Small is Better: Avoiding Latency Traps in Virtualized Data Centers

Yunjing Xu, Michael Bailey, Brian Noble, Farnam Jahanian
University of Michigan
{yunjing, mibailey, bnoble, farnam}@umich.edu

Abstract

Public clouds have become a popular platform for building Internet-scale applications. Using virtualization, public cloud services grant customers full control of guest operating systems and applications, while service providers still retain the management of their host infrastructure. Because applications built with public clouds are often highly sensitive to response time, infrastructure builders strive to reduce the latency of their data center's internal network. However, most existing solutions require modification to the software stack controlled by guests. We introduce a new host-centric solution for improving latency in virtualized cloud environments. In this approach, we extend a classic scheduling principle—Shortest Remaining Time First—from the virtualization layer, through the host network stack, to the network switches. Experimental and simulation results show that our solution can reduce median latency of small flows by 40%, with improvements in the tail of almost 90%, while reducing throughput of large flows by less than 3%.

Categories and Subject Descriptors: D.4.4 **[Operating Systems]:** Communications Management; D.4.8 **[Operating Systems]:** Performance
Keywords: cloud computing, virtualization, latency

1 Introduction

Large data centers have become the cornerstone of modern, Internet-scale Web applications. They are necessary for the largest of such applications, but they also provide economies of scale for many others via multi-tenancy. Such data centers must support significant aggregate throughput, but also must ensure good response time for these Web applications. This is easier said than done. For example, fulfilling a single page request on Amazon's retail platform typically requires calling over 150 services, which may have inter-dependencies that prevent them from being requested in parallel [15]. For this class of applications, latency is critical—and its improvement lags behind other measures [34]. Worse, low average response time is insufficient—the tail of the latency distribution is a key driver of user perception [3, 4, 52].

We are not the first to recognize this, and progress has been made in both data center networking [3, 44, 4, 52, 41, 23] and operating system support [37, 22, 32]. However, the increasing importance of public cloud services like Amazon's Elastic Compute Cloud (EC2) [6] presents new challenges for latency-sensitive applications. Unlike dedicated data centers, a public cloud relies on *virtualization* to both hide the details of the underlying host infrastructure as well as support *multi-tenancy*.

In virtualized environments, control of the host infrastructure and the guest virtual machines (VMs) are separate and tend to operate at cross purposes. This well-known challenge of virtualization (i.e., the semantic gap [13]) is exacerbated by multi-tenancy in the cloud—cloud providers administer the host infrastructure, while disparate customers control guest VMs and applications. Both virtualization and multi-tenancy create significant challenges for existing latency reduction methods because cloud providers must be host-centric. That is, they should neither trust the guest VMs nor require their cooperation. For example, cloud providers may wish to deploy new congestion control mechanisms tailored for data center traffic [3, 4], but lack the ability to control a guest's choice of TCP algorithms. Likewise, new packet scheduling mechanisms that rely on applications to provide scheduling hints (e.g., priority or deadline) [44, 41, 23, 52, 5] are untenable unless cloud providers can fully trust guests to provide such information honestly and correctly.

SoCC'13, October 01 - 03 2013, Santa Clara, CA, USA.
ACM 978-1-4503-2428-1/13/10$15.00.
http://dx.doi.org/10.1145/2523616.2523620

In this paper, we introduce a novel host-based solution that tackles the need for improved latency in cloud environments. Our solution is surprisingly simple, though it comes with a twist. We apply the Shortest Remaining Time First (SRTF) scheduling policy—known to minimize job queueing time [39]—to three different areas of the system: the VM scheduler, the host network stack, and the data center switches. Using this solution, we can reduce mean latency of small flows by 40%, with improvements in the tail of almost 90%, while reducing throughput of large flows by less than 3%.

Although these gains are significant, we expect that more work is required. Much like the work of scaling up to an ever larger number of shared-memory CPU cores [12], there is no silver bullet that solves the "latency problem." Instead, it is death by a thousand cuts. For example, we argue that the VM scheduler is biased too far towards throughput rather than latency for this workload mix, delaying interrupt processing by tens of milliseconds in some cases. In the host network stack, we uncover obscured short tasks by applying software packet fragmentation earlier in the stack. On data center switches, we prioritize flows of small size, which is inferred in the host, much like CPU schedulers infer short tasks. While finding each of these problems takes time, solving each is relatively straightforward—a few hundred lines of code in total for all three mechanisms. However, once you remove one stumbling block, another appears just down the road [27]. Despite this, we believe that consistently applying latency-aware techniques will continue to bear fruit.

2 Background and Related Work

Data center network latency Based on the design requirements, we classify existing latency reduction solutions as being *kernel-centric*, *application-centric*, or *operator-centric*.

DCTCP [3] and HULL [4] are kernel-centric solutions, because among other things, they both require modifying the operating system (OS) kernel to deploy new TCP congestion control algorithms. On the other hand, applications can enjoy low-latency connections without any modification.

D^3 [44], D^2TCP [41], DeTail [52], PDQ [23], and pFabric [5] are application-centric solutions. While some of them also require modifying the OS kernel or switching fabrics, they share a common requirement. That is, applications must be modified to tag the packets they generate with *scheduling hints*, such as flow deadline or relative priority.

Operator-centric solutions require no changes to either OS kernel or applications, but they do require operators to change their application deployment. For example, our prior work, Bobtail [51], helps operator determine which virtual machines are suited to deploy latency-sensitive workloads without changing the applications or OS kernel themselves.

