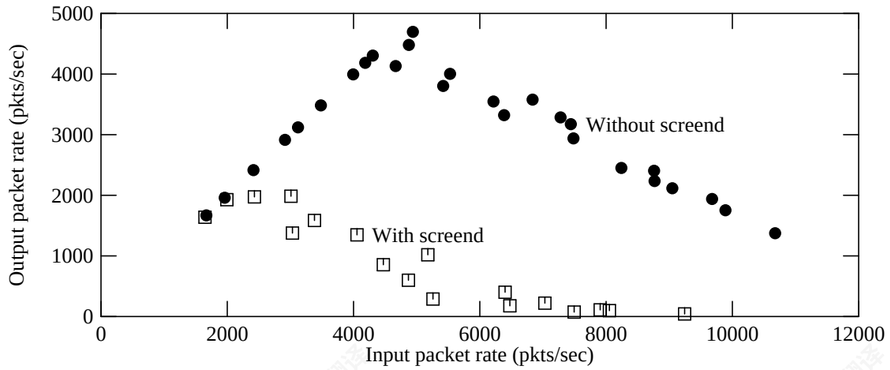

Figure 6-1: Forwarding performance of unmodified kernel

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

6.3. Why livelock occurs in the 4.2BSD model

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

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

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

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

6.4. Fixing the livelock problem

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

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

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

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

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

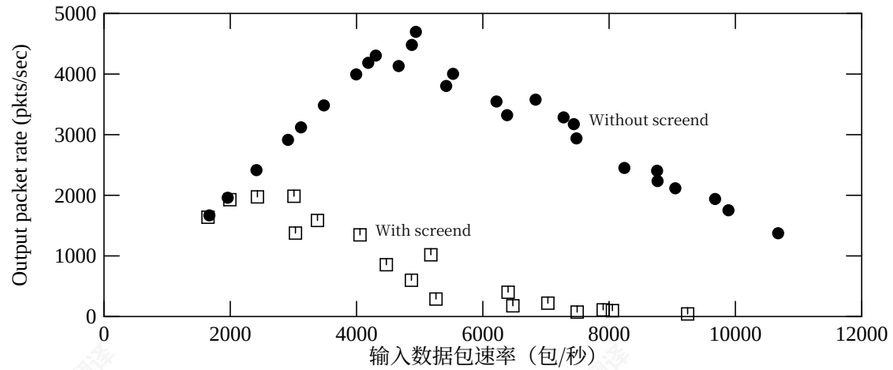

图6-1：未修改的内核转发性能

从这些测试中可以看出，使用screend
running, the router suffered from poor overload be-
行为，当速率超过2000包/秒时，并且完成
死锁设置在大约6000包/秒。即使在没有screend的情况下，
没有screend，路由器在4700包/秒时达到峰值，并且可能
会在
最大以太网数据包速率约为14,880
每秒的数据包。

6.3. 为什么4.2BSD模型中会出现死锁

4.2BSD 遵循第 4.1 节中描述的模型
4.1，并在图6-2中描绘。设备驱动程序
以中断优先级级别（IPL） = SPLIMP 运行，并
IP层通过软件中断在IPL =运行
SPLNET，它低于SPLIMP。队列
在驱动程序和IP代码之间被命名为
‘‘ipintrq,’’ 每个输出接口都由一个
自己的队列。所有队列都有长度限制；
多余数据包被丢弃。在这个
系统实现中断批处理，因此在高输入
比率非常少的中断实际上被采用。

Digital UNIX 遵循类似的模型，IP
层作为单独调度的线程在IPL运行
= 0，而不是作为软件中断处理程序。

现在为什么系统遭受
从接收死锁中恢复。一旦输入速率超过
即设备驱动程序可以拉取新数据包的处理速率，
从界面中提取出来，并添加到IP输入
队列，IP代码永远不会运行。因此，它永远不会
从其队列中删除数据包（ipintrq），这会填充
中断。它不会设置设备的可中断标志

系统的CPU资源已饱和。
导致它消耗大量CPU时间后丢弃每个数据包
在提升的IPL中已投入大量资源。这是愚蠢的；一
旦数据包通过设备驱动程序，它就代表了一项投
资，如果可能的话，应该被处理到底。在一个
路由器，这意味着数据包应该转-
mitted on the output interface. When the system is
过载，它应该尽可能早地丢弃数据包
可能（即在接收接口），以便于丢弃
丢弃的数据包不会浪费任何资源。

