分　数：	
评卷人：	

華中科技大學

研究生（数据中心技术）课程论文（报告）

题 目：基于改善数据中心尾延迟问题技术研究综述

学　　号 M202273726

姓　　名 罗承辰

专　　业 计算机技术

课程指导教师 施展、童薇、胡燏翀

院（系、所） 计算机科学与技术学院

2023 年 1 月 10 日

基于改善数据中心尾延迟问题技术研究综述

罗承辰

(华中科技大学计算机科学与技术学院 武汉 430074)

摘 要 数据中心对低延迟操作的需求持续增长。一个关键的驱动因素是存储、计算和内存在网络上的分解，以节省成本。随着分解的进行，需要低延迟的信息传递来挖掘下一代存储的潜力。低延迟的消息传递，对于数据中心网络（DCN）中的许多应用也是至关重要的，这些应用通常是面向用户的，即使流量完成时间存在一个非常小的延迟也会降低应用性能，降低用户体验，并造成经济损失。为了解决这种长尾延迟问题，学者采用了各种策略，包括速率控制，无损网络，到主动传输。然而，这些建议中的大多数都引入了非实质性的实施困难，如全局状态监测，复杂的网络控制和交换机的修改。所以有学者采用了带有主动冗余传输的FEC。还有一些学者使用内核旁路网络来改善内核网络堆栈导致的尾延迟。本文主要介绍了导致数据中心网络长尾延迟的原因以及改善长尾延迟的优化策略，重点介绍了HPCC 、Swift 、PCN 、Aeolus、Cloudburst 、Shenango 和Perséphone，指出了现有方案的优势和面临的缺陷。HPCC利用INT的精确链路负载形成来计算精确的流速更新。Swift通过测量端到端RTT 来调节数据包中的拥塞窗口。PCN 重新构建了无损以太网的拥塞管理,与PFC协调工作。Aeolus是一个专注于”预信用”数据包传输的主动式传输方案。Cloudburst 采用了带有主动冗余传输的FEC。Shenango 是一个通过工作窃取提高CPU利用率的内核旁路调度器。Perséphone 是一个新的内核旁路调度器，实现了应用程序感知的DARC 策略。最后总结了全文，并探讨了未来的研究方向。

关键词 数据中心；尾延迟；速率控制；无损网络；主动传输；前向纠错码

中图法分类号 TP DOI号:1

A review of research based on techniques to cut tail latency in datacenters

LUO Cheng-Chen

(School of Computer Science and Technology,Huazhong University of Science and Technology, Wuhan 430074)

Abstract

The demand for low-latency operations in datacenters continues to grow. A key driver is the disaggregation of storage, compute and memory over the network to save costs. As disaggregation proceeds, low latency messaging is needed to unlock the potential of next-generation storage. Low latency messaging is also critical for many applications in data center networks (DCNs), which are often user-facing, where even a very small delay in traffic completion time can degrade application performance, degrade the user experience, and cause financial loss. To address this long tail latency problem, scholars have employed various strategies ranging from rate control, lossless networks, to proactive transport. However, most of these proposals introduce non-substantial implementation difficulties, such as global state monitoring, complex network control, and switch modifications. So some scholars have adopted FEC with proactive redundant transport. Others use kernel-bypass networks to cut tail latency due to kernel network stack. This paper introduced the causes of long tail latency in datacenter networks and optimization strategies to cut long tail latency, focused on HPCC, Swift, PCN, Aeolus, Cloudburst, Shenango, and Perséphone and pointed out the advantages and drawbacks faced by existing schemes. Swift regulates the congestion window in packets by measuring the end-to-end RTT. PCN reconstructs congestion management for lossless Ethernet, working in coordination with PFC. Shenango is a kernel-bypass scheduler that improves CPU utilization through work stealing and Perséphone is a new kernel-bypass scheduler that implements application-aware DARC policies. Finally, the full paper was summarized and future research directions were discussed.

Keywords datacenter; tail latency; rate control; lossless network; proactive transport; FEC

1 引言

数据中心对低延迟操作的需求持续增长。一个关键的驱动因素是存储、计算和内存在网络上的分解，以节省成本。随着分解的进行，需要低延迟的信息传递来挖掘下一代存储的潜力。例如，行业的最佳实践要求在100k+IOPS下有100μs的访问延迟，以有效地使用闪存。即将推出的NVMe要求在100万以上的IOPS 有10μs的延时。否则昂贵的服务器在等待I/O时就会闲置。

低延迟的消息传递，对于数据中心网络（DCN）中的许多应用是至关重要的，如网络搜索，页面创建，推荐系统，流处理，和在线广告。这些应用通常是面向用户的，即使流量完成时间存在一个非常小的延迟也会降低应用性能，降低用户体验，并造成经济损失。

然而，由于多种原因，长尾延迟问题在数据中心网络中特别突出，主要是由于以下原因：

(1)高扇入突发：基于树形的应用，每一个父层都聚合了来自其子层的结果。同一父层的子层同一时间作出反应，导致父层的扇入突发。这些同步的扇入突发超过了交换机的出口流量，导致队列堆积并可能导致拥挤的数据包丢弃。

(2)交换机的共享缓冲区太浅：共享缓冲区通常被配置为吸收突发，即一部分缓冲区在多个端口之间共享，因此如果一个端口遇到突发，它可以占用所有可用的共享缓冲区。然而，交换机的缓冲区太浅，不能吸收来自应用程序的扇入突发事件。商品交换机中缓冲器的扩展（从4MB到12MB）与每端口带宽10倍以上的增长（从1Gbps到10/40Gbps）不兼容。此外，吞吐量密集型流量需要交换机中一定量的缓冲来实现高吞吐量，因此需要保证在每个端口都有缓冲。当突发发生时，来自延迟敏感流量的数据包可能由于缺乏缓冲区而被丢弃。

(3)错误处理和重传超时：应用程序使用TCP的自动重传请求（ARQ）错误处理（如回退N帧协议（GBN）和基于超时的错误发现）可靠地传递数据。对于每个错误，ARQ至少需要一个往返时延（RTT）来恢复，因此它对长流量很有效，因为对于大的拥塞窗口来说，收到或错过的数据包的确认是分批的。然而，ARQ 对短流量没有帮助，短流量可能在慢速启动阶段完成。如果一个流量的初始窗口数据包与拥塞重合并被丢弃，它只能通过超时重传机制（RTO）发现。因此，设置一个合适的重传超时时间，是快速恢复短流量的关键。数据中心网络中的重传超时时间通常被设置为5ms,这几乎比一般的往返时延（100μs）大两个数量级。一次超时很容易导致长尾延迟问题。缩短重传超时时间肯定会有利于数据包的恢复。但由于频繁的重传，也可能增加网络和服务器的负载，并且需要高精度的计时器。

(4)硬件故障：数据包丢失也可能由于硬件故障而发生，即使是在设计良好的现代数据中心网络与无损结构中也会发生。这种故障可能来自三态内容寻址存储器（TCAM）的不足、收发器的老化等。特别对于无声的数据包掉落，是通过全网诊断工具发现的。因此，这些故障对于单路径传输来说很难恢复，而对于多路径传输来说可能需要多个往返时延才能恢复。

(5)不完善的流量负载平衡（LB）：目前数据中心网络中的负载平衡通常依赖于流量哈希算法，以保持并行路径的均匀利用率。然而，哈希碰撞可能会发生，这可能会暂时使一些链路过载，导致队列长度增加，延长每包延迟并诱发数据包掉落。当所有流量都有相同的目的端口时，随机化负载平衡也不能帮助解决应用突发问题。

现在的高速网络提供几微秒的往返时间（RTTs）。然而，当应用程序在当前的操作系统和网络堆栈上运行时，延迟是以毫秒计的。因此，内核的网络堆栈也是导致数据中心长尾延迟的一个原因。

为了解决这种长尾延迟问题，学者采用了各种策略，包括速率控制[1][2]，无损网络[3]，到主动传输[4]。然而，这些建议中的大多数都引入了非实质性的实施困难，如全局状态监测，复杂的网络控制和交换机的修改。虽然这些方案实现了很好的性能，但是它们很难在现有的数据中心网络中实施。所以有学者采用了带有主动冗余传输的FEC[5]，改善长尾延迟问题的同时易于实施。还有一些学者使用内核旁路网络[6][7]来改善内核网络堆栈导致的尾延迟。

本文主要介绍了导致数据中心网络长尾延迟的原因以及改善长尾延迟的优化策略，重点介绍了HPCC[1]、Swift[2]、PCN[3]、Aeolus[4]、Cloudburst[5]、Shenango[6]和Perséphone[7]。本文安排如下，第二章介绍了改善长尾延迟优化策略的原理和优势，第三章对几种优化策略进行了介绍和对比，最后一章总结了全文并对未来的研究方向进行了介绍。

2 原理和优势

本章在第一章介绍的长尾延迟问题的优化方案的基础上详细地介绍各个优化方案的原理和优势。改善长尾延迟的策略包括速率控制、无损网络、主动传输、带有主动冗余传输的FEC以及内核旁路网络。

（1）速率控制：根据ECN、延迟或带内网络遥测（INT）[1]信号来控制流量的速率，我们可以

控制队列的建立，从而降低对延迟敏感的流量在交换机上的排队延迟。

在传统网络中缺乏细粒度的网络负载信息。ECN是终端主机可以从交换机得到的唯一反馈，而RTT是一个没有交换机参与的纯端到端测量。然而，这种情况最近发生了变化。随着带内网络遥测（INT）功能在新的交换ASICs中的应用，获得细粒度的网络负载信息并利用它来改进拥塞控制已经在生产网络中成为可能。HPCC[1]的关键思想是利用INT的精确链路负载形成来计算精确的流速更新。在大多数情况下，HPCC只需要一个流速更新步骤。使用来自INT 的精确信息使HPCC能够解决当前CC方案中的三个限制。首先，HPCC发送者可以快速提高流速以提高利用率，或降低流速以避免拥堵。第二，HPCC发送者可以快速调整流速，以保持每个链路的输入速率略低于链路的容量，防止排队。在保持较高的链路利用率的同时，也避免了链路的堆积。最后，由于发送率是根据交换机的直接测量结果精确计算的，HPCC只需要3个独立的参数，用于调整公平和效率。