In a virtualized multi-tenant data center, all of these solutions are in fact *guest-centric*—they require changing the *guest* OS kernel, the *guest* applications themselves, or the way *guest* applications are deployed—none of which are controlled by cloud providers. In contrast, the solution we propose is *host-centric*—it does not require or trust guest cooperation, and it only modifies the host infrastructure controlled by cloud providers.

EyeQ also adopts a host-centric design [26]. While it mainly focuses on bandwidth sharing in the cloud, like our work, EyeQ also discusses the trade-offs between throughput and latency. In comparison, our solution does not require feedback loops between hypervisors to coordinate, and it does not need explicit bandwidth headroom to reduce latency. Additionally, the bandwidth headroom used by EyeQ only solves one of the three latency problems addressed by our solution.

EC2 and the Xen hypervisor Amazon's Elastic Compute Cloud (EC2) [6] is a major public cloud service provider used by many developers to build Internet-scale applications. Measurement studies are devoted to understanding the performance of the EC2 data center network [43] and applications [38, 30, 10]. These studies find that virtualization and multi-tenancy are keys to EC2's performance variation. In particular, workloads running in EC2 may exhibit long tail network latency if co-scheduled with incompatible VMs [51]. In this paper, we consider this co-scheduling problem as one of three major latency sources, and we design a holistic solution to solve all three problems at the *host* level.

EC2 uses the Xen hypervisor [9] to virtualize its hardware infrastructure [43]; thus, we also choose Xen to discuss our problems and solutions. Xen runs on bare-metal hardware to manage guest VMs. A privileged guest VM called dom0 is used to fulfill I/O requests for non-privileged guest VMs; other virtualization platforms may have a host OS to process I/O requests for guest VMs. Because Xen's dom0 and the host OS are functionally equivalent for the discussion in this paper, we use these two terms interchangeably despite their technical difference.

Virtual machine scheduling To improve Xen's I/O performance, new VM scheduling schemes, such as vSlicer [49], vBalance [14], and vTurbo [48], are proposed to improve the latency of interrupt handling by using a smaller time slice for CPU scheduling [49, 48] or by migrating interrupts to a running VM from a preempted one. However, unlike our host-centric design, these approaches either require modifications to

the guest VMs [14, 48] or to trust the guests to specify their workload properties [49], neither of which are easily applicable to public clouds. Similarly, the soft real-time schedulers [31, 29, 47] designed to meet latency targets also require explicit guest cooperation. In addition, by monitoring guest I/O events and giving preferential treatment to the I/O-bound VMs, their I/O performance can be improved without any guest changes [17, 28, 24]. In comparison, our design does not require such gray-box approaches to infer guest VM I/O activities, and thus avoids the complexity and overhead.

Meanwhile, using hardware-based solutions like Intel's Virtual Machine Device Queues (VMDq) [25], guest VMs can bypass the host operating system to handle packets directly and significantly improve their network I/O performance. The deployment of such hardware would eliminate the need to modify the host operating system to reduce the queueing delay in its software queues, which solves one of three latency problems discussed in this work. On the flip side, our modification to reduce the latency in the host network stack is software-only so that it is also applicable to the physical machines without the advanced hardware virtualization support.

Shortest remaining time first SRTF is a classic scheduling policy known for minimizing job queueing time [39] and widely used in system design. For networking systems, Guo *et al.* use two priority classes—small and large flows—to approximate the SRTF policy for Internet traffic; it obtains better response time without hurting long TCP connections [18]. Similarly, Harchol-Balter *et al.* change Web servers to schedule static responses based on their size using SRTF and demonstrate substantial reduction in mean response time with only negligible penalty to responses of large files [20]. In addition, pFabric uses SRTF to achieve near-optimal packet scheduling if applications can provide scheduling hints in their flows [5]. In this paper, we apply SRTF holistically to three areas of virtualized data center infrastructure by following the same principle as these existing systems. Importantly, we approximate SRTF without requiring guest cooperation or explicit resource reservation.

3 Motivation

This section motivates the work by demonstrating three latency traps in a virtualized multi-tenant environment. Using live EC2 measurement and testbed experiments, we show that these problems not only have significant impact on inter-VM network latency separately, but they also affect different parts of the latency distribution orthogonally. Thus, we need a holistic solution to tackle all three problems together.

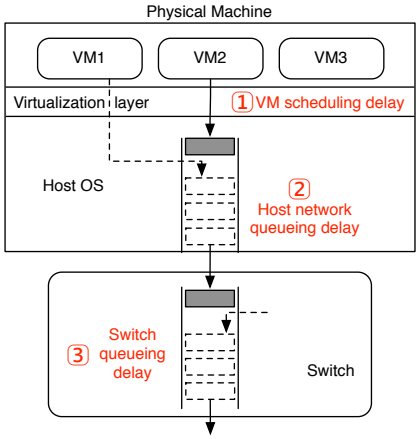


Figure 1: Three possible latency sources.

3.1 Sources of latency

Figure 1 shows three possible latency sources. To understand them, consider a simple scenario where a client VM sends queries to a server VM for its responses. The time from the client to send a query and receive a complete response is defined as flow completion time (FCT).

(1) VM scheduling delay. When query packets arrive at the physical host running the server VM, that VM is notified of the reception. But the server VM cannot process the packets until scheduled by the hypervisor; this gap is called scheduling delay [24]. While such delay is usually less than one millisecond, it may occasionally be as large as tens of milliseconds for Xen's default VM scheduler [46, 16], if there are many VMs sharing the physical host with the server VM. This is a real problem observed in EC2 and it causes large tail latency between EC2 instances [51]. Such delay may also exist for applications running on bare-metal operating systems, but the added virtualization layer substantially exacerbates its impact [43].