6.4. 解决死锁问题

我们通过尽可能在内核线程中完成
工作，而不是在中断处理程序中，并通过消除IP输入

队列及其相关的队列操作和 1
软件中断（或线程调度）。一旦我们
决定从接收接口获取一个数据包，
我们尝试在稍后不丢弃它，因为这样会
代表浪费的努力。

我们还尝试小心地 “‘调度’’ 工作
在这个线程中完成。可能无法使用
系统真实调度器来控制其处理
每个数据包，所以我们让这个线程使用一个poll-
一种技术来高效模拟轮转
数据包处理调度。轮询线程
使用额外的启发式算法来帮助满足我们的性能-
mance goals.

在新系统中，一个的
接口驱动程序几乎不做任何工作。相反，
它简单地为轮询线程（如果它还没有
已经被调度），记录其对速率的需求，
然后从接口中断返回，并将它们添加到IP输入
中断。它不会设置设备的可中断标志
标志，所以系统不会被额外的中断分散注意力
额外的中断，直到轮询线程
已处理所有挂起的包。

在启动时，修改的接口驱动程序
向轮询系统注册自己，提供

1 这并不是一个如此激进的想法；范·雅各布森已经将其用作提
高终端系统TCP性能的方式
性能[4]。

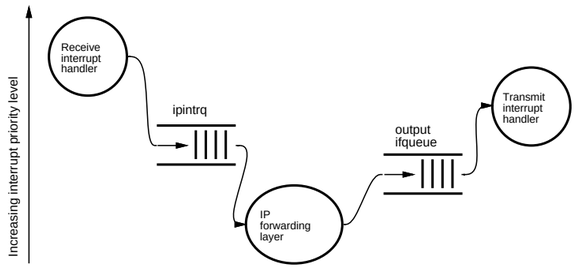

Figure 6-2: IP forwarding path in 4.2BSD

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

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

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

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

6.5. Results and analysis

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

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

6.6. Scheduling heuristics

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

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

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

6.6.1. Feedback from full queues

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

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

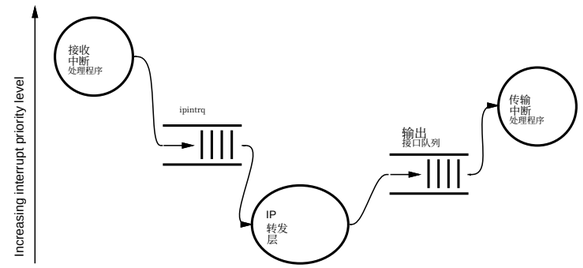

图6-2：4.2BSD中的IP转发路径

移接收和处理回调程序，
传输的数据包，并用于启用中断。
当轮询线程被调度时，它会检查所有
注册设备，以查看它们是否已请求
处理，并调用适当的回调程序来执行中断处理程序会做的
事情
在未修改的内核中已经完成的。
接收数据包的回调程序调用
IP输入处理例程直接，而不是
将接收到的包排队以供后续处理；
这意味着从接口接受的任何包
面部尽可能进行进程处理（例如，以输出）
转发队列，或到队列
向进程交付）。如果系统落后，则
接口的输入缓冲区将吸收数据包一段时间
资源浪费，并且任何多余的数据包将被丢弃
系统在浪费任何资源之前接口的速度一样快
在它上面。
轮询线程传递回调程序
对它们允许处理的数据包数量有配额
处理。一旦回调用完了它的配额，它
必须返回轮询线程。这允许
线程在多个接口之间轮转
和在任何给定的输入和输出处理之间
接口，以防止单个输入流从
垄断CPU。
当接口上所有待处理的数据包都
被处理完毕后，轮询线程也会调用
通过驱动程序的中断启用回调来处理的，以便一个子-
顺序数据包事件将导致中断。

6.5. 结果与分析

图6-3总结了我们
变化，当不使用screend时。显示了几个不同的内核配置，
使用不同的标记
图上的符号。修改后的内核（显示
带有方形标记）略微提高了MLFRR，
并避免在高输入速率下死锁。
修改后的内核可以配置为表现得好像
它是一个未修改的系统（用空心
圆圈），尽管这似乎表现稍好
比实际的未修改系统还要差（填充
圆圈）。原因尚不明确，但可能与
稍微更长的代码路径、不同的编译器或未-
指令缓存冲突的幸运变化。

