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

消除中断驱动内核中的接收活锁

Jeffrey Mogul, DEC西方研究实验室K. K. Ramakrishnan, AT&
T贝尔实验室

有关USENIX协会的更多信息，请联系:

1. 电话: 10 528 - 8649
2. 传真: 510 548 - 5738
3.。电子邮件:5 office@usenix.org
4. 网址:http://www.usenix.org

消除中断驱动内核中的接收活锁

杰弗里·莫卧尔
*数字设备公司西方研究实验室*K. K. Ramakrishnan
AT&T贝尔实验室

摘要。

大多数操作系统使用接口中断来调度网络任务。中断驱动的系统可以在低提供的负载下提供低开销和良好的延迟，但在更高的到达速率下显著降低，除非采取措施防止一些病态。这些是各种形式的***接收活锁***，在这些活锁中，系统将其所有时间用于处理中断，而排除了其他必要的任务。在极端条件下，没有包被交付给用户应用程序或系统的输出。

为了避免活锁和相关问题，操作系统必须像调度进程执行一样仔细地调度网络中断处理。我们修改了一个中断驱动的网络实现来做到这一点;这消除了接收活锁，而不会降低系统性能的其他方面。我们展示了证明我们方法成功的测量结果。

1. 介绍

大多数操作系统利用中断在内部调度与I/O事件相关的任务的性能，特别是对网络协议软件的调用。中断是有用的，因为它们允许CPU将大部分时间用于进行有用的处理，同时快速响应事件，而不必经常轮询事件到达。

轮询是昂贵的，特别是当I/O事件相对罕见时，如磁盘的情况，磁盘很少中断超过每秒几百次。轮询还会增加对事件的响应延迟。现代系统可以在几十微秒内响应一个中断;为了使用轮询实现相同的延迟，系统将不得不每秒轮询数万次，这会产生过多的开销。对于通用系统来说，中断驱动的设计效果最好。

大多数现存的操作系统被设计来处理每隔几mil秒中断一次的I/O设备。磁盘倾向于按每转一次的顺序发出事件;第一代局域网环境往往每秒为任何单个端系统生成几百个数据包。尽管人们理解需要降低中断的成本，但通常这个成本足够低，以至于任何正常的系统只需要花费CPU时间的一小部分来处理中断。

世界已经发生了变化。操作系统通常使用相同的中断机制来控制网络处理和传统的I/O设备，然而许多新的应用程序生成数据包的频率比磁盘生成寻道的频率高几个数量级。多媒体和其他实时应用将变得广泛。客户机-服务器应用程序，如NFS，运行在快速的客户机和服务器上可以产生大量的RPC负载。组播和广播协议使无辜的旁观者主机承受完全不感兴趣的负载。因此，网络实现现在必须处理明显更高的事件率。

许多多媒体和客户机-服务器应用程序共享另一个令人不快的特性:不像传统的网络应用程序(Telnet、FTP、电子邮件)，它们不受流量控制。一些多媒体应用程序希望提供恒速率、低延迟的服务;基于rpc的客户端-服务器应用程序经常使用数据报风格的传输，而不是可靠的流量控制协议。请注意，虽然I/O设备(如磁盘)只会由于来自操作系统的请求而产生中断，因此本质上是流量控制的，但网络接口会产生不请自来的接收中断。

向更高的事件率和非流量控制协议的转移会使主机发生充血性崩溃:一旦事件率饱和了系统，没有负反馈回路来控制源，就没有办法优雅地卸载负载。如果主机在这些条件下以全吞吐量运行，并对所有信源提供公平的服务，这至少保留了稳定的可能性。但如果吞吐量随着提供的负载增加而下降，那么整个系统就会变得不稳定。

中断驱动的系统往往在过载情况下表现不佳。在中断级别执行的任务，

根据定义，具有绝对优先级高于所有其他任务。如果事件率高到足以导致系统把所有时间都花在响应中断上，那么就不会有其他事情发生，系统吞吐量就会降为零。我们称这种情况为*接收活锁(receive*活锁):系统没有死锁，但它的任何任务都没有进展。

任何使用固定中断优先级的纯中断驱动系统在输入过载条件下都会受到接收活锁的影响。一旦输入速率超过处理一个输入事件的CPU成本的倒数，任何以较低优先级调度的任务都将没有机会运行。

然而，我们并不想轻易放弃中断驱动设计的明显好处。相反，我们应该将网络中断处理子系统的控制集成到操作系统的调度机制和策略中。在本文中，我们提出了一些对纯中断驱动模型的简单修改，并表明它们在过载下保证吞吐量和改善延迟，同时在轻负载下保持中断驱动系统的理想质量。

2. 激励应用程序

我们的调查是由一些可能受到活锁影响的特定应用程序引导的。这样的应用程序可以构建在专用的单一目的系统上，但通常是使用通用目的系统(如UNIX⊠)构建的，我们希望找到一个通用的解决方案来解决活动锁问题。这些应用程序包括:

基于主机的*路由*:虽然传统上网络间路由是使用专用(通常非中断驱动)路由器系统完成的，但路由通常使用更传统的主机完成。几乎所有的互联网“防火墙”产品都使用UNIX或Windows NT⊠系统进行路由[7,13]。在UNIX上进行了许多新的路由算法的实验[2]，特别是对于IP多播。

· 被动*网络监控*:网络管理人员、开发人员和研究人员通常使用UNIX系统，其网络接口处于“混杂模式”，以监控局域网上的流量以进行调试或统计收集[8]。

· 网络*文件服务*:用于协议(如NFS)的服务器通常是从UNIX系统构建的。

这些应用程序(以及其他类似的应用程序，如Web服务器)都可能暴露于沉重的、非流控制的负载。我们在这三个应用程序中都遇到过活锁，已经解决或减轻了问题，并将解决方案交付给了客户。本文的其余部分集中于基于主机的路由，因为这简化了问题的上下文，并允许简单的性能测量。

3. 调度网络任务的要求

当系统受到短暂或长期的输入过载时，通常会出现性能问题。理想情况下，通信子系统可以在不饱和的情况下处理最坏情况下的输入负载，但是成本考虑经常阻止我们建立这样强大的系统。系统的大小通常支持一个指定的设计中心负载，在过载情况下，我们所能要求的最好的是控制和优雅的退化。

当一个终端系统涉及处理相当大的网络流量时，它的性能主要取决于它的任务是如何调度的。调度数据包处理和其他任务的机制和策略应该保证可接受的系统*吞吐量*、合理的*延迟*和*抖动*(延迟的差异)、*公平*的资源分配和整体系统的*稳定性*，而不施加过多的开销，特别是当系统过载时。