(2) Host network queueing delay. After the server VM processes the query and sends a response, the response packets first go through the host network stack, which processes I/O requests on behalf of all guest VMs. The host network stack is another source of excessive latency because it has to fulfill I/O requests for multiple VMs, which may contend to fill up the queues in the host network stack. In fact, reducing queueing delay in the kernel network stack has been a hot topic in the Linux community, and considerable advancements have been made [22, 32]. However, these existing mechanisms are designed without virtualization in mind; the interaction between the host network stack and guest VMs compromises the efficacy of these advancements.

(3) Switch queueing delay. Response packets on the wire may experience switch queueing delay on congested links in the same way as they do in dedicated data centers. Measurement studies show that the RTTs seen

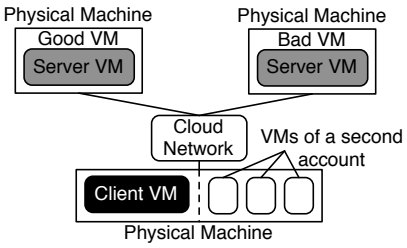


Figure 2: The setup of the EC2 measurements.

on a congested link in dedicated data centers vary by two orders of magnitude [3, 52]; virtualized data centers are no exception. What makes this problem worse in public clouds is that guest VMs on the same host may contend for limited bandwidth available to that host without the knowledge of each other. To continue our example, if the client VM is sharing hardware with a neighbor that receives large bursts of traffic, the response arriving to the client VM may experience queueing delay on its access link, even though the client VM itself is only using a small fraction of the bandwidth assigned to it.

In our example, the queueing delay for the host network stack and switches can occur for both query and response messages, while the VM scheduling delay almost always happens to query messages. In the next few subsections, we demonstrate these problems using live EC2 measurements and testbed experiments.

3.2 EC2 measurements

In this subsection, we demonstrate VM scheduling delay and switch queueing delay using live EC2 experiments. Host network queueing delay is illustrated in the next subsection by varying kernel configurations on a testbed.

Figure 2 shows the setup of our EC2 measurements. We measure the FCT between a client VM and two server VMs of the same EC2 account. In addition, a second EC2 account is used to launch several VMs co-located with the client VM on the same host; these VMs are used to congest the access link shared with the client VM. All VMs used in this experiment are standard medium EC2 instances.

In prior work, we have already demonstrated VM scheduling delay in EC2 [51]; relevant results are reproduced here only for completeness. For the same client VM, we launch several server VMs and measure their FCT distribution separately. The expectation is that the FCT for some server VMs have much larger tail latency (Bad VMs) than for others (Good VMs) if the former are sharing CPU cores with certain VMs running CPU-bound jobs on the same host.

For switch queueing delay, we use the VMs of the second account to show that shared network links can be congested by an arbitrary EC2 customer without violating EC2's terms of use. Because it is hard to control the congestion level in the core EC2 network, our demonstration aims to congest the shared access link of the client VM. To do so, we launch 100 VMs with the second account and keep the ones that are verified to be co-located with the client VM on the same host. The techniques for verifying co-location are well studied [36, 50], so we omit the details.

With enough co-located VMs, we can congest the access link of the client VM by sending bulk traffic to the co-located VMs of the second account from other unrelated VMs. This means that the client VM may suffer from a large switch queueing delay caused by its neighbors on the same host, even though the client VM itself is only using a small fraction of its maximum bandwidth (1Gbps). During the measurement, we observe that the EC2 medium instances have both egress *and* ingress rate limit of 1Gbps, and the underlying physical machines in EC2 appear to have multiple network interfaces shared by their hosted VMs; therefore, we need at least three co-located VMs from the second account to obtain the congestion level needed for the demonstration.

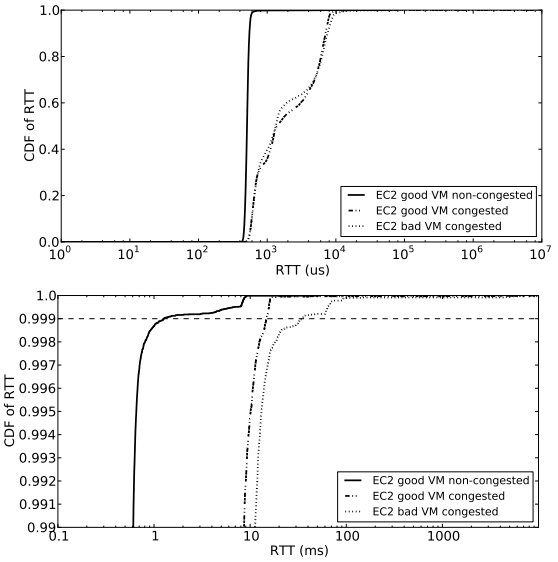


Figure 3: An EC2 experiment showing the VM scheduling delay (bad VM vs. good VM) and switch queueing delay (congested vs. non-congested). Network congestion impacts both the body and tail distribution, while VM scheduling delay mostly impacts the tail at a larger scale.

Scenarios	50th	90th	99th	99.9th
Good VM non-congested	1X	1X	1X	1X
Good VM congested	2.7X	12.9X	14.2	11.7X
Bad VM congested	2.6X	13.7X	18.5	27.7X

Table 1: The relative impact of switch queueing delay (congested vs. non-congested) and VM scheduling delay (bad VM vs. good VM).