6.6. 调度启发式算法

图6-3显示，如果轮询线程放置
无配额对回调处理的包数量
当输入速率超过时，处理程序可以处理的
对于MLFRR，总吞吐量几乎降至零
(如图中菱形所示)。这个死锁
发生的原因是尽管数据包不再
在IP输入队列中被丢弃，它们仍然在堆积
在输出队列中（并且被丢弃）
接口。这个队列是无法避免的，因为存在
没有保证输出接口运行的速度与
输入接口。
系统为什么无法清空输出
队列？如果数据包到达太快，输入处理
回调永远无法完成其工作。这意味着
轮询线程永远无法调用输出处理
传输接口的回调，
防止释放发射器缓冲描述符
用于进一步的传输包。这是
类似于已识别的传输饥饿条件
在 4.4 节中。
无配额的情况下，结果实际上更差，
修改后的内核，因为在那个系统中，数据包是
因输出队列空间不足而被丢弃，
而不是在IP输入队列上。未修改的内核对每个丢弃的数据
包处理的工作量较少，并且
因此偶尔能够快速丢弃它们以
赶上突发输入数据包。

6.6.1. 来自满队列的反馈

修改后的系统在时表现如何
screend程序是用于什么？图6-4比较了
未修改内核的性能（实心圆圈）
和几个修改后的内核。
按照上述方式修改内核后
(squares), 系统 performs about as badly as the