HPCC[1]使用来自交换机的显式反馈来保持网络队列的长度和RPC的完成时间。它们可以提供良好的性能，但在大的持续时间和IOPS密集型的工作负载下，它们没有帮助。特别是，主机上的拥塞堆积是一个没有解决的实际问题。与交换机的紧密协调也使部署性和可维护性变得复杂。Swift[2]建立在主机上，独立地根据端到端延迟调整速率。当我们使用NIC时间戳准确地测量延迟并仔细地推理目标时，这种设计可以实现高水平的性能，并且具有许多其他优点。延迟很好地对应于我们寻求满足的更高级别的服务级别目标(SLOs)。它整齐地分解为网络和主机部分，分别响应不同的拥塞原因。在数据中心中，可以很容易地针对不同的路径和竞争流调整延迟目标。使用延迟作为信号可以让我们部署新一代交换机，而无需考虑功能或配置，因为延迟总是可用的，就像经典TCP的数据包丢失一样。

（2）无损网络：由数据包丢失引起的重新传输很容易导致延迟增加。为了防止网络拥堵而导致数据包丢失，无损网络已经成为数据中心和集群计算系统的一个有吸引力的趋势。为了保证无损，并在这种不同的流量模式下提供令人满意的工作完成时间，拥塞管理成为无损以太网的关键和挑战。IEEE DCB规定了一个CM的框架，包括两个基本功能，包括端到端拥堵控制和逐跳流量控制。端到端拥塞控制根据测量变量反映的拥塞信息，如交换机队列长度或RTT，主动调节源发送速率。代表性的解决方案包括由IEEE开发的QCN802.1Qau和DCQCN，以及基于RTT的方案TIMELY。尽管这些协议可以约束交换队列的长度，并相应地减少丢包率，但并不能充分保证零丢包。实际上，不可控制的突发事件可能在信息源意识到网络拥堵之前就已经丢失了，特别是当拥堵控制环路延迟比较大或者突发程度和并发性比较重的时候。

为了避免由于不可控制的突发造成的数据包丢失，IEEE802.1Qbb定义了基于优先级的流量控制（PFC），以确保无损失。通过PFC，当入口队列长度超过某个阈值时，交换机向其上游设备（交换机或网卡）发送一个PAUSE帧以停止传输。而当队列长度低于另一个阈值时，就会发送一个RESUME帧。尽管PFC可以保证由于网络拥堵造成的零丢包，但它会导致一些性能问题，如不公平甚至死锁。

为了解决无损以太网部署在数据中心的性能问题，有学者发现性能问题的根本原因是无损以太网的拥塞管理架构中的无效因素，包括不恰当的拥塞检测机制和不适当的速率调整法，并提出了PCN[3]，它由两个基本组成部分组成。(1)一种新的拥堵检测和识别机制，以识别哪些流量是真正造成拥堵的；(2)一种接收器驱动的速率调整方法，以缓解短至1RTT的拥堵。

（3）主动传输：在生产环境下，60-90%的流量可以在一个RTT内完成。因此，在每一个RTT上保持低延迟和高吞吐量对传输至关重要。因此，传统的尝试和回退传输（如HPCC[1]、Swift[2]）不适合这些要求，因为它们只在事后对拥塞信号（如INT或延迟）做出反应，并且需要多轮才能收敛到正确的速率。虽然他们可以为长流量保持良好的平均性能，但很难在每一轮中达到正确的速率，这对小流量和尾延迟至关重要。

主动传输以”请求和分配”的方式运行，带宽被作为”信用”主动分配给发送者，然后发送者可以以合适的速率发送”预定的数据包”，以确保高链路利用率、低延迟和零数据包丢失。主动传输的关键概念是在活动流量中明确分配瓶颈链路的带宽，并主动防止拥堵。主动式传输的卓越性能的核心是对活动流量进行完美的信用分配，因此任何新的发送方需要一个RTT，我们称之为预信用阶段，以通知接收器/控制器分配信用。

”预信用”阶段可以在高链路速度下承载大量的流量，但现有的主动式解决方案都没有适当地处理它。Aeolus[4]是一个专注于”预信用”数据包传输的解决方案，作为主动式传输的构建块。Aeolus包含非常规的设计原则，如预定包优先（SPF），它取消了第一RTT包的优先权，而不是像优先工作那样对它们进行优先处理。它进一步利用了主动传输的保留、确定的性质，作为有效恢复丢失的第一RTT数据包的一种手段。

（4）带有主动冗余传输的FEC：现有的改善尾延迟的方案通常是对短流量不友好的。

基于带内网络遥测（INT）[1]或延迟[2]信号来控制速率的方案仍然依赖于负载平衡，当流量分布不均时，短流量可能会遭受长延迟。此外，先进的INT信号甚至可能在交换机上无法使用，因此需要定制硬件。

无损网络[3]的解决方案在大规模的情况下，由于错误的配置或硬件故障，数据包丢失仍然可能发生。

主动传输的解决方案需要至少一个RTT来分配信用给一个新的流量，这对短流量来说是不可接受的。虽然最近的方案[4] 启用线速启动，对于短流量来说，问题仍然存在。例如，在最后一个RTT终止之前，很难给发送者分配合适的信用。对于短流量来说，过大的信用会导致链路利用率不足，而过小的信用额度会引入大的延迟。

在DCN中，可以采用容错功能来处理数据包丢失。也就是说，我们不需要重传丢失的数据包，同时仍然保证可靠的传输任务。Cloudburst[4]采用了带有主动和遗忘冗余传输的FEC。每个编码的数据包包含多个原始数据包的信息。这样一来，即使一些编码的数据包丢失了，原始数据包仍然可以在接收器处被重构。Cloudburst在多条路径上传播编码的数据包，这就无意识地利用了现代DCN中丰富的路径多样性。如果存在任何无拥堵的路径，Cloudburst将利用它们，而没有额外的信号开销。

（5）旁路内核网络： 由于当今内核的高开销，实现微秒级延迟的最佳解决方案是内核旁路网络，它将CPU内核用于应用程序的自旋查询网卡。但这种方法浪费了CPU，即使在适度的平均负载下，也必须为预期的峰值负载提供足够的内核。

Shenango[6]实现了可比的延迟，但CPU效率要高得多。它以非常精细的粒度在各个应用之间重新分配内核,使延迟敏感应用的未使用的周期能够被批处理应用有效地使用。

最近的内核旁路调度器通过工作窃取[6]提高了利用率，但这些技术只适用于均匀和轻度的工作负载。Perséphone[7]是一个内核旁路的操作系统调度器，旨在最大限度地减少微秒级的应用程序的尾延迟，并表现出广泛的服务时间分布。Perséphone集成了一个新的调度策略，即动态应用感知保留核心（DARC），它为时间短的请求保留核心。与现有的内核旁路调度器不同，DARC并不持续工作。DARC对应用请求进行分析，当队列中没有短的请求时，会留下少量的内核空闲，所以当短的请求到达时，它们不会被运行时间长的请求阻塞。反其道而行之，让核心闲置让DARC在更高的利用率下保持较低的尾延迟，减少为相同工作负载服务所需的核心总数，从而更好地利用数据中心资源。

3 研究进展

3.1 HPCC[1]

在传统网络中缺乏细粒度的网络负载信息。ECN是终端主机可以从交换机得到的唯一反馈，而RTT是一个没有交换机参与的纯端到端测量。然而，这种情况最近发生了变化。随着带内网络遥测（INT）功能在新的交换ASICs中的应用，获得细粒度的网络负载信息并利用它来改进拥塞控制已经在生产网络中成为可能。HPCC[1]的关键思想是利用INT的精确链路负载形成来计算精确的流速更新。在大多数情况下，HPCC只需要一个流速更新步骤。使用来自INT 的精确信息使HPCC能够解决当前CC方案中的三个限制。首先，HPCC发送者可以快速提高流速以提高利用率，或降低流速以避免拥堵。第二，HPCC发送者可以快速调整流速，以保持每个链路的输入速率略低于链路的容量，防止排队。在保持较高的链路利用率的同时，也避免了链路的堆积。最后，由于发送率是根据交换机的直接测量结果精确计算的，HPCC只需要3个独立的参数，用于调整公平和效率。

HPCC是一个由发送方驱动的CC框架。如图1所示，发送方发送的每个数据包都会得到接收方的确认。在数据包从发送方传播到接收方的过程中，路径上的每个交换机利用其交换ASIC的INT功能，插入一些元数据，报告数据包出口的当前负载，包括时间戳（ts）、队列长度（qLen）、传输字节（txBytes）和链接带宽容量（B）。当接收方得到数据包时，它将所有由交换机记录的元数据复制到它发回给发送方的ACK消息中。发送方决定每次收到带有网络负载信息的ACK时如何调整其流速。

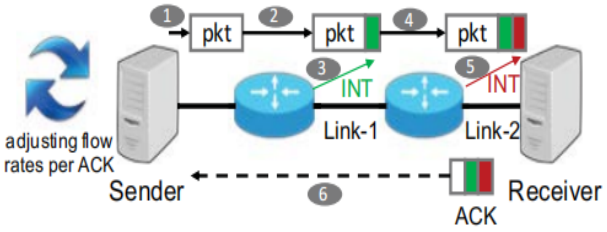


图1 HPCC框架

HPCC是一种基于窗口的CC方案，它控制着飞行中的字节数。飞行字节指的是已经发送的数据量，但在发送方还没有确认。与控制速率相比，

控制机上字节有一个重要的优势。在没有拥堵的情况下，机上字节和速率是可以互换的，公式为$inflight = rate \times T$，其中T 是基础传播RTT。然而，控制机上字节在拥堵期间，对机内字节的控制大大改善对延迟反馈的容忍度。与纯粹的基于速率的CC方案相比，它在反馈到来之前不断地发送数据包，对飞行字节的控制确保了飞行字节的数量在一定范围内,使得发送者在达到限制时立即停止发送，无论反馈被延迟多长时间。因此，整个网络得到了极大的稳定。