Figure 3 depicts the measurement results. There are three scenarios with progressively worsening FCT distributions to show the impact of individual latency sources: one with no delay, one with switch queueing delay only, and one with both switch queueing delay and VM scheduling delay. The top figure shows the entire FCT distribution, and the bottom one shows the distribution from the 99th to 100th percentile. Switch queueing delay has a large impact on both the body and the tail of the distribution, while VM scheduling delay affects mostly the tail of the distribution at a larger scale.

To make things clearer, Table 1 summarizes the relative impact of both latency sources on different parts of the FCT distribution. The key observation is that while switch queueing delay alone can increase the latency by 2X at the 50th percentile and cause a 10X increase at the tail, VM scheduling delay can cause another 2X increase at the 99.9th percentile *independently*. Thus, to avoid latency traps in virtualized data centers, we must tackle these problems together.

3.3 Testbed experiments

In this subsection, we demonstrate the impact of queueing delay in the host network stack on a testbed. The host network stack has at least two queues for packet transmission: one in the kernel network stack that buffers packets from the TCP/IP layer and another in the network interface card (NIC) that takes packets from the first queue and sends them on the wire. NICs may contain many transmission queues, but we only consider the simple case for this demonstration and discuss advanced hardware support in § 6.

A packet may experience queueing delay in the host network stack if either transmission queue is blocked by large bursts of packets from different VMs. This problem is not specific to virtualization-enabled hosts; a bare-metal Linux OS serving multiple applications can exhibit similar behaviors. Thus, we first discuss two features introduced in Linux 3.3 that tackle this problem, and then we explain why the interaction between the host network stack and guest VMs compromises the efficacy of these features.

Herbert *et al.* designed a new device driver interface called Byte Queue Limits (BQL) to manage NICs' transmission queues [22]. Without BQL, the length of NICs' transmission queues was measured by the number of packets, which are variable in size; so, the queueing delay for NICs is often unpredictable. BQL instead measures queue length by the number of bytes in the queue, which is a better predictor of queueing delay. To reduce queueing delay while still maintaining high link utilization with BQL, one can now limit the queue length to the bandwidth-delay product [7] (e.g., for a 1Gbps link with 300us RTT, a queue of 37,500 bytes will suffice).

Scenarios	50th	99th	99.9th
Congestion Free	0.217	0.235	0.250
Congestion Enabled	16.83	21.57	21.73
Congestion Managed	0.808	1.21	1.26

Table 2: Ping latency (ms). Despite the improvement using BQL and CoDel, congestion in the host network stack still increases the latency by four to six times.

Now that NICs' transmission queues are properly managed, the queueing delay problem is pushed to the software queue in the kernel. Nichols *et al.* designed a packet scheduler called "CoDel" that schedules packets based on the amount of time a packet has spent in the queue and then compares it to a dynamically-calculated target of queueing delay [32]. If queued packets have already spent too much time in the queue, the upper layer of the network stack is notified to slow down, regardless of the queue occupancy. Linux's default behavior is to use a large drop tail queue (e.g., 1,000 packets), which incurs a large average queueing delay.

To use BQL and CoDel effectively, it is considered a best practice [45] to disable the packet segmentation offload (TSO and GSO) features offered by the NIC hardware. With TSO and GSO enabled, a packet in either of the transmission queues can be up to 64KB. Compared to Ethernet's MTU of 1,500 bytes, such large packets would again make queue length unpredictable. By disabling TSO and GSO, packets are segmented in software before joining the transmission queues allowing BQL and CoDel to have fine-grained control.

We set up controlled experiments to show the effectiveness of BQL and CoDel by congesting the host network queue and also to demonstrate why they are not sufficient in the presence of virtualization. The testbed has three physical machines running unmodified Xen 4.2.1 and Linux 3.6.6, and they are connected to a single switch. To observe host queueing delay, physical machine A runs two VMs, A1 and A2, that serve as senders. Another two VMs, B1 and C1, run in the remaining two physical machines, B and C, and serve as receivers. In this configuration, we make sure that none of the VMs would suffer from VM scheduling delay or switch queueing delay.

In the experiment, A2 `ping`s C1 10,000 times every 10ms to measure network latency. The traffic between A1 and B1 causes congestion. There are three scenarios:

1. Congestion Free: A1 and B1 are left idle.
2. Congestion Enabled: A1 sends bulk traffic to B1 without BQL or CoDel.
3. Congestion Managed: A1 sends bulk traffic to B1 with BQL and CoDel enabled.

For cases 2 and 3, A1 sends traffic to B1 using `iperf`, which saturates the access link of physical machine A.

Table 2 shows the distribution of the `ping` latency in milliseconds. Without BQL or CoDel, the contention for host network resources alone increases the latency by almost two orders of magnitude. While BQL and CoDel can significantly reduce the latency, the result is still four to six times as large when compared to the baseline. § 4.2 explains why the interaction between the host network stack and guest VMs is the source of such latency.

3.4 Summary

Switch queueing delay increases network tail latency by over 10 times; together with VM scheduling delay, it becomes more than 20 times as bad. In addition, host network queueing delay also worsens the FCT tail by four to six times. Worse still, the impact of these latency sources can be superimposed on one another. Therefore, we need a holistic solution that tackles all of these problems together.

4 Design and Implementation

Applying prior solutions to reduce network latency in virtualized multi-tenant data centers would require guests to change their TCP implementation [3, 4], modify their applications to provide scheduling hints [5, 23, 41, 44, 52], or complicate their workload deployment process [51]. Alternatively, network resources have to be reserved explicitly [26]. Meanwhile, even a combination of these approaches cannot solve all three latency traps discussed in § 3.