我们可以将吞吐量定义为系统将数据包交付给其最终消费者的速率。消费者可以是在接收主机上运行的应用程序，也可以是主机充当路由器，将数据包转发给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能够跟上所提供的负载，达到一个称为*最大无损失接收速率*(MLFRR)的点，并且在更高的负载下，吞吐量不应低于这个速率。

当然，有用的吞吐量不仅取决于数据包的成功接收;系统还必须传输数据包。因为分组接收和分组传输经常竞争相同的资源，所以在输入过载的情况下，调度子系统必须确保分组传输以足够的速率继续进行。

许多应用，如分布式系统和交互式多媒体，往往更多地依赖于低延迟、低抖动的通信，而不是高吞吐量。即使在过载期间，我们也希望避免增加延迟的长队列，以及增加抖动的突发调度。

当一个主机因接收到的网络数据包而过载时，它还必须继续处理其他任务，这样才能使系统对管理和控制请求保持响应，并允许应用程序利用到达的数据包。调度子系统必须在分组接收、分组传输、协议之间公平地分配CPU资源

处理、其他I/O处理、系统管理和应用程序处理。

主机在过载时表现不佳，也会危害网络上的其他系统。例如，路由器中的活锁可能导致控制消息的丢失，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，导致不正确的路由信息在整个广域网上传播。更糟糕的是，控制消息的丢失或延迟会导致网络不稳定，这是由于在生成控制流量[10]时造成正反馈。

4. 中断驱动的调度及其后果

调度策略和机制会显著影响系统在过载情况下的吞吐量和延迟。在中断驱动的操作系统中，中断子系统必须被视为调度系统的一个组成部分，因为它在决定什么时候运行什么代码方面起着主要作用。我们已经观察到中断驱动的系统在满足第3节中讨论的要求方面有困难。

在本节中，我们首先描述中断驱动系统的特征，然后识别中断驱动系统中由网络输入过载引起的三种问题:

过载下的*接收活锁*:当输入过载持续时，交付的吞吐量下降到零。

· 数据包投递或转发的*延迟*增加:系统在处理后续数据包中断(可能是突发的)时，延迟了一个数据包的投递。

· 数据包传输的饥饿:即使CPU跟上输入负载，严格的优先级分配可能会阻止它传输任何数据包。

4.1. 中断驱动系统的描述

中断驱动的系统在网络输入过载下表现很差，因为它优先执行作为网络输入结果的任务的方式。我们首先描述一个典型的操作系统的处理和优先处理网络任务的结构。我们在示例中使用4.2BSD[5]模型，但我们发现其他操作系统，如VMS⊠、DOS和Windows NT，甚至几个以太网芯片，都具有类似的特性，因此也存在类似的问题。

当数据包到达时，网络接口通过中断CPU来发出此事件的信号。设备中断通常具有固定的中断优先级(IPL)，并抢占在较低IPL上运行的所有任务;中断不会抢占在同一IPL上运行的任务。中断导致进入相关的网络设备驱动程序，该驱动程序对数据包进行一些初始处理。在4.2BSD中，只有缓冲区管理和数据链路层处理发生在“设备IPL”上。然后，设备驱动程序将数据包放置在一个队列上，并产生一个软件中断，以导致对数据包的进一步处理。软件中断以较低的IPL进行，因此该协议处理可以被后续中断抢占。(我们避免在高IPL下长时间运行，以减少处理某些其他事件的延迟。)

在不同的ipl执行的步骤之间的队列提供了一些隔离，以防止由于瞬态过载而造成的数据包丢失，但通常它们有固定的长度限制。当一个分组应该排队，但队列已满时，系统必须丢弃该分组。选择适当的队列限制，从而在系统的各层之间分配缓冲，是良好性能的关键，但超出了本文的范围。

请注意，操作系统的调度器不参与任何这一活动，事实上完全不知情。

由于这种结构，传入数据包的沉重负载可能会在设备IPL上产生高中断率。分派中断是一个代价高昂的操作，因此为了避免这种开销，网络设备驱动程序试图*批量处理*中断。也就是说，如果分组突然到达，中断处理程序试图在从中断返回之前处理尽可能多的分组。这将处理一个中断的成本分摊到几个分组上。

即使使用批处理，输入数据包过载的系统将把大部分时间花在设备IPL上运行的代码上。也就是说，设计给予处理传入数据包绝对的优先级。在20世纪80年代早期开发4.2BSD时，这样做的基本原理是网络适配器只有很少的缓冲内存，因此如果系统无法将接收到的数据包迅速移动到主内存中，则后续数据包可能会丢失。(对于低成本接口来说，这仍然是一个问题。)因此，从4.2BSD派生出来的系统在设备IPL上只进行最少的处理，并赋予此处理优先于所有其他网络任务。

现代网络适配器可以在没有主机干预的情况下接收许多背靠背数据包，无论是通过使用丰富的缓冲还是高度自治的DMA引擎。这将系统与网络隔离开来，并消除了对处理接收到的分组的前几个步骤给予绝对优先级的许多理由。

4.2. 接收活锁

在中断驱动的系统中，接收中断优先于所有其他活动。如果数据包到达得太快，系统将花费所有的时间处理接收端中断。因此，它将没有剩余的资源来支持将到达的分组交付给应用程序(或者，在路由器的情况下，转发和传输这些分组)。系统的有效吞吐量将降为零。

在[11]之后，我们将这种情况称为*接收活锁(receive*活锁):系统的一种状态，在这种状态下，没有任何有用的进展，因为处理接收端中断完全消耗了一些必要的资源。当输入负载下降到足够大时，系统离开这种状态，并且再次能够向前推进。这不是死锁状态，即使输入速率下降到零，系统也不会从死锁状态中恢复。

当输入负载增加时，系统可以表现为三种方式之一。在理想的系统中，交付的吞吐量总是与提供的负载相匹配。在可实现系统中，在最大无丢失*接收率(Maximum Loss Free* Receive Rate, MLFRR)之前，交付的吞吐量与提供的负载保持一致，此后相对恒定。当负载高于MLFRR时，系统仍在取得进展，但它正在减少一些提供的输入;通常，分组在不同优先级发生的处理步骤之间的队列中被丢弃。