发送者通过发送窗口限制机上字节：每个发送方都有一个发送窗口，它限制了它可以发送的飞行字节数。使用窗口是TCP的一个标准想法，但在数据中心对反馈延迟的容忍度有很大的好处，因为排队延迟可能比超低的基本RTT高几个数量级。初始发送窗口的大小应设置为使流量能够以线速度开始，所以我们使用$W_{init} = B_{NIC} \times T$，其中$B_{NIC}$是NIC带宽。除了窗口之外，我们还将数据包的发送速度调整,以避免突发流量。网卡中一般都有数据包加速器。步调速率为$R = \frac{W}{T}$，即一个窗口大小为W在一个基本RTT为T的网络中可以实现的速率。

基于机上字节的拥塞信号和控制法：除了发送窗口外，HPCC的拥塞信号和控制法也是基于机上字节的。飞行字节直接对应于链接的利用率。具体来说，对于一条链路，飞行字节数是所有穿越该链路的流量的总飞行字节数。假设一条链路的带宽是B，穿越它的第i个流有一个窗口大小W_i。这条链接的飞行字节数为$I = \sum W_i$。因此，我们的目标是将I控制在略小于$B \times T$的范围内，这样就不会出现拥堵和排队现象。

估计每条链路的机内字节数:第一个问题是发送方如何使用INT信息来估计其路径上每个链接j的I_j。具体来说，机上字节包括队列和管道中的数据包。因此，对于每个链接j，我们使用其队列长度（qlen）和输出速率（txRate）来估计I_j，如公式（1）。

$$I_j = qlen + txRate \times T \tag{1}$$

其中qlen直接来自INT。txRate是用txBytes和ts计算的。$txRate = \frac{ack_1.txBytes - ack_2.txBytes}{ack_1.ts - ack_1.ts}$，其中$ack_1$和$ack_2$是两个ACKs。$txRate \times T$估计管道中的字节数。公式(1)假设所有流量具有相同的已知基本RTT。这在数据中心是可能的，由于拓扑结构的规律性，大多数服务器对之间的RTT非常接近。

对信号做出反应。每个发送方应调整其窗口，使其流量路径上的每个链路j的I_j略低于$B_j \times T$。具体来说，是$\eta \times B_j \times T$（η是一个接近1的常数，例如95%）。因此，对于链接j，每个发送者可以乘法减少其窗口的系数$k_j = \frac{I_j}{\eta \times B_j \times T} = U_J/\eta$,其中$U_j$是标准化的链接j的机上字节：

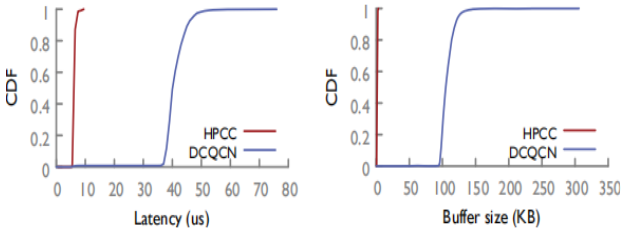
$$U_j = \frac{I_j}{B_j \times T} = \frac{qlenT}{B_j \times T} + \frac{txRate_j}{B_j} \tag{2}$$

发送方i应该对最拥挤的链接做出反应:

$$W_i = \frac{W_i}{max_j(k_j)} + W_A I = \frac{W_i}{max_j(U_j)/\eta} + W_A I \tag{3}$$

其中$W_A I$是确保公平性的而增加的部分，非常小。注意公式（3）中的第一项是一个MIMD项。这将利用率控制与公平性控制解耦，以确保发送者迅速抢占空闲带宽或避免拥堵。

实验结果表明，HPCC具有更低的网络延迟。我们通过一条被两个大流量饱和的链路持续发送小流量（每个1KB），并测量小流量的延迟和缓冲区大小。图2e和2f显示HPCC保持一个接近零的队列，因此小流量的延迟接近5.4 μs，即基本RTT。DCQCN在ECN标记阈值附近保持一个常设队列，因此延迟一直高于35 μs。



(e) *Elephant-Mice* latency　　(f) *Elephant-Mice* queue size

图2　HPCC的延迟性能

3.2　Swift[2]

HPCC使用来自交换机的显式反馈来保持网络队列的长度和RPC的完成时间。它们可以提供良好的性能，但在大的持续时间和IOPS密集型的工作负载下，它们没有帮助。特别是，主机上的拥塞堆积是一个没有解决的实际问题。与交换机的紧密协调也使部署性和可维护性变得复杂。Swift[2]建立在主机上，独立地根据端到端延迟调整速率。当我们使用NIC时间戳准确地测量延迟并仔细地推理目标时，这种设计可以实现高水平的性能，并且具有许多其他优点。延迟很好地对应于我们寻求满足的更高级别的服务级别目标(SLOs)。它整齐地分解为网络和主机部分，分别响应不同的拥塞原因。在数据中心中，可以很容易地针对不同的路径和竞争流调整延迟目标。使用延迟作为信号可以让我们部署新一代交换机，而无需考虑功能或配置，因为延迟总是可用的，就像经典TCP的数据包丢失一样。

Swift通过测量端到端RTT来调节数据包中的

拥堵窗口，采用加法-增法-减法（AIMD）算法，目的是将延迟维持在目标延迟周围。Swift将端到端RTT分解为网卡到网卡（结构）和终端延迟部分，以分别应对网络与主机/网卡的拥堵。

Swift是在Pony Express中实现的，Pony Express是一个网络堆栈，在Snap中提供自定义的可靠传输实例。它使用网卡以及软件时间戳来进行精确的RTT测量。它使用Pony Express来实现CPU的高效运行和低延迟。图3显示了Swift在Pony Express中的位置。Pony Express提供command和completion queue API：应用程序向Pony Express提交命令，也被称为"Ops"，并接收完成。Ops映射到网络流，而Swift管理每个流的传输速率。

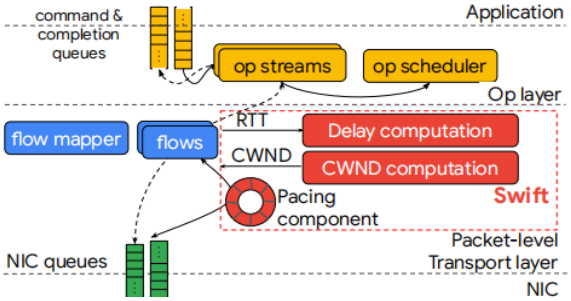


图3 在PonyExpress架构的背景下，Swift作为一个包级拥塞控制。

延迟是Swift中的主要拥塞信号，因为它符合我们所有的要求。TIMELY指出，RTT可以用现代硬件精确测量，而且它提供了一个多比特的拥塞信号，即它编码了拥塞的程度，而不仅仅是其存在。Swift进一步分解了端到端RTT，将网络与主机分开。通过结合网卡硬件中的时间戳和Pony Express等基于轮询的传输，使得测量更精确。

组成RTT的延迟:图4(a)显示了构成RTT的组成部分，从本地发送数据包到接收相应远程端点的确认帧。

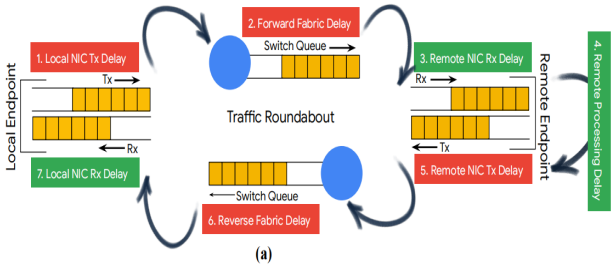


图4 端到端RTT的组成部分

本地NIC Tx 延迟是指数据包在网络上发出之前在NIC Tx队列中所花费的时间。当网络栈使用拉动模型时，主机在网卡准备好发送数据包时将其交给网卡，所以这个延迟可以忽略不计。

前向网络延迟是数据包在源和目的地之间的交换机上的序列化、传播和排队延迟之和。它还包括网卡的序列化延迟。

远程NIC Rx 延迟是指数据包在被远程栈选中之前在远程NIC 队列中停留的时间。当主机是瓶颈时，这个延迟可能很重要。例如，在Snap的背景下，当数据包处理能力因内存预设和CPU调度而下降时，这个延迟会迅速增加。

远程处理延迟是指网络栈处理数据包和生成ACK包的时间，包括任何ACK延迟。

远程NIC Tx 延迟是指ACK 数据包在NIC Tx 队列中花费的时间。

反向网络延迟是指ACK包在反向路径上花费的时间。注意，正向和反向路径可能不是对称的。

本地NIC Rx 延迟是指ACK包在被网络栈处理以标记数据包的交付之前所花费的时间。

简单的目标延迟窗口控制：核心的Swift算法是一个简单的AIMD控制器，基于测量的延迟是否超过目标延迟。我们发现，随着TIMELY向Swift的演进，简单是一种美德，并消除了一些复杂性，例如，通过使用RTT和目标延迟之间的差异而不是RTT梯度。下面我们描述了基于固定目标延迟的算法。

控制器在收到ACK数据包时被触发。Swift通过使用瞬时延迟，而不是最小或低通滤波延迟，对拥堵做出快速反应。此外，Swift不明确地延迟ACK。这两种选择都缓解了使用延迟作为拥塞信号的呆滞性问题。算法1中的第4-14行提供了Swift在收到ACK时的反应；如果延迟小于目标延迟，cwnd（以包为单位）将增加$\frac{ai}{cwnd}$(ai=加法增量)，这样，RTT累计增幅等于ai。否则，cwnd将被乘法减少，减少的幅度取决于延迟离目标有多远，也就是说，我们使用延迟信号的多个比特进行精确控制。乘法减少被限制在每个RTT一次，这样Swift就不会对同一个拥堵事件做出多次反应。我们通过检查最后一次cwnd减少的时间来做到这一点。cwnd的初始值在我们的设置中影响不大，因为Pony Express保持了长期的流量。

结果显示，Swift在目标附近实现了低延迟。图5显示了整个数据中心的网卡到网卡的往返时间（或结构RTT），由网卡时间戳测量。我们可以看到，Swift能够将网络的平均往返时间保持在配置的目标延迟附近，并且对尾延迟的控制比基于非延迟的GCN好得多。这些延迟按照基本目标延迟进行归一化处理。