Our solution has *none* of the preceding requirements and yet it holistically tackles the three latency traps—we only modify the host infrastructure in a way that is transparent to the guests, and there is no need to explicitly reserve network resources. The details of dealing with individual latency traps are discussed in the subsequent subsections, all of which follow the same principles:

Principle I: Not trusting guest VMs We do not rely on any technique that requires guest cooperation. The multi-tenant nature of public data centers implies that guest VMs competing for shared resources are greedy—they only seek to optimize the efficiency of their own resource usage. The prior solutions tailored for dedicated data centers [3, 4, 5, 23, 41, 44, 52] have proven very effective because the application stacks, operating systems, and hardware infrastructure are controlled by a single entity in such environments. In public data centers, however, trusting guests to cooperatively change the TCP implementation in the guest operating systems or provide scheduling hints would reduce the effectiveness or worse, the fairness of resource scheduling.

Principle II: Shortest remaining time first We leverage Shortest Remaining Time First (SRTF) to schedule bottleneck resources. SRTF is known for minimizing job queueing time [39]. By consistently applying SRTF from the virtualization layer, through the host network stack, to the data center network, we can eliminate the need of explicit resource reservation, but still significantly reduce network latency.

Principle III: No undue harm to throughput SRTF shortens latency at the cost of the throughput of large jobs; we seek to reduce the damage. Due to the fundamental trade-off between latency and throughput in system design, many performance problems are caused by the design choices made to trade one for another. Our solutions essentially revisit such choices in various layers of the cloud host infrastructure and make new trade-offs.

4.1 VM scheduling delay

§ 3.2 shows that VM scheduling can increase tail latency significantly. Such delay exists because Xen fails to allow latency-bound VMs to handle their pending interrupts soon enough. We claim that Xen's current VM scheduler does apply the SRTF principle, but it is applied *inadequately*. That is, there exists a mechanism in the scheduler to allow latency-sensitive VMs to preempt the CPU-bound VMs that mostly use 100% of their allocated CPU time, but it leaves a chance for the VMs that use less CPU time (e.g., 90%) to delay the interrupt handling of their latency-bound neighbors. Thus, our solution is to *apply SRTF in a broader setting* by allowing latency-sensitive VMs with pending interrupts to preempt *any* running VMs.

To understand the preceding argument, one needs to know Xen's VM scheduling algorithm. Credit Scheduler is currently Xen's default VM scheduler [46]. As the name implies, it works by distributing credits to virtual CPUs (VCPUs), which are the basic scheduling units. Each guest VM has at least one VCPU. By default, a VCPU receives up to 30ms CPU time worth of credits based on its relative weight. Credits are redistributed in 30ms intervals and burned when a VCPU is scheduled to use a physical CPU. A VCPU that uses up all its credits enters the `OVER` state, and a VCPU with credits remaining stays in the `UNDER` state. The scheduler always schedules `UNDER`s before any `OVER`s. Importantly, if a VCPU waken up by a pending interrupt has credits remaining, it enters the `BOOST` state and is scheduled before any `UNDER`s.

The `BOOST` state is the mechanism designed to approximate SRTF: A VCPU that only spends brief mo-

ments handling I/O events is considered a small job; hence it is favored by the scheduler to preempt CPU-bound jobs as it stays in the BOOST state. Unfortunately, latency-bound VMs may still suffer from large tail latency despite the BOOST mechanism [51], and this is a known limitation [16].

By analyzing Xen's source code, we find that the credit scheduler is still biased far towards throughput. The BOOST mechanism only prioritizes VMs over others in UNDER or OVER states; BOOSTed VMs cannot preempt each other, and they are round-robin scheduled. Thus, if a VM exhausts its credits quickly, it stays in the OVER state most of time, and BOOSTed VMs can almost always preempt it to process their interrupts immediately. However, if a VM sleeps to accumulate credits for a while and then wakes up on an interrupt, it can monopolize the physical CPU by staying BOOSTed until its credits are exhausted, which implies a delay of 30ms or even longer (if more than one VM decide to do that) to neighboring VMs that are truly latency-sensitive [51].

This problem can be solved by *applying SRTF in a broader setting*. We deploy a more aggressive VM scheduling policy to allow BOOSTed VMs to preempt each other. This change is only made to the hypervisor (one line in the source code!) and thus transparent to guest VMs. As a result, true latency-bound VMs can now handle I/O events more promptly; the cost is CPU throughput because job interruption may become more frequent under the new policy. However, such preemption cannot happen arbitrarily: Xen has a `rate limit` mechanism that maintains overall system throughput by preventing preemption when the running VM has run for less than 1ms in its default setting.

4.2 Host network queueing delay

§ 3.3 shows that guest VMs doing bulk transferring can increase their neighbor's host network queueing delay by four to six times, even when BQL and CoDel are both enabled. The root cause is that network requests from bandwidth-bound guest VMs are often *too large and hard to be preempted* in Linux kernels' and NICs' transmission queues. Thus, our solution is to *break large jobs into smaller ones* to allow CoDel to conduct fine-grained packet scheduling.

To understand the above argument, we need to explain Xen's approach to manage network resources for guest VMs. Xen uses a split driver model for both network and disk I/O. That is, for each device, a guest VM has a virtual device driver called `frontend`, and the dom0 has a corresponding virtual device driver called `backend`; these two communicate by memory copy. A packet sent out by guest VMs first goes to the `frontend`, which copies the packet to the `backend`. The job for the `backend` is to communicate with the real network device driver on dom0 to send out packets on the wire. To do so, Xen leverages the existing routing and bridging infrastructure in the Linux kernel by treating the virtual `backend` as a real network interface on the host so that every packet received by a `backend` will be routed or bridged to the physical NICs. Packet reception simply reverses this routing/bridging and memory copy process.