然而，在易于接收活动锁的系统中，对于高于MLFRR的输入速率，吞吐量随着提供负载的增加而降低。接收活锁发生在吞吐量降为零的点。一个活锁系统浪费了所有它投入到部分处理接收到的数据包的工作，因为它们都被丢弃了。

接收端-中断批处理稍微使情况复杂化。通过提高系统在大负荷下的效率，分批处理可以提高MLFRR。批处理可以转移活锁点，但不能单独防止活锁。

在6.2节中，我们展示了在实际情况下如何发生活锁的测量结果。在[11]中给出了额外的测量，以及对问题的更详细的讨论。

4.3. 过载下的接收延迟

虽然中断驱动的设计通常被认为是一种减少延迟的方法，但它们实际上可以增加数据包交付的延迟。如果突发的数据包到达得太快，系统将在对第一个数据包进行任何更高层的处理之前对整个突发进行链路级处理，因为链路级处理是在更高的优先级上进行的。因此，直到完成了对突发中所有数据包的链路级处理后，才会将突发的第一个数据包交付给用户。在一个突发中交付第一个数据包的延迟几乎与接收整个突发所需的时间相同。例如，如果突发由几个独立的NFS RPC请求组成，这意味着服务器的磁盘在它可以做有用的工作时处于空闲状态。

其中一位作者之前描述了证明这种效果的实验[12]。

4.4. 过载下的饥饿传输

在大多数系统中，数据包传输过程包括从输出队列中选择数据包，将它们交给接口，等待接口已发送数据包，然后释放作为关联的缓冲区。

分组传输通常以比分组接收更低的优先级完成。这种策略表面上是合理的，因为当到达的分组超过可用的缓冲空间时，它将分组丢失的概率降到最低。然而，更高层的协议和应用程序的合理运行，要求传输处理取得足够的进展。

当系统长时间过载时，使用固定的较低优先级进行传输，会导致吞吐量降低，甚至包传输完全停止。数据包可能正在等待传输，但传输接口处于空闲状态。我们称之为*传输饥饿*。

如果发送端中断的优先级低于接收端，就可能发生传输饥饿;或者它们以相同的优先级中断，但接收器的事件首先由驱动程序处理;或者如果通过轮询检测到传输完成，并且轮询是在比接收方事件处理更低的优先级上完成的。

这种效果在之前的[12]中也有描述。

5. 通过更好的调度避免活锁

在本节中，我们将讨论几种避免接收活锁的技术。我们在本节中讨论的技术包括控制传入中断率的机制，基于轮询的机制以确保资源的公平分配，以及避免不必要的抢占的技术。

5.1. 限制中断到达速率

我们可以通过限制在系统上施加中断的速率来避免或推迟接收活锁。系统检查中断处理是否占用了超出其资源份额的资源，如果是，则暂时禁用中断。

由于队列溢出而丢弃数据包，或者由于高层协议处理或用户代码没有进展，或者通过测量用于数据包处理的CPU周期的比例，系统可以推断即将发生的活动锁。一旦系统对一个即将进入的分组投入了足够的工作，使其达到将要排队的程度，那么处理该分组完成比丢弃它并挽救随后到达的分组不被丢弃在接收接口更有意义，这是一个可以*无限重复的循环*。

当系统因为内部队列已满而即将丢弃接收到的数据包时，这强烈建议它应该禁用输入中断。然后，主机可以在已经为更高级别处理排队的分组上取得进展，这具有释放缓冲区用于后续接收的分组的副作用。同时，如果接收接口本身有足够的缓冲区，额外的传入数据包可能会在那里积累一段时间。

我们还需要一个触发器来重新启用输入中断，以防止不必要的数据包丢失。当内部缓冲空间变得可用时，或在定时器到期时，中断可能被重新启用。

我们还可能希望系统为用户级代码保证一些进展。系统可以观察到，在一些时间间隔内，它已经花费了太多的时间来处理数据包输入和输出事件，并暂时禁用中断，以给更高的协议层和用户进程运行的时间。在具有细粒度时钟寄存器的处理器上，数据包输入代码可以在输入时记录时钟值，从退出时看到的时钟值中减去该值，并保持增量的总和。如果这个总和(或运行平均值)超过总运行时间的指定比例，内核将禁用输入中断。(Digital的GIGAswitch⊠系统使用类似的机制[15]。)

在没有细粒度时钟的系统上，可以通过对每个时钟中断(时钟中断通常抢占设备中断处理)上的CPU状态进行采样来粗略地模拟这种方法。如果系统发现自己正在为一系列这样的样本处理中断，它可以禁用中断几个时钟滴答。

5.2. 轮询的使用

限制中断率可以防止系统饱和，但可能无法保证进度;系统还必须在输入和输出处理之间，以及在多个接口之间公平地分配分组处理资源。我们可以通过使用轮询调度仔细轮询所有分组事件的来源来提供公平性。

在一个纯轮询系统中，调度器将调用设备驱动程序来“监听”即将到来的数据包和传输完成事件。这将控制设备级处理的数量，也可以在事件源之间公平分配资源，从而避免活锁。然而，简单地以固定的间隔轮询，会给数据包的接收和传输增加不可接受的延迟。

轮询设计和中断驱动设计在策略决策的位置上有所不同。当任务的行为无法预测时，我们依靠调度程序和中断系统来动态定位CPU资源。当可以预期任务以可预测的方式表现时，任务本身就能够更好地做出调度决策，轮询依赖于任务之间的自愿合作。

由于纯中断驱动的系统会导致活锁，而纯轮询系统会增加不必要的延迟，因此我们采用了混合设计，即系统只在由中断触发时进行轮询，而中断只在轮询暂停时发生。在低负载期间，数据包的到达是不可预测的，我们使用中断来避免延迟。在高负载期间，我们知道数据包正在到达或接近系统的饱和率，因此我们使用轮询来确保进度和公平性，只有在没有更多工作等待时才重新启用中断。

5.3. 避免抢占

正如我们在4.2节中所展示的，receive活锁发生是因为中断处理抢占了所有其他分组处理。我们可以通过使更高层次的包处理不可抢占来解决这个问题。我们观察到这可以遵循以下两种一般方法之一:在高IPL下做(几乎)所有事情，或者在高IPL下(几乎)什么都不做。

按照第一种方法，我们可以修改4.2BSD设计(参见4.1节)，通过消除软件中断，轮询事件接口，并处理接收到的数据包，直到在设备IPL完成。因为更高级别的处理发生在设备IPL上，所以它不能被另一个数据包到达所抢占，因此我们保证活锁不会发生在内核的协议栈中。我们仍然