在图6中，Swift在大规模和整个集群中实现了接近目标延迟的平均RTT。平均RTT 与部署中使用

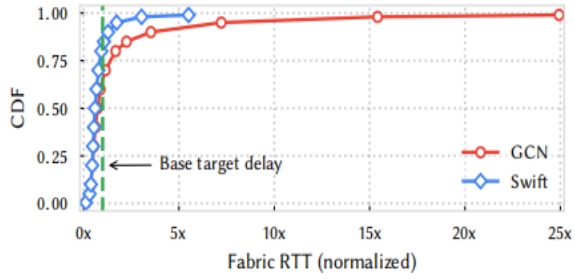


图5　Fabric RTT：Swift比GCN更严格地控制网络延迟。

的基本目标延迟大致相符。这一行为被证明是非常有用的，因为在全面部署的过程中调整了目标延迟–早期的Swift部署使用的基本目标延迟是当前配置的2倍。这种变化是逐步进行的，而且很谨慎，确保在把握延迟-吞吐量权衡的过程中，不会因为减少应用获得的带宽而导致应用性能的退步。

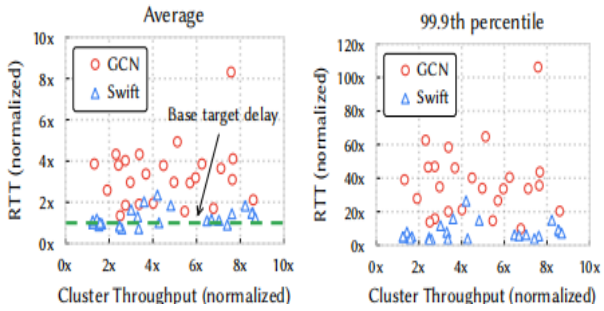


图6　集群Swift/GCN的吞吐量与平均RTT。虚线是基本目标延迟（归一化为1）。

3.3　PCN[3]

由数据包丢失引起的重新传输很容易导致延迟增加。为了防止网络拥堵而导致数据包丢失，无损网络已经成为数据中心和集群计算系统的一个有吸引力的趋势。为了保证无损，并在这种不同的流量模式下提供令人满意的工作完成时间，拥塞管理成为无损以太网的关键和挑战。IEEE DCB规定了一个CM的框架，包括两个基本功能，包括端到端拥塞控制和逐跳流量控制。端到端拥塞控制根据测量变量反映的拥塞信息，如交换机队列长度或RTT，主动调节源发送速率。代表性的解决方案包括由IEEE开发的QCN802.1Qau和DCQCN，以及基于RTT的方案TIMELY。尽管这些协议可以约束交换队列的长度，并相应地减少丢包率，但并不能充分保证零丢包。实际上，不可控制的突发事件可能在信息源意识到网络拥堵之前就已经丢失了，特别是当拥塞控制环路延迟比较大或者突发程度和并发性比较重的时候。

为了避免由于不可控制的突发造成的数据包

Algorithm 1: SWIFT REACTION TO CONGESTION

Parameters: ai: additive increment, β: multiplicative decrease constant, max_mdf: maximum multiplicative decrease factor
$cwnd\_prev \leftarrow cwnd$
$bool\ can\_decrease \leftarrow (now - t\_last\_decrease \geq rtt)$ ▷ Enforces MD once every RTT
On Receiving ACK
  $retransmit\_cnt \leftarrow 0$
  $target\_delay \leftarrow$ TargetDelay() ▷ See S3.5
  if $delay < target\_delay$ **then** ▷ Additive Increase (AI)
    if $cwnd \geq 1$ **then**
      $cwnd \leftarrow cwnd + \frac{ai}{cwnd} \cdot num\_acked$
    else
      $cwnd \leftarrow cwnd + ai \cdot num\_acked$
  else ▷ Multiplicative Decrease (MD)
    if $can\_decrease$ **then**
      $cwnd \leftarrow \max(1 - \beta \cdot (\frac{delay - target\_delay}{delay}), 1 - max\_mdf) \cdot cwnd$
On Retransmit Timeout
  $retransmit\_cnt \leftarrow retransmit\_cnt + 1$
  if $retransmit\_cnt \geq RETX\_RESET\_THRESHOLD$ **then**
    $cwnd \leftarrow min\_cwnd$
  else
    if $can\_decrease$ **then**
      $cwnd \leftarrow (1 - max\_mdf) \cdot cwnd$
On Fast Recovery
  $retransmit\_cnt \leftarrow 0$
  if $can\_decrease$ **then**
    $cwnd \leftarrow (1 - max\_mdf) \cdot cwnd$
$cwnd \leftarrow$ clamp$(min_cwnd, cwnd, max_cwnd)$ ▷ Enforce lower/upper bounds
if $cwnd \leq cwnd\_prev$ **then**
  $t\_last\_decrease \leftarrow now$
if $cwnd < 1$ **then**
  $pacing\_delay \leftarrow \frac{rtt}{cwnd}$
else
  $pacing\_delay \leftarrow 0$;
Output: $cwnd, pacing\_delay$

丢失，IEEE802.1Qbb定义了基于优先级的流量控制（PFC），以确保无损失。通过PFC，当入口队列长度超过某个阈值时，交换机向其上游设备（交换机或网卡）发送一个PAUSE帧以停止传输。而当队列长度低于另一个阈值时，就会发送一个RESUME帧。尽管PFC可以保证由于网络拥堵造成的零丢包，但它会导致一些性能问题，如不公平甚至死锁。

导致无损以太网部署在数据中心的性能问题的根本原因是无损以太网的拥塞管理架构中的无效因素，包括不恰当的拥塞检测机制和不适当的速率调整法。PCN[3]重新构建了无损以太网的拥塞管理，它被设计成一种基于速率的、端到端的拥塞控制机制，与PFC协调工作。如图7所示，PCN由三

部分组成：反应点（RP）、拥堵点（CP）和通知点（NP）。一般来说，CP总是指拥堵的交换机，使用非停止ECN（NP-ECN）方法标记通过的数据包，以检测出口端口是否处于真正的拥堵状态。用NP-ECN标记的数据包并不肯定意味着处于拥塞状态，它需要NP做出最终决定。NP，即接收方，识别拥堵的流量，计算其接收率，并定期向RP发送拥堵通知包（CNP）。RP，是发送者的NIC，根据CNP 中的信息调整每个流量的发送速率。

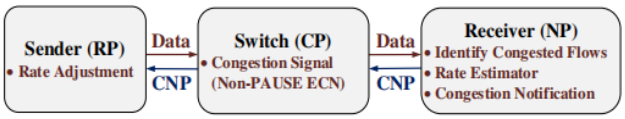


图7 PCN框架。

CP算法：NP-ECN方法被开发用于检测拥堵并生成拥堵信号。CP算法遵循图8中的状态机。假设当一台交换机的一个出口端口收到来自其下游节点的RESUME帧时，在相关的等待队列中有N个数据包。NP-ECN将设置其计数器PN=N。然后端口重新开始传输数据包。每传输一个数据包，计数器就减一，直到所有的N个暂停的数据包都被传输完毕。对于这N个数据包，它们将不会被标记。而对于后来没有暂停的数据包，相应的PN=0，交换机将以传统的方法对其进行ECN标记，阈值为0。

这样一来，真正拥堵的出口端口的所有数据包都会被标记为ECN。相反，在非拥堵端口的数据包永远不会被标记为ECN。而对于准拥堵端口，暂停的数据包不会被标记为ECN。同时，当入口端口的队列不是空的时候，到达和离开RESUME状态的端口的数据包被标记为ECN，也就是说，准拥堵端口的数据包被部分标记为ECN。在PCN中，CP只用于标记数据包的拥塞信号，让NP节点最终确定流量是否拥塞。

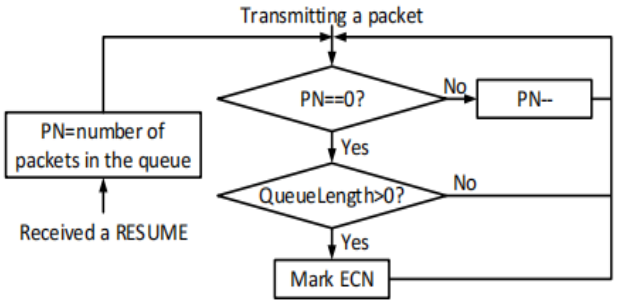


图8 CP状态机。

NP算法：NP的功能包括识别拥塞流，估计接收率和定期发送拥塞通知包（CNP）。T表示CNP生成周期。

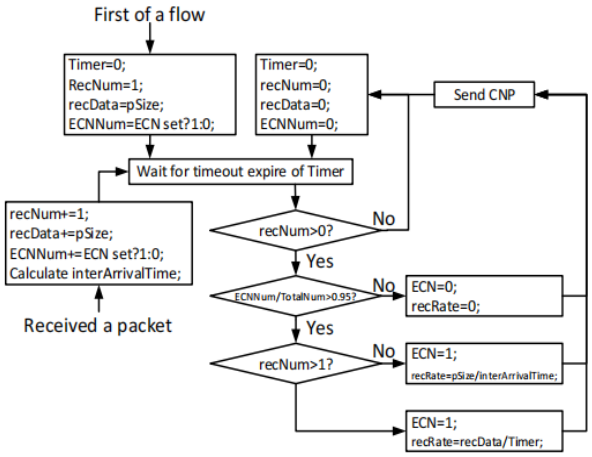


图9 NP状态机。

识别拥塞流量：NP根据NP-ECN机制所标记的ECN信号来识别拥堵流。如果在CNP生成周期T中重新收到的95%的数据包被标记为ECN，则该流量被认为是拥塞的。95% 这个值是根据经验设定的，以过滤实践中一些微小的干扰，如队列振荡和优先级安排，这使得一个或多个真正拥挤的流量的数据包不可能被标记为ECN。

估计接收率：接收率直接用到达的数据包的总大小除以T来计算。值得注意的是，流量的接收率可能非常小，以至于在几个CNP生成期内只有一个数据包到达。为了解决这种特殊情况，PCN也记录了数据包在NP的到达时间。当到达间时间大于T时，NP通过用到达时间代替T来估计接收率。

生成CNPs：NP发送CNP以通知流量的源头在T周期内的接收速率，该周期与DCQCN类似，被设定为50μs。此外，当流量需要降低或增加速率时，PCN明确地生成CNP，当在周期T内没有收到任何数据包时，也不会生成CNP。具体来说，CNP数据包的格式与RoCEv2中的CNP数据包兼容。CNP封装的主要信息包括IPv4/IPv6头中的1位ECN和保留段中的32位RecRate，保留段中包含了以1Mbps为标准的接收速率。NP算法的状态机总结于图9。

RP算法：算法1描述了RP如何根据CNP的信息调整发送速率的伪代码。在开始时，流量以线性速率开始，以改善短流量的流量完成时间（FCT）。

速率下降：当RP收到一个带有ECNmarked的CNP时，它按照第6行的规则降低发送速率，这样就可以消耗掉交换机中已建立的队列。

速率上升：当RP收到没有ECNmarked的CNP时，它将按照第10和11行

的规则调整发送速率。具体来说，RP通过计算其当前值和线性速率的加权平均值提高发送率。

Algorithm 1 PCN RP Algorithm.