The overhead of packet copy between guest and host may be prohibitive during bulk transferring. Thus, Xen allows guest VMs to consolidate outgoing packets to up to 64KB regardless of NICs' MTU (e.g., 1,500 bytes for Ethernet). Unfortunately, this optimization also exacerbates host network queueing delay. Recall that the key to the success of BQL and CoDel is the fine-grained control of transmission queue length. By allowing bursts of 64KB packets, there are often *large jobs blocking the head of the line of host transmission queues*. Thus, our solution is to *segment large packets into smaller ones* so that CoDel would allow packets of latency-sensitive flows (small jobs) to preempt large bursts.

The question is *when* to segment large packets. Recall that to use BQL and CoDel effectively, best practice suggests turning off hardware segmentation offloading to segment large packets in software [45]. Unfortunately, it only works for packets generated by dom0 itself; traffic from guest VMs is routed or bridged directly, without segmentation, before reaching the NICs. Thus, a straightforward solution is simply to disallow guest VMs to send large packets and force them to segment in software before packets are copied to `backend`. However, according to Principle I, we cannot rely on guests to do so cooperatively, and it consumes guests' CPU cycles. Alternatively, Xen's `backend` can announce to the guest VMs that hardware segmentation offload is not supported; then guests have to segment the packets before copying them. While this approach achieves the goal without explicit guest cooperation, it disables Xen's memory copy optimization completely.

Thus, the key to a practical solution is to delay the software segmentation as much as possible—from guest kernel to host device driver, earlier software segmentation implies higher CPU overhead. Our solution is to segment large packets *right before they join the software scheduling queue managed by CoDel in the host network stack*. In order to give CoDel fine-grained jobs to schedule, this is the latest point possible.

4.3 Switch queueing delay

§ 3.2 shows that switch queueing delay can increase median latency by 2X and cause a 10X increase at the tail. This problem is not unique to public data centers but also exists in dedicated data centers. However, the added

layer of virtualization exacerbates the problem and renders the solutions that require kernel or application cooperation impractical because these software stacks are now controlled by the guests instead of cloud providers.

Our host-centric solution approximates SRTF by letting switches favor small flows when scheduling packets on egress queues. The key insight here is that we can infer small flows like CPU schedulers infer short tasks—based on their resource usage (e.g., bandwidth). Importantly, we *infer flow size in the host* before any packets are sent out. This design also avoids explicit resource reservation required by alternative host-centric solutions [26]. In order to realize this design, we need to answer a few questions: a) How to define a flow? b) How to define flow size? c) How to classify flows based on their size?

To define what a flow is for the purpose of network resource scheduling, we first need to understand what the bottleneck resource is. Because switch queueing delay on a link is proportional to the occupancy of the switch egress queue, any packet would occupy the queueing resource. Thus, we define a flow as the collection of *any* IP packets from a source VM to a destination VM. We ignore the natural boundary of TCP or UDP connections because, from a switch's perspective, one TCP connection with N packets occupies the same amount of resources as two TCP connections with N/2 packets each for the same source-destination VM pair.

By ignoring the boundary of TCP or UDP connections, the size of a flow in our definition can be arbitrarily large. Therefore, we adopt a message semantic by treating a query or a response as the basic communication unit and define flow size as the *instant size of a message* the flow contains instead of the absolute size of the flow itself. In reality, it is also difficult to define message boundaries. Thus, we measure flow by *rate* as an approximation of the message semantic. That is, if a flow uses a small fraction of the available link capacity in small bursts, it is sending small messages and thus treated as a small flow. In the context of SRTF policy, it states that a flow at a low sending rate behaves just like a short job with respect to the usage of switch queueing resources.

Based on the preceding definitions, we classify flows into two classes, small and large, as is done by systems that apply SRTF to Internet traffic [18] and Web requests [20]. Bottleneck links can now service small flows with a higher priority than the large ones. As in the prior work, lower latency of small flows is achieved at the cost of the throughput of large flows. To ensure scalability and stay transparent to the guests, the classification is done in the host of the source VM of the target flow. Each packet then carries a tag that specifies its flow classification for bottleneck switches to schedule.

One important advantage of this rate-based scheme is that it can avoid starving large TCP connections. Imagine if we use TCP connection boundaries and define flow size as the number of packets each TCP connection contains. A large TCP connection may starve if the capacity of a bottleneck link is mostly utilized by high-priority small flows. This is because a large TCP connection lowers its sending rate for packet drops to avoid congestion, but small flows often send all packets before reaching the congestion avoidance stage. If we classify TCP connections by their absolute size, the large ones are always tagged as low priority and serviced after the small ones, regardless of their instant bandwidth usage. However, if classified by rate, a low priority flow may eventually behave just like a high priority one, by dropping its sending rate below a threshold. It will then get its fair share on the bottleneck link.

To implement this policy, we need two components. First, we build a monitoring and tagging module in the host that sets priority on outgoing packets without guest intervention. Small flows are tagged as high-priority on packet headers to receive preferential treatment on switches. Second, we need switches that support basic priority queueing; but instead of prioritizing by application type (port), they can just leverage the tags on packet headers. It is common for data center grade switches to support up to 10 priority classes [44], while we only need two. Moreover, if more advanced switching fabrics like pFabric[5] become available, it is straightforward to expand our solution to include fine-grained classification for more effective packet scheduling.