需要使用速率控制机制来确保用户级应用的进度。

在遵循第二种方法的系统中，中断处理程序只运行足够长的时间来设置“服务需要”标志，并在轮询线程尚未运行时调度它。轮询线程以零IPL运行，检查标志以决定哪些设备需要服务。只有当轮询线程完成时，它才会重新启用设备中断。轮询线程最多只能被每个设备中断一次，因此它会全速前进而不受干扰。

这两种方法都消除了在设备驱动程序和高层协议软件之间对数据包进行排队的需要，尽管如果协议栈必须阻塞，则必须在稍后的点对进入的数据包进行排队。(例如，当数据准备交付给用户进程时，或者当接收到IP片段而其配套片段尚未可用时，就会发生这种情况。)

5.4. 技术总结

综上所述，我们通过以下几点来避免活锁:

· 使用中断仅用于启动轮询。

· 使用轮询来公平地在事件源之间分配资源。

· 当来自完整队列的反馈，或CPU使用的限制，表明其他重要的任务正在等待时，暂时禁用输入。

· 尽早丢弃数据包，而不是推迟，以避免浪费工作。一旦我们决定接收一个数据包，我们就会尽力把它处理好。

我们通过

· 在没有工作等待时重新启用中断，以避免轮询开销并保持低延迟。

· 让接收接口缓冲区突发，以避免丢包。

· 消除IP输入队列和相关开销。

我们顺便观察到，低效的代码往往会通过降低系统的MLFRR，从而增加发生活锁的可能性，加剧接收活锁。激进的优化、“快速路径”设计和删除不必要的步骤都有助于推迟活锁的到来。

6. 基于bsd的路由器中的活锁

在本节中，我们考虑使用数字UNIX(以前的DEC OSF/1)构建的IP分组路由器的具体示例。我们选择这个应用程序是因为路由性能很容易测量。此外，由于防火墙通常使用UNIX-basedrouters，它们必须防活锁，以防止拒绝服务攻击。

我们的目标是:(1)获得尽可能高的最大吞吐量;(2)即使超载，也要保持高吞吐量;(3)为用户模式任务分配足够的CPU周期;(4)最小化延迟;(5)避免在其他应用程序中降低性能。

6.1. 测量方法

我们的测试配置包括一个待测路由器连接两个未加载的以太网。源主机以各种速率生成IP/UDP数据包，并通过路由器将它们发送到目的地址。(目的主机不存在;我们通过在路由器的ARP表中插入一个幻影条目来欺骗路由器。)我们通过统计给定时间内成功转发的数据包数量来测量路由器的性能，得出平均转发率。

被测试的路由器是一个DECstation□3000/300基于alpha的系统，运行数字UNIX V3.2, SPECint92等级为66.2。我们选择了可用的最慢的Alpha主机，以使活锁问题更加明显。源主机为DECstation 3000/400, SPECint92评分74.7。我们稍微修改了它的内核，以允许更有效地生成输出数据包，这样我们就可以尽可能地对待测路由器施加压力。

在这里报告的所有试验中，包生成器发送了10000个携带4字节数据的UDP包。这个系统并不能生成精确步调的数据包流;所报告的分组速率是在几秒钟内平均的，并且短期速率与平均速率略有不同。我们通过使用“netstat”程序(在路由器机器上)在每次试验之前和之后对输出接口计数(“Opkts”)进行采样来计算交付包速率。我们使用存根以太网上的网络分析器检查，该计数准确地报告了在输出接口上传输的数据包数量。

6.2. 未修改内核的测量

我们首先测量未修改的操作系统的性能，如图6-1所示。每一个标记代表一次试验。填充的圆圈表示基于内核的转发性能，开放的正方形表示使用屏幕程序[7]的性能，该程序在一些防火墙中用于筛选不需要的数据包。这个用户模式程序对每个包做一个系统调用;数据包转发路径包括内核和用户态代码。在这种情况下，*screend*被配置为接受所有数据包。

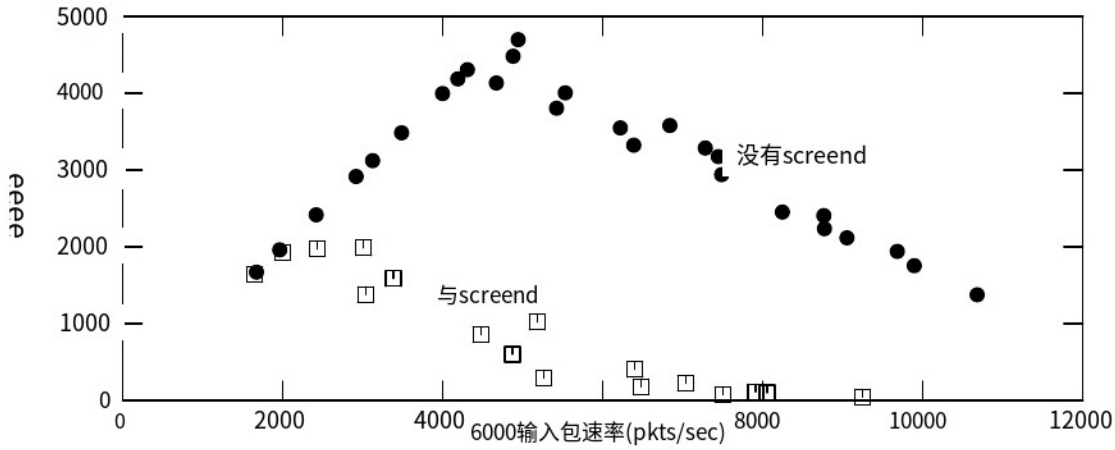

图6-1:未修改内核的转发性能

从这些测试中可以明显看出，在*screend运行时*，路由器在2000包/秒以上的速率上遭受了糟糕的过载行为。，而完整的活锁设置在大约6000包/秒。即使没有*屏幕*，路由器的峰值也达到了4700包/秒。，并且可能会稍微低于最大以太网数据包速率(约14,880个数据包/秒)。

6.3. 为什么在4.2BSD模型中会出现livelock

4.2BSD遵循4.1节中描述的模型，如图6-2所示。设备驱动程序运行在中断优先级(IPL) = SPLIMP，IP层通过软件中断运行在IPL = SPLNET，这比SPLIMP低。驱动程序和IP代码之间的队列被命名为“ipintrq”，每个输出接口都由自己的队列缓冲。所有队列都有长度限制;多余的数据包将被丢弃。该系统中的设备驱动程序实现了中断批处理，因此在高输入速率下，实际上很少中断。