1: $sendRate \leftarrow lineRate$
2: $w \leftarrow w_{min}$
3: **repeat** per CNP (CE, $recRate$)
4: **if** $CE == 1$ **then**
5: (CNP *notifies rate decrease*)
6: $sendRate \leftarrow \min\{sendRate,\ recRate \cdot (1 - w_{\min})\}$
7: $w \leftarrow w_{\min}$
8: **else**
9: (CNP *notifies rate increase*)
10: $sendRate \leftarrow sendRate \cdot (1 - w) + lineRate \cdot w$
11: $w \leftarrow w \cdot (1 - w) + w_{\max} \cdot w$
12: **until** End of flow

来自不同主机的平均和99百分位数FCTs被画在图10的右边子图中。底部的实心条表示平均FCT，上部的条纹条表示99百分位数的值。很明显，对于所有类型的主机，PCN比QCN、DCQCN和TIMELY表现得更好。

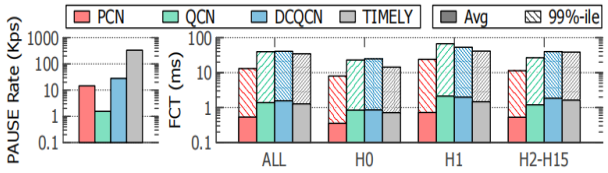
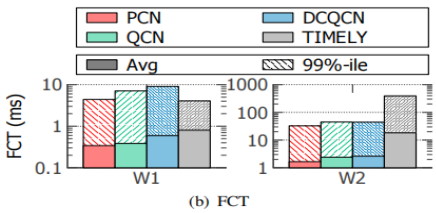


图10　并发突发下的PAUSE生成率和FCT。

在图11(b)中，画出了所有流量的统计FCT。PCN比QCN的99百分位FCT低了1.6倍，这是因为QCN极大地降低了大流量的发送速率，以至于网络变得严重欠载。由于PCN可以快速检测到拥堵点，并对拥堵流量的速率进行调整，短流量的排队延迟很低，并能快速完成。这可以改善整体的FCT。与DCQCN和TIMELY相比，平均FCT降低了了1.75倍和2.35 倍。



(b) FCT

图11　现实工作负载的性能。

3.4　Aeolus[4]

在生产环境下，60-90%的流量可以在一个RTT内完成。因此，在每一个RTT上保持低延迟和高吞吐量对传输至关重要。因此，传统的尝试和回退传输（如HPCC[1]、Swift[2]）不适合这些要求，因为它们只在事后对拥塞信号（如INT或延迟）做出反应，并且需要多轮才能收敛到正确的速率。虽然他们可以为长流量保持良好的平均性能，但很难在每一轮中达到正确的速率，这对小流量和尾延迟至关重要。

主动传输以"请求和分配"的方式运行，带宽被作为"信用"主动分配给发送者，然后发送者可以以合适的速率发送"预定的数据包"，以确保高链路利用率、低延迟和零数据包丢失。主动传输的关键概念是在活动流量中明确分配瓶颈链路的带宽，并主动防止拥堵。主动式传输的卓越性能的核心是对活动流量进行完美的信用分配，因此任何新的发送方需要一个RTT，我们称之为预信用阶段，以通知接收器/控制器分配信用。

"预信用"阶段可以在高链路速度下承载大量的流量，但现有的主动式解决方案都没有适当地处理它。Aeolus[4]是一个专注于"预信用"数据包传输的解决方案，作为主动式传输的构建块。Aeolus包含非常规的设计原则，如预定包优先（SPF），它取消了第一RTT包的优先权，而不是像优先工作那样对它们进行优先处理。它进一步利用了主动传输的保留、确定的性质，作为有效恢复丢失的第一RTT数据包的一种手段。

Aeolus旨在同时实现三个设计目标。(1) 新的流量充分利用空闲带宽，如果可以的话，努力完成，避免超过必要的FCT时间。(2) 保护预定的数据包，以保持主动运输的优先级。(3)使其易于部署在生产数据中心，即Aeolus必须可以用商品交换机实现。

图12概述了Aeolus，它主要包含3个组成部分：速率控制、选择性丢弃和损失恢复。

速率控制：Aeolus在终端主机上采用最小的速率控制：所有流量都以线速开始在预信用阶段，然后根据以后收到的信用来调整他们的发送率。

选择性丢弃：Aeolus保护预定数据包的关键是在网络中执行预定数据包优先（SPF）原则。为了做到这一点，Aeolus引入了一种新颖的选择性丢弃机制。在交换机上，当带宽刚刚用完时，它会选择性地丢弃未安排的数据包，而不影响已安排的数据包。有了这样的方案，Aeolus可以有效地利用剩余的带宽来处理未安排的数据包，而不会削弱主动传输的理想特性。此外，我们的选择性丢弃机制很容易在商品交换机进行部署。

损失恢复：鉴于Aeolus已经保护了计划内的数据包，只需要对丢失的未计划内的数据包进行损失恢复。为了恢复，我们利用了受保护的主动传输,它是一种安全和高效的损失重传的手段。我们

提出了一种损失检测机制，可以准确地定位在预信用阶段的非计划包损失，并将其作为后信用阶段引起的计划包重传，只需一次。

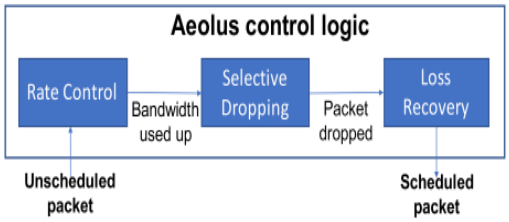
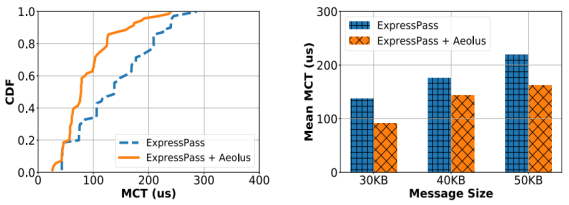


图12 Aeolus概述。

图13显示了当消息大小从30KB到50KB不等时的消息完成时间（MCT）。结果表明，Aeolus可以帮助ExpressPass加快小流量，即使是在受压的传输模式下：中位MCT在消息大小为30KB时提高了43%（图13(a)），而平均MCT为50KB。在不同的信息规模下，MCT提高了19%-33%（图13（b））。



(a) MCT with 30KB message size (b) Mean MCT with different messages sizes

图13 消息完成时间（MCT）。信息大小从30KB到50KB不等。

接下来评估真实工作负载下的性能。图14显示了大小在0到100KB之间的流量的流量完成时间（FCT）分布。我们可以看到，Aeolus明显改善了ExpressPass的FCT：在四个工作负载中，近60%、80%、28%和70%的大小为0-100KB小流量在第一个RTT内完成。

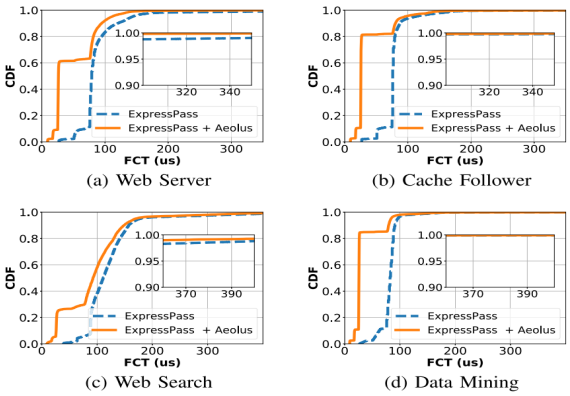
3.5 Cloudburst[5]

现有的改善尾延迟的方案通常是对短流量不友好的。

基于带内网络遥测（INT）[1]或延迟[2]信号来控制速率的方案仍然依赖于负载平衡，当流量分布不均时，短流量可能会遭受长延迟。此外，先进的INT信号甚至可能在交换机上无法使用，因此需要定制硬件。

无损网络[3]的解决方案在大规模的情况下，由于错误的配置或硬件故障，数据包丢失仍然可能发生。

主动传输的解决方案需要至少一个RTT来分配



(a) Web Server (b) Cache Follower (c) Web Search (d) Data Mining

图14 0-100KB流量在一个超额订阅的胖树拓扑中的FCT。网络核心的平均负荷为40%。

信用给一个新的流量，这对短流量来说是不可接受的。虽然最近的方案[4]启用线速启动，对于短流量来说，问题仍然存在。例如，在最后一个RTT终止之前，很难给发送者分配合适的信用。对于短流量来说，过大的信用会导致链路利用率不足，而过小的信用额度会引入大的延迟。

Cloudburst[5]，这是一个简单、有效而又易于部署的解决方案，在不引入上述复杂因素的情况下，达到类似甚至更好的效果。Cloudburst的核心是在多路径上探索前向纠错码（FEC）——它主动地在多路径上传播由信息产生的FEC编码的数据包，并在前几个到达的数据包中恢复它们。因此，Cloudburst能够无意识地利用未被充分利用的路径，从而实现低尾延迟。