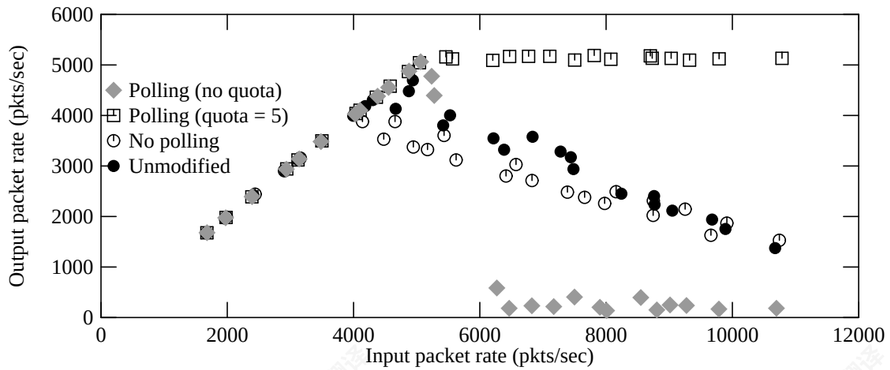

Figure 6-3: Forwarding performance of modified kernel, without using *screend*

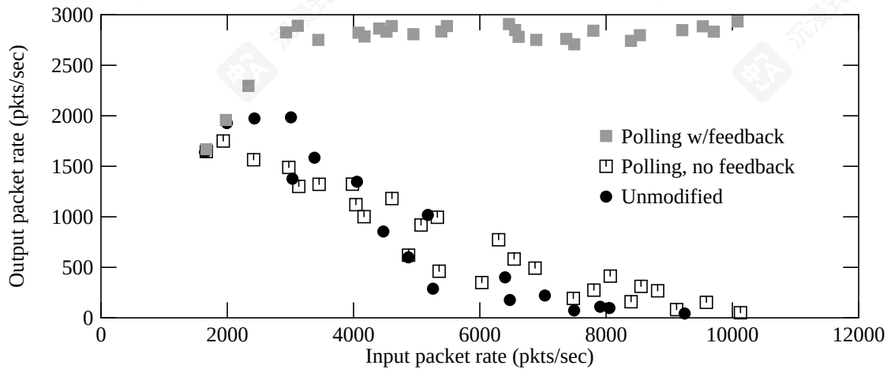

Figure 6-4: Forwarding performance of modified kernel, with *screend*

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

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

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

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

6.6.2. Choice of packet-count quota

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

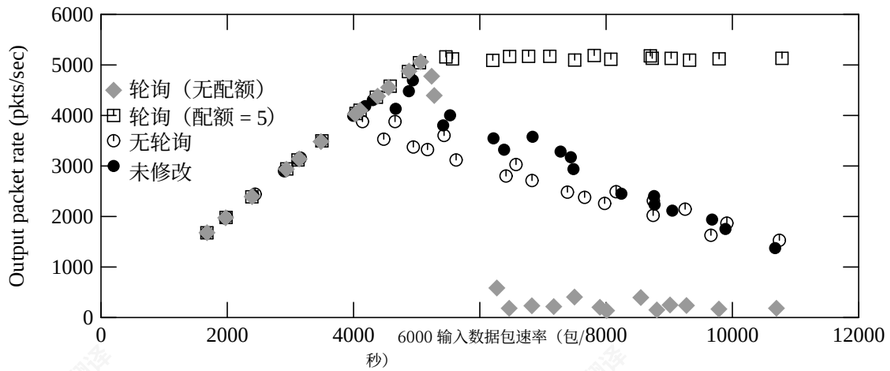

图 6-3：修改后的内核转发性能，未使用 screend

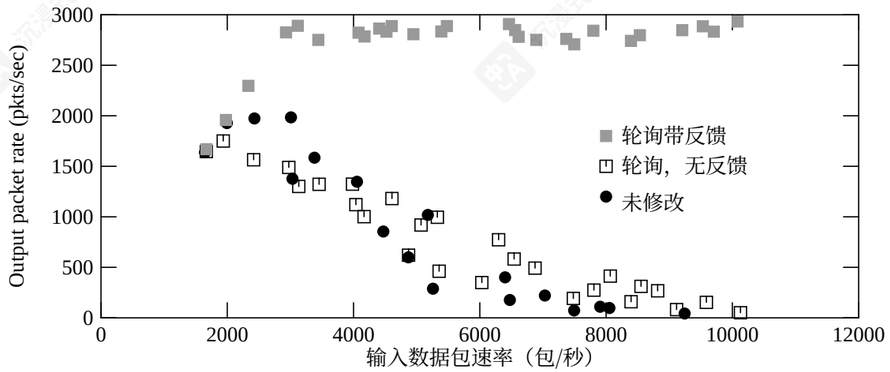

图 6-4：修改后的内核转发性能，带 screend

未修改的内核。问题是，因为
screend 在用户模式下运行，内核必须将数据包排队
排队以交付给 screend。当系统处于
过载，这个队列会填满，数据包将被
丢弃。screend 永远没有机会运行以清空
这个队列，因为系统将其周期用于
处理输入数据包。

要解决这个问题，我们检测到
筛选队列已满并抑制进一步输入网络监控）{v1}等。反馈
策略，这些队列的处理会更复杂，
空间可用。结果以灰色显示
图6-4中的方形标记：没有死锁，并且很多
提高了峰值吞吐量。队列的反馈
状态意味着系统正确分配了CPU
资源来移动数据包全部通过
系统，而不是在中间位置丢弃它们
点。

在这些实验中，轮询配额为 10
数据包，筛选队列限制为 32 个数据包-
在输入处理被抑制时，我们抑制了输入处理。
队列已满75%。输入处理被重新启用
当筛选队列满25%时。我们
我们任意选择了这些高低水位线，和
一些调优可能会有帮助。我们还设置了一个超时（ar-
被任意选择为一次时钟滴答，或大约 1 毫秒）
挂起状态时，输入将被重新启用，以防 screend
程序挂起，因此其他消费者的数据包
不会无限期丢弃。

同样的队列状态反馈技术可以
应用于系统中的其他队列，例如
接口输出队列、数据包过滤器队列（用于使用
处理（和输入中断）直到这些队列有更多队列
这些队列的策略会更复杂，
因为它可能很难确定输入
处理负载实际上在阻止进度
这些队列。由于screend程序通常是
作为系统上唯一的应用程序运行，但是，一个
完整筛选队列是一个明确的信号，表明太多
许多数据包正在到达。

6.6.2. 数据包计数配额的选择

为了避免非 screend 配置中的死锁，我们
tion，我们不得不对数据包的数量设置配额
每个回调处理的数据包数量，所以我们调查了sys-
系统吞吐量随着配额的变化而变化。
图6-5显示了结果；较小的配额效果更好