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

为什么系统会受到接收活锁的影响，现在已经很明显了。一旦输入速率超过了设备驱动程序从接口取出新数据包并将其添加到IP输入队列的速率，IP代码就永远不会运行。因此，它永远不会从它的队列(ipintrq)中删除包，它会填满，所有后续接收到的包都会被丢弃。

系统的CPU资源已经饱和，因为在提高IPL时，在投入了大量CPU时间之后，它会丢弃每个数据包。这是愚蠢的;一旦一个数据包通过了设备驱动程序，它就代表着一项投资，并且应该在可能的情况下被处理到完成。在路由器中，这意味着数据包应该在输出接口上传输。当系统过载时，它应该尽早(即在接收接口)丢弃数据包，这样丢弃的数据包就不会浪费任何资源。

6.4. 修复活锁问题

我们解决活锁问题的方法是尽可能多地在内核线程中工作，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软件中断(或线程分派)1。一旦我们决定从接收接口获取一个数据包，我们就尽量不要在以后丢弃它，因为这将代表浪费精力。

我们还尝试仔细地“安排”在这个线程中完成的工作。可能不可能使用系统的真正调度器来控制对每个数据包的处理，所以我们让这个线程使用轮询技术来有效地模拟分组处理的轮询调度。轮询线程使用额外的启发式方法来帮助实现我们的性能目标。

在新系统中，接口驱动程序的中断处理程序几乎不做任何工作。相反，它简单地调度轮询线程(如果它还没有被调度)，记录它对分组处理的需求，然后从中断中返回。它不会设置设备的interrupt-enable标志，因此在轮询线程处理完所有待处理的数据包之前，系统不会因附加中断而分心。

在启动时，修改后的接口驱动程序将自己注册到轮询系统，provid-

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

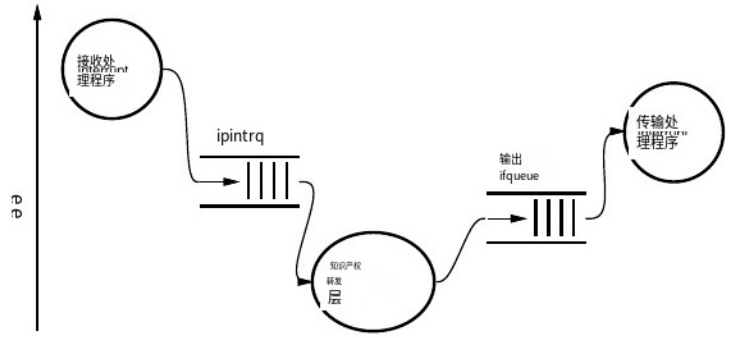

图6-2 4.2BSD IP转发路径

Ing接收和发送数据包的回调过程，以及使能中断。当轮询线程被调度时，它检查所有注册的设备，看看它们是否请求处理，并调用适当的回调程序来做中断处理程序在未修改的内核中会做的事情。

接收到的数据包回调过程直接调用IP输入处理例程，而不是将接收到的数据包放在队列中等待稍后处理;这意味着从接口接收的任何数据包都尽可能地进行处理(例如，到输出接口队列进行转发，或到队列进行交付到进程)。如果系统落后了，接口的输入缓冲区会在一段时间内吸收数据包，任何多余的数据包都会在系统在它上面浪费任何资源之前被接口丢弃。

轮询线程给回调程序分配了允许它们处理的数据包数量的配额。回调程序一旦用完配额，就必须返回到轮询线程。这允许线程在多个接口之间轮转，以及在任何给定接口上的输入和输出处理之间轮转，以防止单个输入流垄断CPU。

一旦某个接口上待处理的所有数据包都被处理完，轮询线程还会调用驱动程序的interrupt-enable回调函数，以便后续的数据包事件会导致中断。

6.5. 结果和分析

图6-3总结了我们在不使用*screend*时的变化结果。显示了几种不同的内核配置，在图中使用不同的标记符号。修改后的内核(用方形标记表示)略微改善了MLFRR，并避免了在较高输入速率下的活动锁。

修改后的内核可以被配置成好像它是一个未修改的系统(用开圆显示)，尽管这似乎比实际未修改的系统(实心圆)执行得稍差。原因尚不清楚，但可能涉及稍长的代码路径，不同的编译器，或不幸的指令缓存冲突的变化。

6.6. 调度启发式

图6-3显示，如果轮询线程对回调过程可以处理的数据包数量没有设置配额，当输入速率超过MLFRR时，总吞吐量几乎下降到零(图中菱形部分所示)。发生此活动锁的原因是，尽管数据包不再在IP输入队列中丢弃，但它们仍然在输出接口队列中堆积(并被丢弃)。这个队列是不可避免的，因为不能保证输出接口的运行速度和输入接口一样快。

为什么系统无法榨干输出队列?如果数据包到达得太快，输入处理回调函数永远不会完成它的工作。这意味着轮询线程永远不会调用发送接口的输出处理回调，这阻止了发送器缓冲区描述符的释放，以便在进一步的数据包传输中使用。这类似于4.4节中确定的传输饥饿状态。

在无配额修改的内核中，结果实际上更糟，因为在该系统中，由于输出队列上的空间不足而丢弃数据包，而不是丢弃IP队列上的空间不足。未修改的内核对每个被丢弃的数据包所做的工作更少，因此偶尔会以足够快的速度丢弃它们，赶上输入数据包的突发。