Cloudburst的工作方式如图15。

用FEC对信息进行编码：前向纠错编码(FEC）已被部署在许多应用中,它使用主动和冗余的方法来容忍错误。在DCN中，可以采用容错功能来处理数据包丢失。也就是说，我们不需要重传丢失的数据包，同时仍然保证可靠的传输任务。Cloudburst探索了传输层设计的编码层面。它采用了带有主动和遗忘冗余传输的FEC。每个编码的数据包包含多个原始数据包的信息。这样一来，即使一些编码的数据包丢失了，原始数据包仍然可以在接收器处被重构。

在多条路径上突发：Cloudburst在多条路径上传播编码的数据包，这就无意识地利用了现代DCN中丰富的路径多样性。如果存在任何无拥堵的路径，Cloudburst将利用它们，而没有额外的信号开销。

分离的、大小有限的交换机队列：Cloudburst短流量的缓冲区使用量被限制到最小，这样Cloudburst数据包的每跳排队延迟就会确定地很低（小于几微秒）。这是通过

将Cloudburst短流量和其他流量分离在不同的队列中，并将Cloudburst队列的最大深度限制在一个很小的值（几个数据包）来实现的。对Cloudburst流量的缓冲区使用的这种限制也使其对其他流量友好。

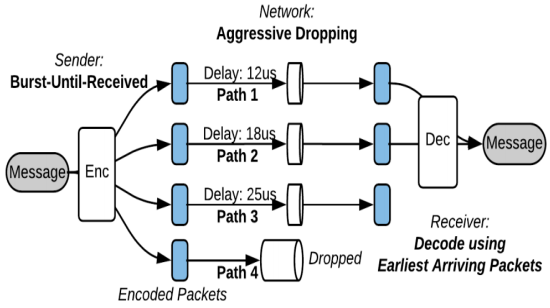


图15　Cloudburst概述

接下来详细介绍Cloudburst的设计细节。

（1）选择FEC进行编码： 图16是FEC工作原理的一个简化图示。假设一个M比特的信息由两个M/2的包A和B组成，此时再生成一个包C，c=A按位异或B，将这三个包在三条不同的路径中发送，接收方只要接收到三个包中最先到的两个包，就能解码出要发送的信息。此时，在网络中发送了3/2M比特的数据，因此编码率是3/2。在这种情况下，即使一些编码的数据包丢失，原始数据包仍然可以在接收方重建，不再需要因为数据包丢失触发重传超时机制，解决了长尾延迟的问题。

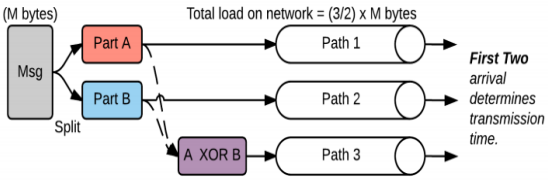


图16　3路FEC的例子

这种方案下一条路径拥塞可行，合适的编码率是3/2，我们称这种方案为固定编码率。但如果两条路径拥塞，固定编码率就有局限性。可变编码率擦除码，或称喷泉码，更适合DCN中的动态流量特性，因为其码率可适应动态信道条件，它可以从一组给定的源符号中产生无限的编码符号序列。Cloudburst中采用LT码（LTC），因为它的复杂度很低，其中一个编码包是由原始包的随机线性组合产生的。

LT Code编码过程： 图17是LT Code的编码过程。首先将原始信息打包成等长的k个包，然后从某个概率分布中选择d的值，在k个原始包中随机选择d个包进行按位异或编码，d被定义为已编码包的度。最后在不同路径发送已编码的包。

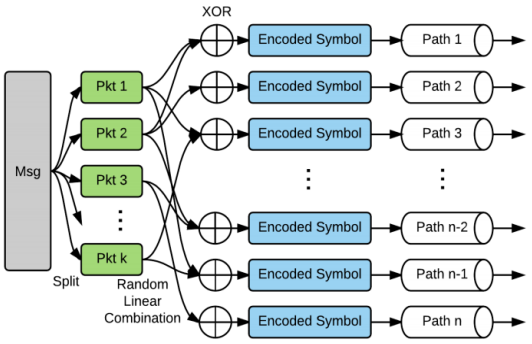


图17　一轮传输中的随机线性编码

LT Code解码过程： 解码算法使用标准的高斯消除法。我们用图18中的一个例子来说明解码的情况。在步骤1中，收到y_1，其度数为1（一个符号的度数定义为其中编码的未解码原始数据包的数量），因此相应的x_1被解码。在第二步中，收到了y_2，它的度数是2。但由于x_1已经被解码，y_2的度数减少到1，所以x_2 被解码。通过反复寻找度数为1的符号，原始信息被恢复了。

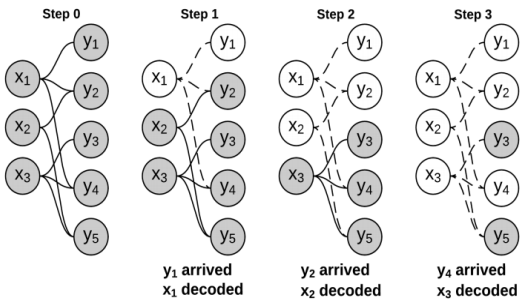


图18　解码实例

（2）突发直到接收方接收： Cloudburst的终端主机操作的顺序图见图19。我们在下文中详细说明了发送方和接收方的操作。

发送者操作： 作为喷泉码的一个变种，LT编码可以为一个给定的信息生成无尽的编码包序列。如算法1所述，LT编码使Cloudburst发送器能够连续生成编码数据包并在多条路径上发送。它只有在收到接收方发出的”停止”信号后才会停止。

接收者操作： 首先所有编码的数据包都由监听线程接收，并转发到相应的解码器。然后解码器完全恢复消息后将消息返回给接收方的监听线程。最后接收者向发送者发出STOP信号。

（3）积极丢弃： 当数据包被纠删编码时，丢掉一些并不是问题，因为信息可以通过从

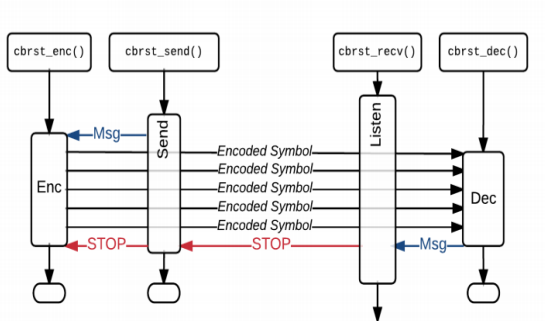


图19 终端主机操作顺序图

Algorithm 1: `cbrst_send(·)`

Input: Receiver IP & Port, Message m, Transmission timeout t, Ratio r
Output: Success/Failure

```
if sizeof(m) > MAX_MSG_SIZE then
    return Failure
Set up count down timer of t
while STOP not received do
    Wait for (1/r * NUM_PATHS * PKT_SIZE / LINK_CAP)
    foreach path p to Receiver do
        Get symbol y from encoding thread and send it on p
        if timer expires then
            return Failure
return Success
```

源头发送的数据包的子集恢复。积极丢弃通过将Cloudburst和其他流量的缓冲区分开，并将Cloudburst流量的缓冲区使用量限制到最低来实现。这可以保护其他流量不受Cloudburst流量的影响。因此，Cloudburst流量的扇入突发不再影响浅共享缓冲区交换机上的其他流量。

我们将Cloudburst与现有的可在DCN中实现的方案进行比较。如图20所示，Cloudburst在所有负载下，尾延迟都最低。

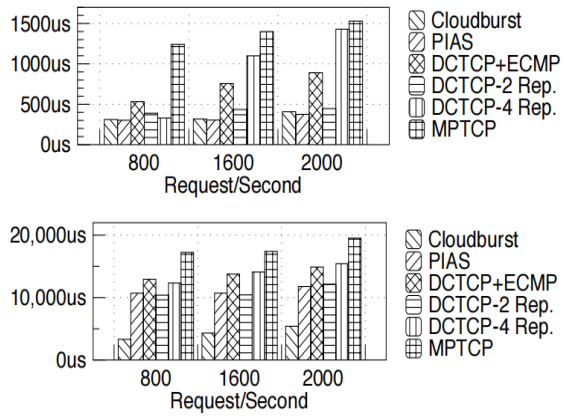


图20 平均消息完成时间和99百分位消息完成时间

图21是模拟大型DCN来补充测试平台的实验，与现有的优化方案相比，Cloudburst在所有大小的信息传输中尾延迟都最小。

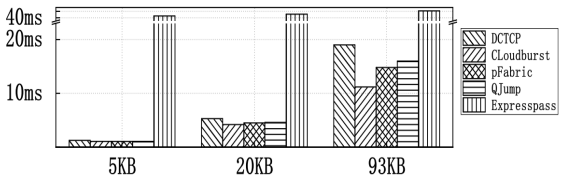


图21 99百分位消息完成时间

3.6 Shenango[6]

Shenango的目标是通过授予每个应用程序尽可能少的内核来优化CPU效率，同时避免称之为计算拥堵的情况，在这种情况下，如果不授予一个额外的内核给一个应用程序，将导致工作延迟超过几微秒。Shenango的目标释放了未被使用的内核，供其他应用程序使用，同时仍然保持了对尾延迟的控制。

而实现这一目标，主要有两大挑战。

（1）内核分配带来了开销。内核重新分配的速度最终受限于重新分配的开销，包括确定一个内核应该被重新分配、指示一个应用程序产生一个核心等等。

（2）估算所需的核心是困难的。以前的系统已经使用了应用层面的指标，如延迟、吞吐量或核心利用率来估计长时间尺度的核心需求。然而，这些指标不能在微秒级的时间间隔内应用。

Shenango通过两个关键想法来解决这些挑战。首先，Shenango将线程和数据包排队延迟视为计算拥堵的信号，并引入了一种高效的拥堵检测算法，利用这些信号来决定一个应用程序是否会从更多的核心中受益。这种算法需要对每个应用的线程和数据包队列进行细粒度、高频率的观察。

如算法1所示，拥堵检测算法根据两个负载源确定运行时间是否过载：排队的线程和排队的入口包。如果在检测算法的连续两次运行中发现有任何项目出现在队列中，就表明一个包或线程排队至少5微秒了。因为排队的数据包或线程代表了可以在另一个核心上并行处理的工作，运行时间被认为是”拥挤的”，并且IOKernel授予它一个额外的核心。排队的持续时间是一个比队列长度更稳健的信号，因为使用队列长度需要针对不同持续时间的请求仔细调整一个阈值参数。