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

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

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

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

7. Guaranteeing progress for user-level processes

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

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

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

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

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

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

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

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

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

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

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

随着配额的增加，死锁变得
这更像是一个问题。

然而，当screend被使用时，队列状态实
反馈机制防止死锁，并且小
配额略微降低最大吞吐量（通过
约5%）。我们认为通过处理更多
每个回调的数据包，系统摊销了
轮询更有效，但增加了配额
这也可能增加最坏情况下的每包延迟。
一旦配额足够大，足以填满筛选
队列中有一个数据包突发，反馈
机制可能隐藏了任何潜在的改进-
中。图 6-6 显示了当 screend
进程正在使用中。

总之，无论是带 screend 还是不带 screend 的测试
表明配额在10到20个数据包之间
产生稳定和接近最优的行为，对于
测试的硬件配置。对于其他CPU和
一部分，正确的值可能会有所不同，因此
此参数应该是可调的。

7. 保证用户级进程

轮询和队列状态反馈机制
这些机制可以确保所有必要
数据包处理阶段的进度会推进，即使
在输入过载期间。它们对
其他活动的需求，然而，因此用户级
进程可能仍然缺乏CPU周期。这
使系统的用户界面无响应并
干扰了维护任务（例如路由
表维护）。

我们通过运行一个计算-
我们修改的路由器上的绑定进程，然后
用最小尺寸的数据包淹没路由器以
转发。路由器在输出处理时转发数据包。
满速率（即，好像没有用户模式进程在
消耗资源），但用户进程没有
可测量的进展。

由于根本问题是数据包输入
处理子系统占用过多CPU，我们
应该能够通过简单地测量实-
接收处理所花费的CPU时间量
数据包，如果这超过一个
阈值。

我们在Alpha架构上进行了这些
实验，包括高分辨率低开销
该寄存器计算每次实际中断；这些周期不被
生命周期（在当前实现中）并且可以
一次读取一条指令，没有任何数据缓存
未命中。其他现代 RISC 架构支持
相似的计数器；英特尔的奔腾已知有一个作为不受支
持的功能。

我们测量了在定义的期间内CPU使用率
作为多个时钟滴答（10毫秒，在我们当前的im-
现，任意选择以匹配反馈机制，防止死锁，并
调度器的量子）。每个周期，一个计时器
函数清除在中所使用的CPU周期累计总数
数据包处理代码。

每次我们的修改后的内核开始轮询
循环，它读取循环计数器，并在循环结束时再次读取
循环的末尾，以测量周期数
在处理输入和输出数据包期间花费了
循环。（配额机制确保了这种互-
价值相对较短。）这个数字然后被添加到
累计总数，如果这个总数超过一个
阈值，输入处理立即被抑制。
在当前周期的结束时，一个计时器重新启用
输入处理。系统空闲线程的执行
还重新启用输入中断并清除运行
总体而言。

通过将阈值调整为网络接口的
一个周期内的总周期数，可以控制
相当精确地控制处理数据包所花费的CPU时间。我们
尚未为这项控制实现编程接口；在我们的测试中，

我们简单地修补了一个内核全局变量表示分配给网络处理的百分比，
表示分配给网络处理的百分比，
以及内核自动将此转换为第6.4节中描述的数字，
周期数。

图7-1显示了有多少CPU时间可用
能够为计算密集型用户进程，对于几个
周期阈值和不同输入速率的设置。
曲线显示随着输入速率增加，行为相当稳定。
速率增加，但用户进程没有获得足够的
比阈值设置所暗示的更多的CPU时间。

差异的部分来自于系统过载-
头；即使没有输入负载，用户进程也得到
大约94%的CPU周期。此外，周期限制
该机制抑制数据包输入处理，但不是
在更高的输入速率下，在输入
被抑制时，输出队列填满足够多以至于可以吸收
额外的CPU周期。

测量误差可能导致一些额外的
差异。周期阈值仅被检查
在处理一批输入数据包（对于这些例-
验来改善这种情况，回调配额为5个数据包）。使用
系统转发约5000包/秒，
处理这种突发大约需要1毫秒，或大约
阈值检查周期的10%。

50% 和 75% 的曲线初始下降
阈值可能反映了处理计数寄存器的成本。
计入指令周期（在当前实现中）并且可以
阈值，并且在输入速率低于饱和时，
每个传入数据包都可能被足够快地处理
确保不会发生中断批处理。