6.6.1. 来自满队列的反馈

使用*screend*程序时，修改后的系统表现如何?图6-4比较了未修改的内核(实心圆)和几个修改后的内核的性能。

按照目前所描述的修改内核(正方形)，系统的性能大约和

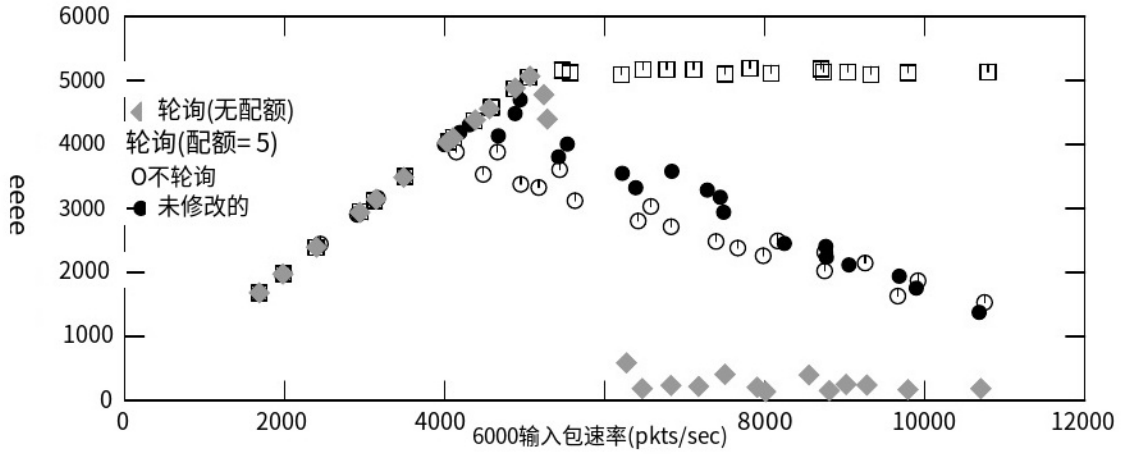

图6-3:修改后内核的转发性能，不使用screen

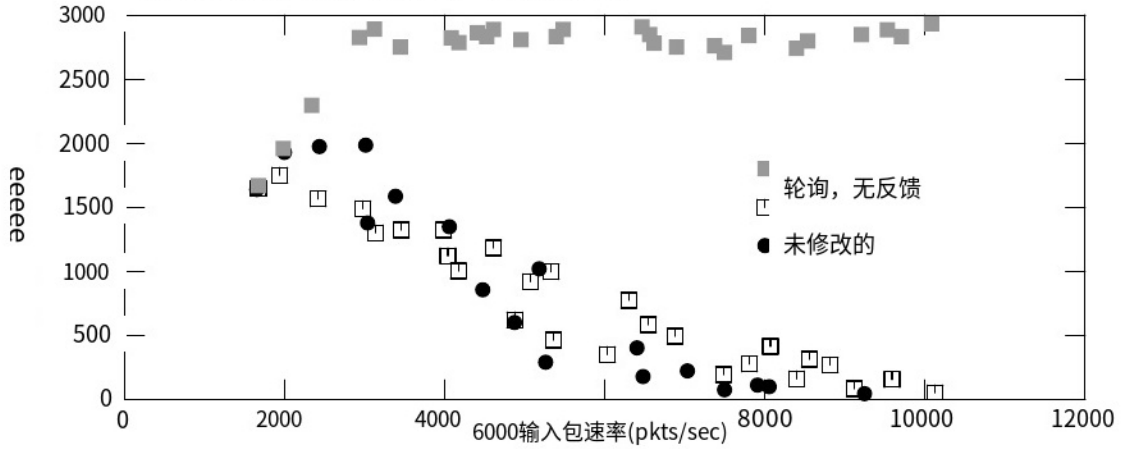

图6-4:修改后的内核的转发性能，带有*screend*

修改的内核。问题是，因为*screend*是在用户模式下运行的，内核必须将数据包排队交付给*screend*。当系统过载时，这个队列就会被填满，数据包就会被丢弃。*Screend*永远没有机会运行来清空这个队列，因为系统将其周期用于处理输入数据包。

为了解决这个问题，我们检测筛选队列何时满，并抑制进一步的输入处理(和输入中断)，直到有更多的队列空间可用。结果用图6-4中的灰色正方形标记显示:没有活锁，峰值吞吐量得到了很大提高。来自队列状态的反馈意味着系统正确地分配CPU资源，以便在整个系统中移动数据包，而不是在中间点丢弃它们。

在这些实验中，轮询配额为10个数据包，筛选队列被限制为32个数据包，当队列满75%时，我们抑制了输入处理。当筛选队列满25%时，重新启用输入处理。我们随意选择了这些高水位和低水位标记，一些调整可能会有所帮助。我们还设置了一个超时时间(随机选择为一个时钟滴答，或大约1毫秒)，在重新启用输入后，以防*屏幕程序挂*起，这样给其他消费者的数据包就不会无限期地丢失。

同样的队列状态反馈技术可以应用于系统中的其他队列，如接口输出队列、包过滤队列(用于网络监控)[9,8]等。这些队列的反馈策略将更加复杂，因为可能很难确定输入处理负载是否实际上阻碍了这些队列的进展。然而，*由于屏幕*程序通常作为系统上唯一的应用程序运行，一个完整的筛选队列是一个明确的信号，表明有太多的数据包到达。

6.6.2. 包数配额的选择

为了避免非屏幕配置中的活锁，我们必须设置每个回调处理的数据包数量的配额，因此我们调查了配额变化时系统吞吐量的变化情况。图6-5显示了结果;较小的配额发挥作用

更好。随着配额的增加，活锁问题越来越严重。

然而，当使用*screend*时，队列状态反馈机制会阻止活锁，小配额会略微降低最大吞吐量(约5%)。我们认为，通过每个回调处理更多的数据包，系统可以更有效地摊销轮询的成本，但增加配额也可能增加最坏情况下的每个数据包延迟。一旦配额大到足以用突发的数据包填满筛选队列，反馈机制可能会隐藏任何改进的潜力。

图6-6展示了使用*screend*过程时的结果。

总之，使用和不使用*screend*的测试表明，对于所测试的硬件配置，10到20个数据包之间的配额可以产生稳定和接近最佳的行为。对于其他cpu和网络接口，正确的值可能不同，因此这个参数应该是可调的。

7. 保证用户级进程的进度

6.4节中描述的轮询和队列状态反馈机制可以确保分组处理的所有必要阶段都取得进展，即使在输入过载期间也是如此。但是，它们对其他活动的需求漠不关心，因此用户级进程仍然需要CPU周期。这使得系统的用户界面无响应，并干扰了内务任务(如路由表维护)。

我们通过在我们修改后的路由器上运行一个计算绑定进程来验证这种效果，然后用要转发的最小大小的数据包淹没路由器。路由器以全速率转发数据包(即，好像没有用户态进程在消耗资源)，但用户进程没有取得可衡量的进展。

由于根本问题是数据包输入处理子系统占用了太多的CPU，因此我们应该能够通过简单地测量处理接收到的数据包所花费的CPU时间，并在超过阈值时禁用输入处理来改善这一问题。

Alpha架构，我们在其上做了这些实验，包括一个高分辨率的低开销计数器寄存器。这个寄存器计算每个指令周期(在目前的实现中)，可以在一条指令中读取，不会有任何数据缓存缺失。其他现代RISC架构也支持类似的计数器;众所周知，英特尔的Pentium有一个不支持的功能。