Shenango的第二个关键想法是为一个叫做IOKernel的集中式软件实体提供一个繁忙旋转的内核。IOKernel进程以root权限运行，作为应用程序和网卡硬件队列之间的中间人。通过忙碌的旋转，IOKernel可以微秒级检查线程和数据包队列，以协调核心的分配。此外，它可以提供对网络的低延迟访问，并在软件中实现对数据包的引导，允许

Algorithm 1 Congestion Detection Algorithm

```
for each application app do
    for each active kthread k of app do
        runq ← k's runqueue
        prev_runq ← k's runq last iteration
        inq ← k's ingress packet queue
        prev_inq ← k's inq last iteration
        if runq contains threads in prev_runq or
            inq contains packets in prev_inq then
            try to allocate a core to app
            break        ▷ go to next app in outer loop
```

数据包引导规则在内核被重新分配时被快速重新配置。其结果显示，内核重新分配只需5.9微秒即可完成，并且需要不到两微秒的IOKernel计算时间来协调。这些开销支持一个核心分配率，其速度足以适应负载的变化，并迅速纠正我们的拥堵检测算法中的任何错误预测。

图22显示了Shenango架构。(a)表示用户应用程序作为单独的进程运行，并与我们的内核旁路运行时间连接。(b)表示IOKernel在一个专门的内核上运行，转发数据包并将内核分配给运行时间。(c)表示运行时间在每个内核上安排轻量级的应用线程，并使用工作窃取来平衡负载。

在测试Shenango 性能时，文章[6]主要从以下几个方面入手：(1)比较Shenango 和其他跨不同工作负载和服务时间分布的系统的延迟和CPU 效率；(2)Shenango 对突发负载的反应。

首先比较Shenango 和其他跨不同工作负载和服务时间分布的系统的延迟和CPU 效率。memcached使用USR 工作负载:请求遵循泊松到达流程，由99.8%的GET 请求和0.2%的SET 请求组成。对于Shenango，限制memcached最多使用12个超线程，因为这样memcached的性能最好。

图23显示了当我们增加提供给memcached（x轴）的负载时，memcached的99百分位延迟、memcached的中位延迟和批处理应用程序的吞吐量（y轴）如何变化。我们只显示实现的负载在提供的负载的0.1%以内的数据点。Shenango 始终保持低中位延迟和99百分位延迟，与ZygOS 相当，同时允许批处理应用程序使用未使用的周期。

然后测试Shenango 对突发负载的反应。在本实验中，以1s的模拟工作生成TCP 请求，并测量负载突然增加对尾延迟的影响。实验提供的基线负载为每秒100,000 个请求，持续1 秒，然后立即增加到一个更高的速率。在以新的速率增加一秒后，负载下降到基线。未使用的核被分配到批处理中，保持CPU 利用率在100%。

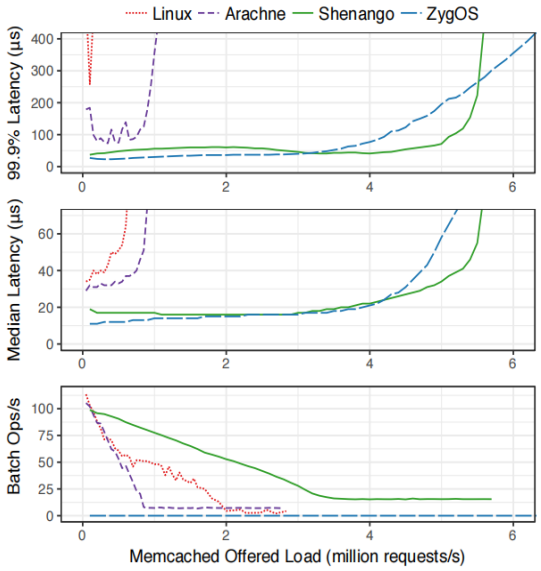


图23　Shenango 始终保持低中位延迟和99百分位延迟，与ZygOS 相当，同时允许批处理应用程序使用未使用的周期。

图24显示了Arachne和Shenango的99.9百分位的尾部延迟和吞吐量（在10毫秒的窗口内计算）。相比之下，Arachne 最终能够满足实验中所提供的负载，即每秒100 万个请求。然而，由于它的内核分配速度很慢，在负载转移之后，它可能需要超过500 毫秒的时间来添加足够的内核来适应，这导致它积累了一个等待请求的积压。因此，即使是在相对温和的负载转移之后，Arachne 也会经历几毫秒的尾延迟。相比之下，shenangoa 的响应速度非常快，几乎没有额外的延迟，即使在处理每秒10万到500 万个请求的极端负载变化时也是如此。

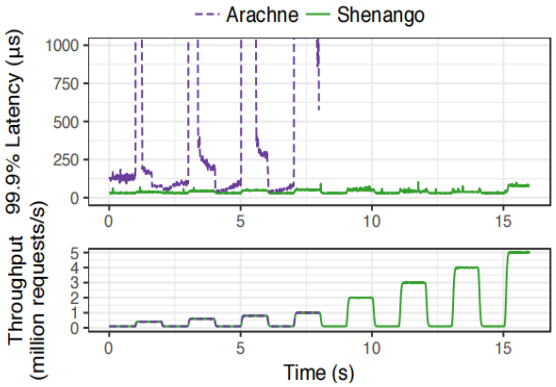


图24　Arachne和Shenango的99.9百分位的尾部延迟和吞吐量（在10毫秒的窗口内计算）

3.7　Perséphone[7]

最近的内核旁路调度器通过工作窃取[6]提高了

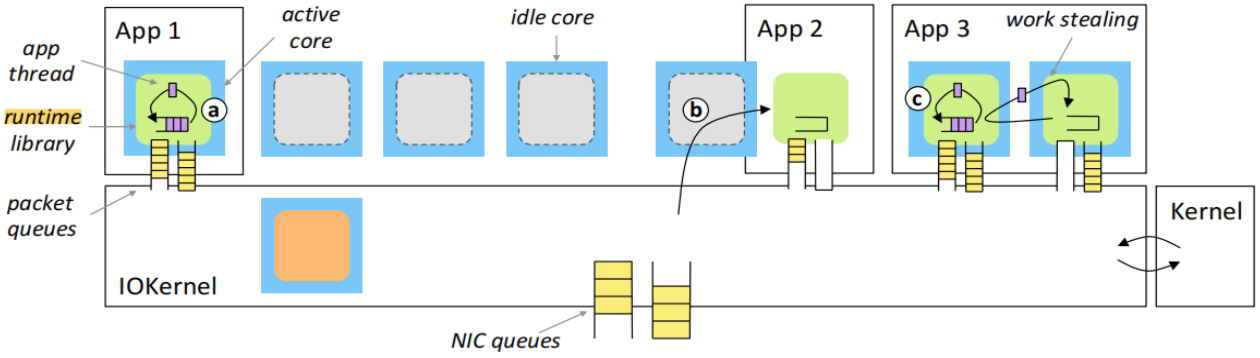


图22 Shenango架构

利用率，但这些技术只适用于均匀和轻度的工作负载。Perséphone[7]是一个内核旁路的操作系统调度器，旨在最大限度地减少微秒级的应用程序的尾延迟，并表现出广泛的服务时间分布。根据先前研究的启发，Perséphone与操作系统调度器的思路相反，它小心翼翼地让内核闲置，以防止短的请求排在任意长的请求后面，而后者通常优先考虑持续工作。实验表明，通过在最大可实现的吞吐量上做出微小的牺牲，我们可以在激进的延迟服务级别指标（SLO）下增加可实现的吞吐量，从而提高数据中心的整体CPU利用率。

为了实现这种方法，需要解决两个挑战：（1）预测每个请求类型将占用CPU多长时间；（2）在各类型之间有效地分配CPU资源，同时保留处理突发到达的能力，并尽量减少CPU浪费。Persé-phone是一个应用感知的内核旁路调度器。Per-séphone让应用程序定义请求分类器，并使用这些分类器动态地描述工作负载。使用这些配置文件，Perséphone实现了一种新的调度策略，即动态的、应用感知的预留核心（DARC），它只对短请求进行持续工作，而对长请求则不进行持续工作。DARC以较小的吞吐量代价（在我们的实验中为5%）优先处理短请求，最适合于重视微秒级响应的应用。对于其他应用，现有的内核旁路调度器工作得很好，但是我们相信有一大批数据中心的工作负载可以从DARC中受益。

Perséphone 调度器由三个部分组成，如图25 所示。这些组件作为一个事件驱动的管道运行，并按如下方式处理数据包。1.在进入路径上，网络工作端从网卡获取数据包并将其推送给调度器；2.调度器使用一个用户定义的请求分类器对传入的请求进行分类；3.调度器将分类后的请求存储在类型化队列中（针对单一请求类型的缓冲区)；4. 调度器运行DARC 的调度程序选择一个请求，并将其推送给应用程序端5.应用程序端处理请求，格式化一个响应缓冲区；6.应用程序端将指向该缓冲区的指针推到NIC；7.应用程序端通知调度器它已经完成了请求。

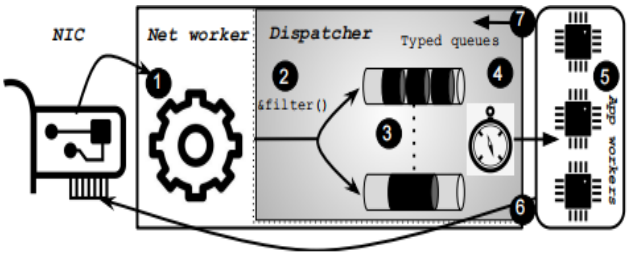


图25 Perséphone 架构

基于此架构，Shenango 实现了三个目标：(1)微秒级的端到端尾延迟和数据中心应用的高吞吐量;(2)多核机器上应用程序的cpu 高效打包;(3)应用程序开发人员的高生产率。