In our current implementation, the monitoring and tagging module is built into Xen's network `backend` running as a kernel thread in dom0. This is a natural place to implement such functionality because `backend` copies every packet from guest VMs, which already uses a considerable amount of CPU cycles (e.g., 20%) at peak load; the overhead of monitoring and tagging is therefore negligible. In addition, because the overhead of monitoring and tagging is proportional to packet count instead of packet size, retaining guest VMs' ability to consolidate small packets to up to 64KB further improves this module's performance.

Finally, the flow classification algorithm is implemented using a standard token bucket meter. We assign a token bucket to each new flow with a `rate` and a `burst` parameter. `Rate` determines how fast tokens (in bytes) are added to the bucket, and `burst` determines how many tokens can be kept in the bucket unused. The token bucket meters a new packet by checking if there are enough tokens to match its size, and it consumes tokens accordingly. If there are enough tokens, the current sending rate of the flow is considered conformant, and the packet is tagged with high priority. Otherwise, it is

classified as bandwidth-bound and serviced on a best effort basis. Low priority flows are re-checked periodically in case their sending rates drop. Similar to [18], flows are garbage-collected if they stay inactive for long enough. A flow may accumulate tokens to send a large burst, so `burst` limits the maximum tokens that a flow can consume in one shot.

4.4 Putting it all together

Our design revisits the trade-offs between throughput and latency. For VM scheduling delay, we apply a more aggressive VM preemption policy to Xen's VM scheduler at the cost of the efficiency of CPU-bound tasks. For host network queueing delay, we segment large packets from guests earlier in the stack for fine-grained scheduling with higher host CPU usage. For switch queueing delay, we give preferential treatment to the small flows at a low sending rate on switches with the loss of throughput for large flows. The performance trade-offs of this design are evaluated in § 5, and the possibility of gaming these new policies is discussed in § 6.

In our current implementation, we change a single line in the credit scheduler of Xen 4.2.1 to enable the new scheduling policy. We also modify the CoDel kernel module in Linux 3.6.6 with about 20 lines to segment large packets in the host. Finally, we augment the Xen's network `backend` with about 200 lines of changes to do flow monitoring and tagging. Our code patch can be found using the link [1].

5 Evaluation

We use both a small testbed and an ns-3-based [33] simulation to do our evaluation. The testbed consists of five four-core physical machines running Linux 3.6.6 and Xen 4.2.1. They are connected to a Cisco Catalyst 2970 switch, which supports priority queueing (QoS). All NICs are 1Gbps with one `tx` ring. We start with a setup that includes all three latency traps to provide a big picture of the improvement our holistic solution brings before evaluating them individually. In addition to evaluating all three problems on the testbed, we use ns-3 to demonstrate the trade-off at scale for switch queueing delay. The other two problems are local to individual hosts, so no large-scale simulation is needed.

The workload for testbed evaluation models the query-response pattern. As in § 3, client VMs measure the round-trip times of each query-response pair as flow completion times (FCTs). Because the size of a response may range from a few kilobytes to tens of megabytes, FCT serves as a metric for both latency-sensitive traffic and bandwidth-bound traffic. This setup is similar to prior studies [4, 52] for dedicated data centers. In addition, `iperf` is used when we only need to saturate a bottleneck link without measuring FCT.

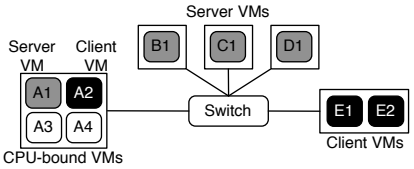


Figure 4: The testbed experiment setup.

The parameters for flow classification include `rate` and `burst` for the token buckets meters, and the timers for cleaning inactive flows and re-checking low priority flows. We set the timers to be 10s and 100ms, respectively, and use 30KB for `burst` and 1% of the link capacity or 10Mbps for flow `rate`. Optimal values would depend on the traffic characteristics of the target environment, so we use the preceding values as our best guest for testbed evaluation and explore the parameter sensitivity using simulation. Meanwhile, measurement studies for data center traffic may help narrow down parameter ranges. For example, Benson *et al.* show that 80% of flows are under 10KB in various types of data centers [11]. Others have reported that flows under 1MB may be time sensitive, and such flows often follow a query-response pattern [3, 44]. Thus, setting `burst` anywhere between 10KB and 1MB would be a reasonable choice.

5.1 Impact of the approach

We first evaluate the overall benefits of our host-centric solution on the testbed by constructing a scenario with all three latency traps enabled. As shown in § 3, this scenario includes the *typical* network resource contention that can happen in virtualized multi-tenant data centers. We compare three cases: the ideal case without any contention, the fully contended case running unpatched Linux and Xen, and the patched case with our new policies. In the unpatched case, BQL and CoDel are enabled to represent the state-of-the-art prior to our solution.

Figure 4 shows the experiment setup. To create contention, we have physical machine E running two VMs, E1 and E2, that serve as clients to send small queries by following a Poisson process and measure FCT. Physical machine A runs four VMs—A1 through A4—with A1 serves small responses to E1, A2 sends bulk traffic using `iperf` to B1 on machine B, and the others run CPU-bound tasks. The query-response flows between E1 and A1 will suffer *VM scheduling delay* caused by A3 and A4, and *host network queueing delay* caused by A2. Moreover, we use physical machine C and D, running VM C1 and D1 respectively, to respond to E2's queries