我们在一个定义为几个时钟节拍的时间段内测量CPU使用情况(在我们当前的实现中，可以任意选择10毫秒，以匹配调度器的量程)。每个周期一次，计时器函数会清除数据包处理代码中使用的CPU周期的运行总数。

每次我们修改过的内核开始轮询循环时，它都会读取循环计数器，并在循环结束时再次读取该计数器，以测量在循环期间处理输入和输出数据包所花费的周期数。(配额机制确保这个间隔是相对较短的。)然后将这个数字加到运行总数中，如果这个总数超过阈值，则立即抑制输入处理。在当前周期结束时，计时器会重新启用输入处理。系统的空闲线程的执行也会重新启用输入中断并清除运行总数。

通过将阈值调整为一个周期内总周期数的一小部分，可以相当精确地控制用于处理数据包的CPU时间量。我们还没有实现这个控制的编程接口;对于我们的测试，我们只是修补了一个表示分配给网络处理的百分比的内核全局变量，内核自动将此转换为若干周期。

图7-1显示了在周期阈值的几种设置和不同的输入速率下，计算密集型用户进程可用的CPU时间。随着输入速率的增加，曲线显示出相当稳定的行为，但是用户进程获得的CPU时间并不像阈值设置所暗示的那样多。

部分差异来自于系统开销;即使没有输入负载，用户进程也能获得大约94%的CPU周期。此外，限环机制抑制了数据包输入处理，但不抑制输出处理。在较高的输入速率下，在抑制输入之前，输出队列会填充到足以吸收额外的CPU周期。

测量误差可能导致一些额外的差异。只有在处理突发的输入数据包(对于这些实验，回调配额是5个数据包)后才会检查周期阈值。在系统转发约5000包/秒的情况下，处理这样的突发大约需要1 msec，约为阈值检查周期的10%。

50%和75%阈值曲线上的初始下降可能反映了处理实际中断的成本;这些周期不计入阈值，并且在输入速率低于饱和时，每个传入的分组可能被处理得足够快，以至于没有中断批处理发生。