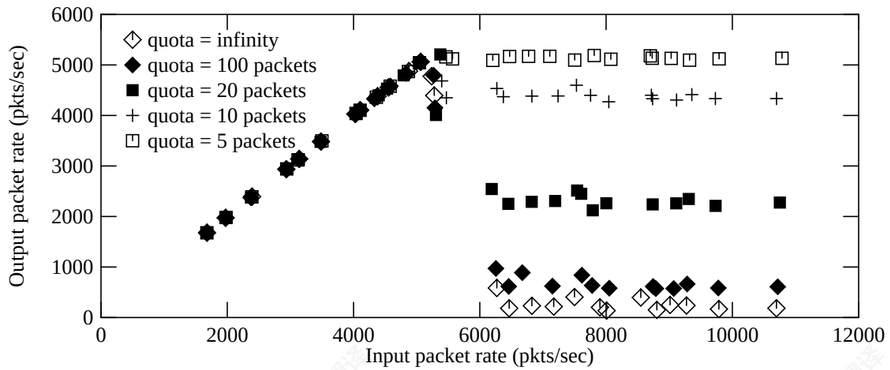

Figure 6-5: Effect of packet-count quota on performance, no *screend*

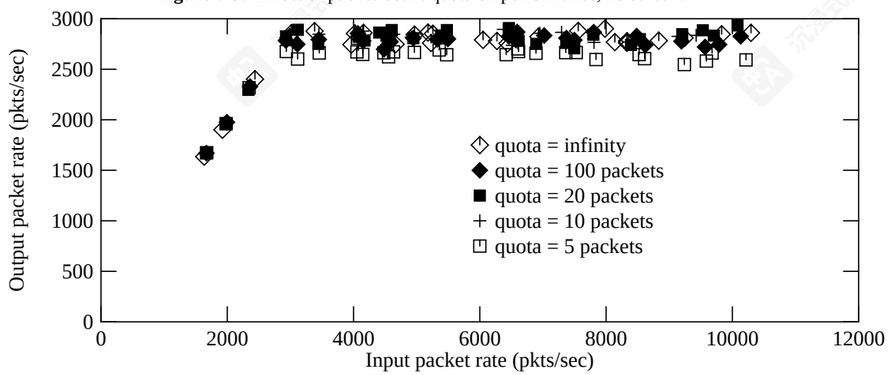

Figure 6-6: Effect of packet-count quota on performance, with *screend*

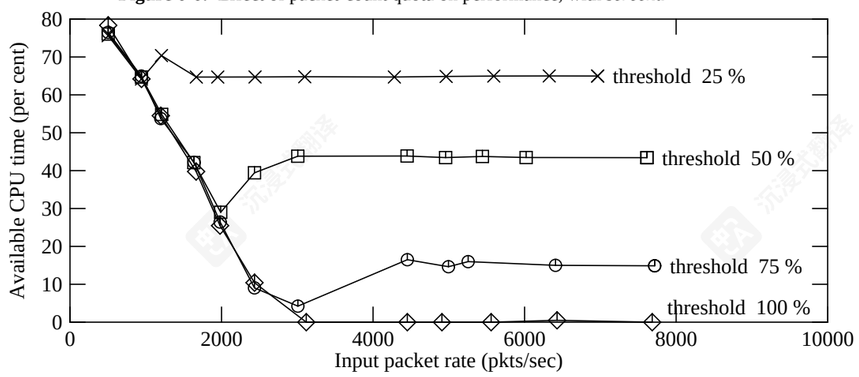

Figure 7-1: User-mode CPU time available using cycle-limit mechanism

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

7.1. Performance of end-system transport protocols

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

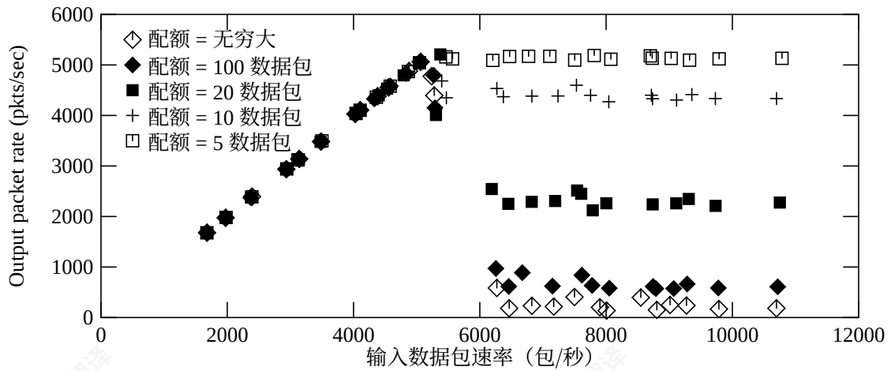

图6-5：数据包计数配额对性能的影响，无screend

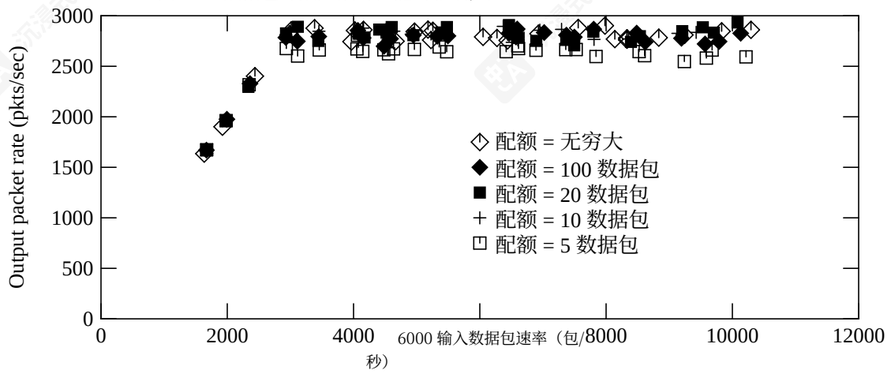

图 6-6：数据包计数配额对性能的影响，with screend

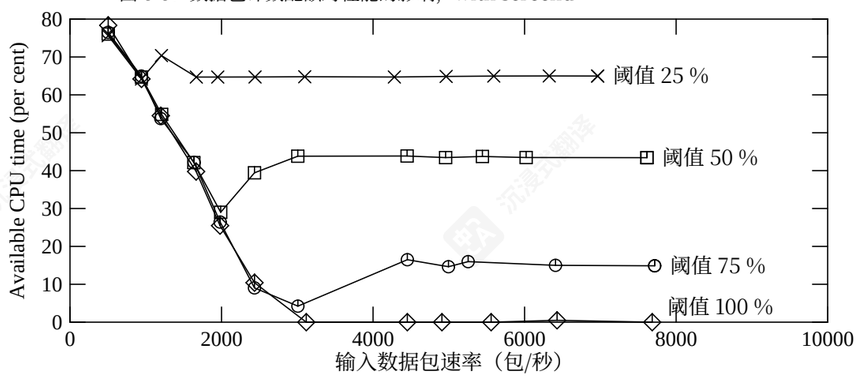

图7-1：使用周期限制机制可用的用户模式 CPU 时间

对数据包处理施加了周期限制，系统主观上远更响应迅速，甚至在重输入过载期间。这种改进，然而，主要对本地用户明显；任何基于网络的交互，例如 Telnet，仍然受影响，因为许多数据包正在被丢弃。

7.1. 终端系统传输的性能协议

我们对内核所做的更改可能都会影响端系统传输的性能协议，例如 TCP 和 UDP/RPC/XDR/NFS 堆栈。由于我们尚未将我们的修改应用于高速网络接口驱动程序，因此

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

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

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

8. Related work

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

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

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

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

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

9. Summary and conclusions

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

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

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

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

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

Acknowledgements

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

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

例如 FDDI 的一个，我们目前还不能测量这种影响。
（测试系统很容易就能使以太网饱和，所以）
测量以太网上的TCP吞吐量显示没有
影响。）

处理接收到的数据包的技术
直接从设备驱动程序到TCP层，
而不会将数据包放置在IP层队列上，这是
范·雅各布森特别用于改进TCP
性能[4]。它应该通过避免队列操作和

任何相关的锁定；它还应该提高
负载失败无法提供一致的性能和良好的
交互行为。死锁永远不会是最好的

轮询接口的技术不应该
降低终端系统性能，因为它被完成
主要在输入过载期间。（一些实
实现使用轮询来避免传输中断
总共 [6]。）在过载时，未修改的 sys-
系统将不会在应用程序或
传输协议；轮询、队列状态的使用，
反馈，并且CPU周期限制应该给予
修改后的系统至少有一些机会
进度。