图26是High Bimodal 工作负载下，不同调度器在不同吞吐量下的减速目标和延迟。DARC为短的请求保留了1个核心,在100 倍离散度（High Bimodal)的工作负载下,对于20 倍的减速目标，DARC 可以分别比Shenango和Shinjuku多维持2.35 倍和1.3 倍的流量。也许更重要的是，与Shinjuku的抢占系统相比，DARC始终为长请求提供更好的尾延迟。我们还观察到Per-séphone的集中式调度对长请求提供了比Shenango 更好的性能。

在1000 倍离散度（Extreme Bimodal)的工作负载下，对于一个50倍的减速目标，Shinjuku和Perséphone都能比Shenango 多维持1.4倍的吞吐量。然而，超过55%的负载，每5微秒积极抢占的开销太大，Shinjuku开始丢弃数据包。相比之下，Perséphone保留了2个核心，以保持短请求的良好尾延迟，并能承受比Shinjuku多1.25倍的负载，同时比Shinjuku少了1.4倍的减速目标。同时，Perséphone为长请求提

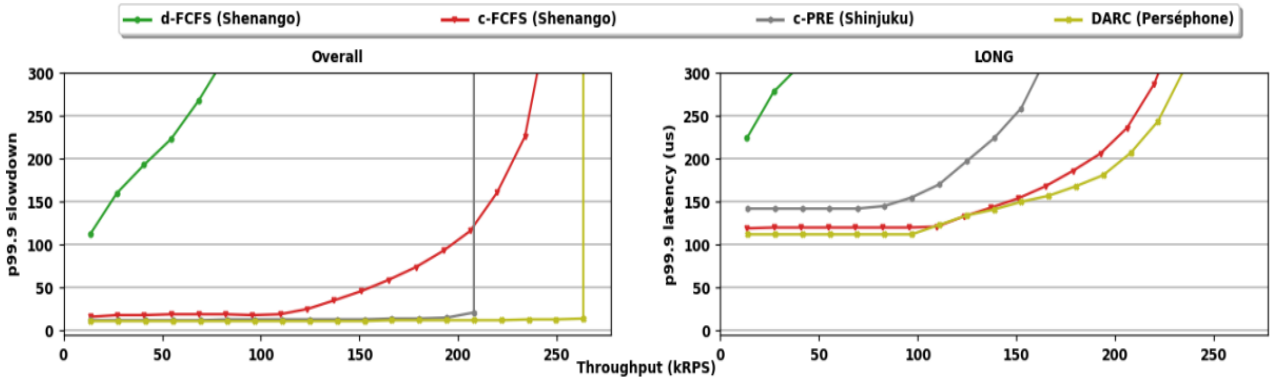


图26　High Bimodal 工作负载下，不同调度器在不同吞吐量下的减速目标和延迟

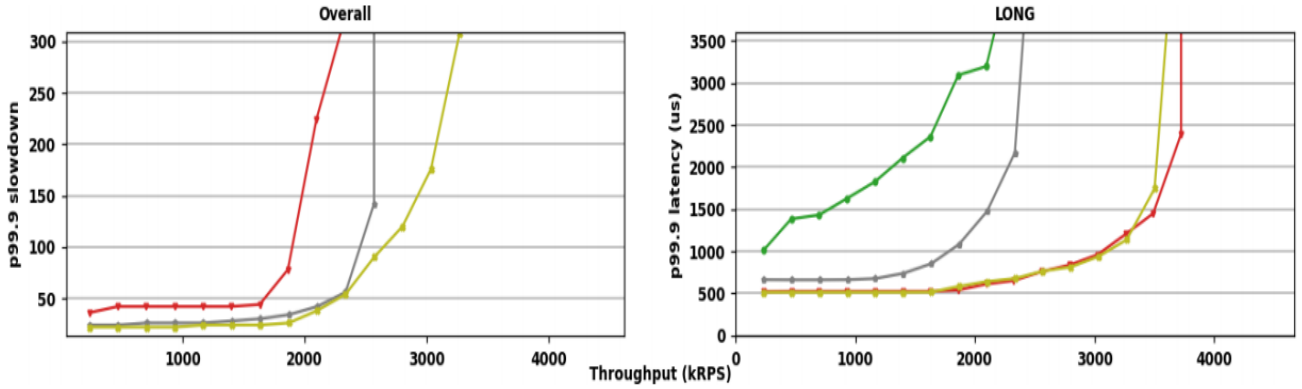


图27　Extreme Bimodal 工作负载下，不同调度器在不同吞吐量下的减速目标和延迟

供能与Shenango竞争的尾延迟。

4　总结和展望

4.1　总结

数据中心对低延迟操作的需求持续增长。一个关键的驱动因素是存储、计算和内存在网络上的分解，以节省成本。随着分解的进行，需要低延迟的信息传递来挖掘下一代存储的潜力。低延迟的消息传递，对于数据中心网络（DCN）中的许多应用也是至关重要的，这些应用通常是面向用户的，即使流量完成时间存在一个非常小的延迟也会降低应用性能，降低用户体验，并造成经济损失。

然而，由于多种原因，长尾延迟问题在数据中心网络中特别突出，主要是由于(1)高扇入突发(2)交换机的共享缓冲区太浅(3)错误处理和重传超时(4)硬件故障(5)不完善的流量负载平衡(6)内核的网络堆栈。

为了解决这种长尾延迟问题，学者采用了各种策略，包括速率控制[1][2]，无损网络[3]，到主动传输[4]。然而，这些建议中的大多数都引入了非实质性的实施困难，如全局状态监测，复杂的网络控制和交换机的修改。虽然这些方案实现了很好的性能，但是它们很难在现有的数据中心网络中实施。基于带内网络遥测（INT）[1]或延迟[2]信号来控制速率的方案仍然依赖于负载平衡，当流量分布不均时，短流量可能会遭受长延迟。此外，先进的INT信号甚至可能在交换机上无法使用，因此需要定制硬件。无损网络[3]的解决方案在大规模的情况下，由于错误的配置或硬件故障，数据包丢失仍然可能发生。主动传输的解决方案需要至少一个RTT来分配信用给一个新的流量，这对短流量来说是不可接受的。虽然最近的方案[4] 启用线速启动，对于短流量来说，问题仍然存在。所以有学者采用了带有主动冗余传输的FEC[5]，改善长尾延迟问题的同时易于实施。还有一些学者使用内核旁路网络[6][7]来改善内核网络堆栈导致的尾延迟。

本文重点介绍了HPCC[1]、Swift[2]、PCN[3]、Aeolus[4]、Cloudburst[5]、Shenango[6]和Perséphone[7]，指出了现有方案的优势和面临的缺陷。HPCC利用INT的精确链路负载形成来计算精确的流速更新。Swift通过测量端到端RTT来调节数据包中的拥堵窗口。PCN 重新构建了无损以太网的拥塞管理,与PFC 协调工作。Aeolus 是一个专注于”预信用”数据包传输的主动式传输方案。Cloudburst 采用了带有主动冗余传输

的FEC。Shenango 是一个通过工作窃取提高CPU利用率的内核旁路调度器,但只适用于均匀和轻度的工作负载。Perséphone 是一个新的内核旁路调度器，实现了应用程序感知的DARC 策略。

4.2 展望

（1）虽然Swift在可预测的延迟方面相对于现有技术水平有了很大的提高，但随着目标传输时间接近数据中心的实际传播时间，支持小于10微秒的短距离传输将需要新的技术。

（2）有可能扩展Aeolus以增强其他传输协议。例如，使用Aeolus，TCP可以从一个大的初始窗口开始，以充分利用可用的带宽，而不是从一个小的初始窗口开始，然后反复增加。Aeolus的理念也可以应用于RDMA传输，使其在没有PFC的情况下运行。Aeolus的选择性掉线机制可以用来取代PFC，以处理开始时的巨大流量突发。没有PFC，网络不再是无损的，所以需要一个更有效的损失恢复机制来处理数据包。

（3）DCN中典型的延迟敏感型应用查询通常小于10KB。因此针对此应用场景，可以进一步优化cloudburst的头开销，并进一步地选择合适的编码包的度。

（4）Shenango 硬件拓展。Shenango的评估没有考虑多插座、NUMA 机器。一个选项可能是运行多个IOKernel 实例，每个套接字一个实例。每个IOKernel 实例都可以与其他实例交换消息，可能会在套接字之间实现粗粒度的负载平衡。这样的设计将使IOKernel 能够进一步扩展。同时，IOKernel 的大部分开销是在转发数据包上，而不是在编排核心分配上。因此，其硬件负载有待进一步探索，比如新的，可以有效地向IOKernel 公开关于队列堆积信息的网卡设计。

（5）Perséphone 的网络模型有待进一步优化.目前网络工作端是一个第2 层转发器，对以太网和IP报头执行简单的检查；应用程序端处理第4层及以上的转发器，直接执行TX。该设计旨在最大化调度程序的性能，但也是Perséphone 的主要瓶颈。

致 谢 感谢施展老师、童薇老师和胡燏翀老师讲授的课程，我受益匪浅。同时也感谢施展老师能体谅我们的身体情况，为我们延迟交报告的日期。

参 考 文 献

[1] Li Y, Miao R, Liu H H, et al. HPCC: High precision congestion control[M]//Proceedings of the ACM Special Interest Group on Data Communication. 2019: 44-58.

[2] Kumar G, Dukkipati N, Jang K, et al. Swift: Delay is simple and effective for congestion control in the datacenter[C]//Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication. 2020: 514-528.

[3] Cheng W, Qian K, Jiang W, et al. Re-architecting Congestion Management in Lossless Ethernet[C]//NSDI. 2020: 19-36.

[4] Hu S, Zeng G, Bai W, et al. Aeolus: A building block for proactive transport in datacenter networks[J]. IEEE/ACM Transactions on Networking, 2021, 30(2): 542-556.

[5] Zeng G, Chen L, Yi B, et al. Cutting tail latency in commodity datacenters with cloudburst[C]//Proc. IEEE INFOCOM. 2022: 1-10.

[6] Ousterhout A, Fried J, Behrens J, et al. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads[C]//16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19). 2019: 361-378.

[7] Demoulin H M, Fried J, Pedisich I, et al. When idling is ideal: Optimizing tail-latency for heavy-tailed datacenter workloads with perséphone[C]//Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles. 2021: 621-637.