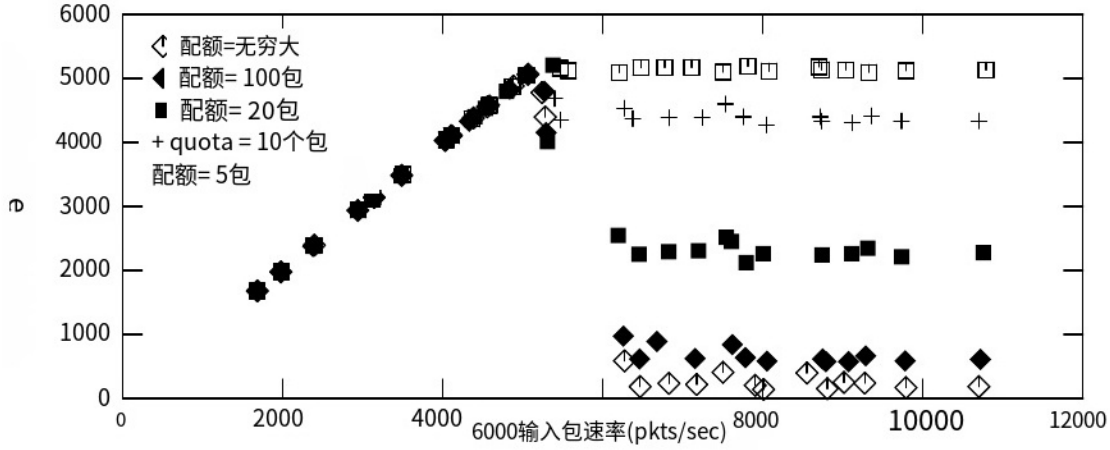

图6-5:包数配额对性能的影响，无屏幕显示

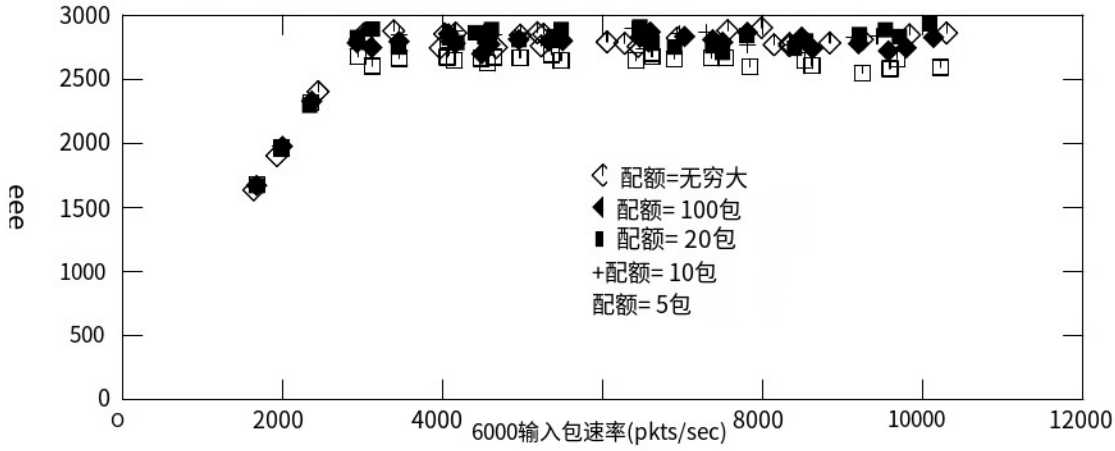

图6-6:包数配额对性能的影响，带screen

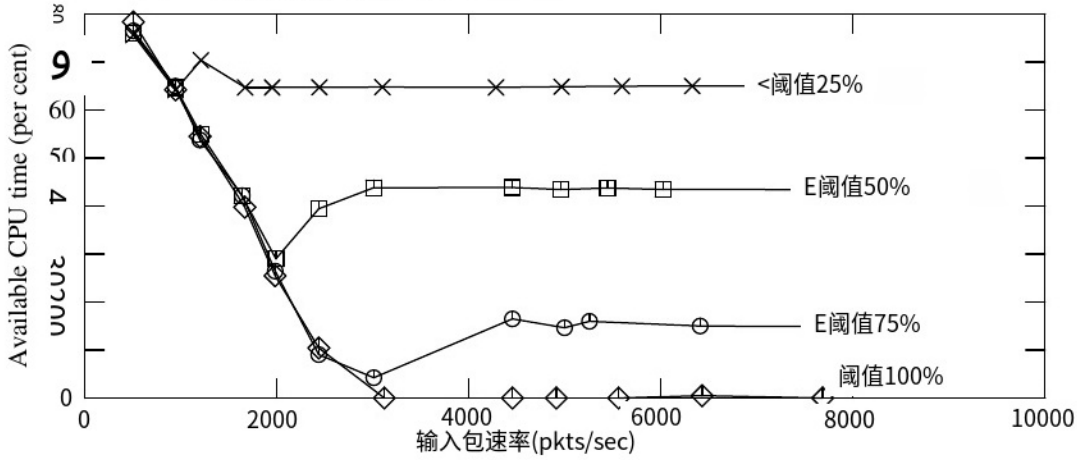

图7-1:使用循环限制机制的用户模式CPU可用时间

通过对分组处理施加循环限制，系统主观上要响应得快得多，即使是在输入过载的情况下。然而，这种改进对于本地用户来说是很明显的;任何基于网络的交互，如Telnet，仍然受到影响，因为许多数据包被丢弃。

7.1. 端系统传输协议的性能

我们对内核所做的更改可能会影响终端系统传输协议的性能，例如TCP和UDP/RPC/XDR/NFS堆栈。因为我们还没有将我们的修改应用于高速网络接口驱动程序，例如

作为FDDI的一个，我们还不能测量这种效果。(测试系统很容易使以太网饱和，因此测量以太网上的TCP吞吐量没有任何影响。)

将接收到的数据包直接从设备驱动程序处理到TCP层，而不将数据包放在ip级队列上的技术被Van Jacobson专门用于提高TCP性能[4]。通过避免队列操作和任何相关的锁定，它应该减少接收数据包的成本;它还应该改善内核到内核交互的延迟(例如TCP确认和NFS rpc)。

轮询接口的技术不应该降低端系统的性能，因为它主要是在输入过载期间进行的。(一些实现使用轮询来完全避免传输中断[6]。)在过载期间，未修改的系统将不会在应用程序或传输协议上取得任何进展;轮询、队列状态反馈和CPU周期限制的使用应该给修改后的系统一个至少取得一些进展的机会。

8. 相关工作

轮询机制之前已经在基于unix的系统中使用过，无论是在网络代码中还是在其他上下文中。然而我们已经使用轮询来提供公平性和保证进度，先前轮询的应用旨在减少与中断服务相关的开销。这确实减少了系统过载的机会(对于给定的输入速率)，但不能防止活锁。

Traw和Smith[14, 16]描述了“时钟中断”的使用，定期轮询来学习到达的数据包，而不需要每个数据包中断的开销。他们指出，很难选择合适的轮询频率:过高，系统将所有时间都花在轮询上;过低，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，他们确实暗示了使用中断触发轮询其他事件的方案的可能性。

如果最近的输入速率增加到某个阈值以上，4.3BSD操作系统[5]显然使用了一种周期性轮询技术来处理从八端口终端接口接收到的字符。其意图似乎是为了避免丢失输入字符(设备几乎没有可用的缓冲)，但人们可以将此视为一种活锁避免策略。一些路由器实现使用轮询作为它们调度包处理的主要方式。

当一个拥塞的路由器必须丢弃一个数据包时，它对丢弃哪个数据包的选择会有很大的影响。我们的修改并不影响丢*弃哪些*分组;我们只*在它们被丢弃时*改变。政策过去是，现在仍然是“甩尾”;其他政策可能会提供更好的结果[3]。

我们在改进接口驱动算法方面的一些初步工作在[1]中描述。

9. 总结和结论

在接收过载下表现不佳的系统无法提供一致的性能和良好的交互行为。活锁从来都不是应对过载的最佳方案。在这篇文章中，我们展示了如何理解系统过载行为以及如何通过在完成分组处理时进行仔细的调度来改进它。

通过对UNIX系统的测量，我们已经表明，传统的中断驱动系统在过载情况下表现很差，导致接收活锁和传输饥饿。因为这样的系统随着数据包进入系统的深入而逐渐降低处理数据包的优先级，当过载时，它们表现出过多的数据包丢失和工作浪费。这种病态不仅可能是由长期的接收过载引起的，也可能是由短期突发到达的瞬时过载引起的。

我们描述了一套有助于解决过载行为不良问题的调度改进。这些包括:

· 限制中断到达率，以减轻过载
轮询以提供公平性
· 处理收到的数据包以完成
明确规范分组处理的CPU使用

我们的实验表明，这些调度机制提供了良好的过载行为，并消除了接收活锁。它们对专用系统和通用系统都有帮助。

致谢。

我们从许多人那里得到了测量和理解系统性能的帮助，包括Bill Hawe, Tony Lauck, John Poulin, Uttam Shikarpur 和 John Dustin。Venkata Padmanabhan、David Cherkus 和 Jeffry Yaplee 在手稿准备期间提供了帮助。

K. K. Ramakrishnan在这篇论文上的大部分工作是在他还是Digital Equipment Corporation的雇员时完成的。

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 Cal-ifornia, 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 Communica-tions 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 于 1979 年 获 得 the massachusetts - massachusetts Institute of Technology的学士学位，1980年和1986年获得Stanford University的硕士学位和博士学位。自1986年以来，他一直是Digital的西方研究实验室的研究员，致力于高性能计算机系统的网络和操作系统问题。他是多个互联网标准的作者或合著者，*《网络互联:研究与经验》的副主编*，并担任1994年冬季USENIX会议的项目主席。

通信地址:数字设备

公司西方研究实验室，大学大道250号，

加州帕洛阿尔托94301 (mogul@wrl.dec.com)

K. K. Ramakrishnan是Technical的成员

AT&T贝尔实验室的工作人员。他拥有理学士学位

1976年毕业于印度班加罗尔大学，获硕士学位

1978年从印度科学研究所毕业，a

1983年获得马里兰大学博士学位。联合国-

1994年之前，他是Digital公司的咨询工程师。

Ramakrishnan的研究兴趣在性能方面

计算机网络算法的分析与设计

工作与分布式系统。

他是技术人员

*IEEE网络杂志*编辑，也是成员之一

互联网研究任务组的End-End成员

研究小组。

地址 为 通信: 美国电话电报公司(AT&T)

实验室，600山大道，默里山，新泽西州，

07974 (kkrama@research.att.com)