8. 相关工作

轮询机制以前在
基于UNIX的系统，无论是在网络代码中还是在
其他上下文。而我们已经使用轮询来
提供公平性和保证进度，但之前的
轮询的前期应用旨在减少
与中断服务相关的开销。这确实会降低系统过载（对于一个
给定的输入速率），但并不能防止死锁。

Traw和Smith [14, 16] 描述了
‘‘时钟中断，’’ 周期性轮询以了解
驱动数据包而不产生每包的开销
中断。他们指出，选择
适当的轮询频率：太高，系统
将所有时间都用于轮询；太低，接收
延迟飙升。他们的分析 [14] 似乎忽略了使用中断批处理来减少中断-
服务开销；然而，他们确实提到了使用方案的
使用中断的方案的可行性，其中
提示轮询其他事件。

4.3BSD操作系统[5] 显然使用了
一种周期性轮询技术来处理接收
来自一个八端口终端接口的字符，如果
最近的输入速率超过了某个阈值。
似乎是为了避免丢失输入
角色（该设备缓冲可用量很少）
但人们可以将其视为一种避免死锁的策略。几种
路由器实现使用轮询作为其调度数据包处理的主
要方式。

当一个拥塞的路由器必须丢弃一个数据包时，它
丢弃哪个数据包的选择可能具有重大意义
影响。我们的修改不会影响哪个包
ets are dropped; we only change when they are
被丢弃。该策略过去是，现在仍然是 ‘‘尾部丢弃’’；
其他策略可能会提供更好的结果 [3]。

我们早期关于改进界面的一些工作
驱动算法的描述在[1]中。

9. 总结和结论

系统在接收内核间交互（如TCP）的延迟过高时表现不佳，
致谢和NFS RPC）。
轮询接口的技术不应该
对过载的响应。在本文中，我们已经展示了
如何理解系统过载行为以及如何
通过仔细调度何时数据包
处理完成。

我们使用 UNIX 的测量结果证明了，
系统上取得任何进展，传统的中断驱动系统
form严重过载，导致接收
死锁和传输的饥饿。因为这样的
系统逐渐降低进程的优先级
在数据包进一步进入系统时接收它，当过载时它们表现出
过多的数据包丢失和
浪费的工作。这种病理可能不是
不仅由长期接收过载，而且还由 tran-
由短期突发到达引起的过载。

我们描述了一组调度改进
那些有助于解决过载问题的轮询应用
havior. 这些包括：

- 限制中断到达率，以卸载过载
- 轮询以提供公平性
- 处理接收到的数据包直至完成
- 显式地调节CPU使用率用于数据包处理

我们的实验表明这些调度
机制提供良好的过载行为和
消除接收死锁。它们应该帮助两者
专用和通用系统。

致谢

我们在进行测量和
从许多方面理解系统性能
包括比尔·霍伊、托尼·劳克和约翰。
普林、乌塔姆·希卡普尔和约翰·达斯汀。文卡塔
帕德马纳布汉、大卫·切尔库斯和杰弗里·亚普利
在文稿准备过程中提供了帮助。

K. K. 拉马库什南关于这篇论文的大部分工作
是在他作为Digital员工时完成的。
设备公司。

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

参考文献

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

杰弗里·莫格尔于1979年在麻省理工学院获得理学学士学位，并于1980年和1986年在斯坦福大学获得理学硕士学位和哲学博士学位。自1986年以来，他一直是意大利西部研究中心的研究员，g y g on network and operating systems issues for high-performance computer systems. He is the author or co-author of several Internet Standards, an associate editor of *Internetworking: Research and Experience*, and was Program Chair for the Winter 1994 USENIX Conference.

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

K. K. Ramakrishnan is a Member of Technical AT&T贝尔实验室的员工。他于1976年在印度班加罗尔大学获得理学学士学位，于1978年在印度科学理工学院获得理学硕士学位，并于1983年在马里兰大学获得哲学博士学位。未- y y til 1994, he was a Consulting Engineer at Digital. Ramakrishnan's research interests are in performance analysis and design of algorithms for computer networks and distributed systems. He is a technical editor for *IEEE Network Magazine* and is a member of the Internet Research Task Force's End-End Research Group.

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

UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company, Ltd.
Windows NT is a trademark of Microsoft, Inc.
GIGAswitch, VMS, and DECstation are trademarks of Digital Equipment Corporation.