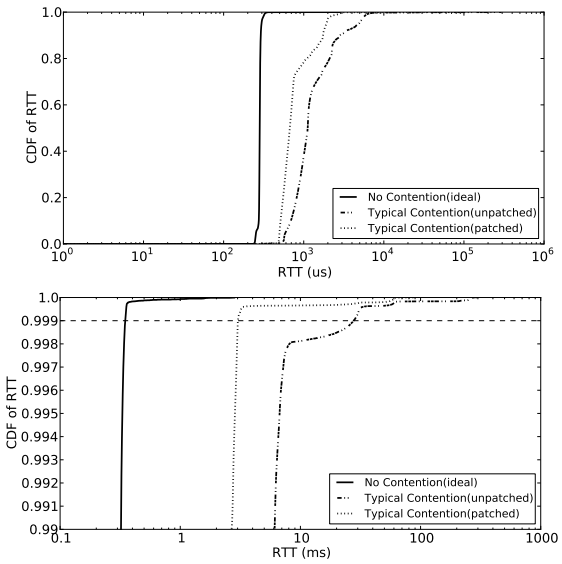


Figure 5: A comparison of FCT distribution for three cases: an ideal case without contention, the default unpatched case under typical contention, and the case patched with our solution under the same contention.

for large flows and congest E's access link, and the responses sent to E1 will suffer *switch queueing delay*. The small query flows generated by E1 expect 2KB responses with 5ms mean inter-arrival time. Queries generated by E2 expect 10MB large responses with 210ms mean inter-arrival time, so E's access link is 40% utilized on average, which is more likely than full saturation [4].

Figure 5 depicts the results. Our solution achieves about 40% reduction in mean latency, over 56% for the 99th percentile, and almost 90% for the 99.9th percentile. While the alleviation of host and switch queueing delay has a large impact on the 99th and lower percentiles, the improvement for the 99.9th percentile is mostly attributed to the change in the VM scheduler. However, compared to the baseline with no contention, there is still room for improvement, which we discuss when evaluating individual components. On the other hand, the average throughput loss for large flows is less than 3%. The impact on CPU-bound tasks and host CPU usage is covered in detail in subsequent subsections.

5.2 VM scheduling delay

To evaluate the new VM scheduling policy, we use E1 to query A1 for small responses and keep A3 running a CPU-bound workload to delay A1's interrupt handling. In order to cause scheduling delay, A1 and A3 need to share a CPU core [51]. Thus, we pin both VMs onto one CPU core and allocate 50% CPU cycles to each VM. Note that the delay caused by VM scheduling happens when a query flow from E1 arrives at A1 and before A1 is allowed to process its (virtual) network interrupt.

Scenarios	99.9th Latency	Avg. CPU Throughput
Xen Default	29.64ms	224.29 ops/s
New Policy	1.35ms	215.80 ops/s

Table 3: The trade-off between network tail latency and CPU throughput measured by memory scans per second.

Now the question is what CPU-bound workload we use to demonstrate the trade-off between network tail latency and CPU throughput. As explained in § 4.1, a workload that always uses 100% CPU time cannot do the job. Instead, we need a workload that follows the pattern of yielding CPU briefly between CPU-hogging operations in order to accumulate credits; then it can get `BOOST`ed and delay its neighbors' interrupt handling [51]. For example, a workload can sleep 1ms for every 100 CPU-bound operations. Note that this pattern is not uncommon for real-world workloads, which often wait for I/O between CPU-bound operations.

The next question is what operations we use to hog the CPU and exhibit slowdown when preempted. We cannot use simple instructions like adding integers because they are not very sensitive to VM preemption: If such operations are interrupted by VM preemption and then resumed, the penalty to their running time is negligible. We instead let the workload burn CPU cycles by scanning a CPU-L2-cache-sized memory block repeatedly. Such operations are sensitive to CPU cache usage because if certain memory content is cached, the read operations finish much faster than the ones fetching from memory directly, and the CPU cache usage is in turn sensitive to VM preemption and context switching [36, 42].

In this experiment, trade-offs between tail latency and CPU throughput are demonstrated for Xen's default scheduling policy and our new policy. To do so, we fix both the number of network requests sent to A1 and the number of memory scan operations conducted on A3 so that each workload contains the same amount of work in both scenarios. The number of memory scan operations per second measured on A3 is used to quantify the throughput for the CPU-bound job.

Table 3 summarizes the results. Our new policy reduces network latency at the 99.9th percentile by 95% at the cost of 3.8% reduction in CPU throughput. In addition to the synthetic workload, we also tested with two SPEC CPU2006 [21] benchmarks, `bzip2` and `mcf`. The difference of their running time under different scheduling policies are less than 0.1%.

The new policy hurts CPU throughput because, while the workloads generate the same number of I/O interrupts in both scenarios, the CPU-bound job is more likely to be preempted in the middle of a scan operation under the new policy, which incurs the overhead of cache

Scenarios	50th	99th	99.9th
Congestion Free	0.217	0.235	0.250
Congestion Enabled	16.83	21.57	21.73
Congestion Managed	0.808	1.21	1.26
Our System	0.423	0.609	0.650

Table 4: The distribution of RTTs in millisecond. Our solution delivers 50% reduction in host network queueing delay in addition to that achieved by BQL and CoDel.

eviction and context switching. In comparison, Xen's default policy is more likely to allow I/O interrupts to be handled when the CPU-bound job is in the sleep phase.

To understand why our new policy only introduces low overhead, we record VCPU state-changing events in the VM scheduler since the loss in throughput is mostly caused by the extra VM preemption. We find that our new policy does increase the number of VM preemption, but the difference is only 2%. This is because the original `BOOST` policy already allows frequent preemption, and our change only affects the cases that correspond to the tail latency (99.9th or higher).

Further improvement is possible. Xen has the default policy that prevents VM preemption when the running VM has run for less than 1ms. If we set it to be the minimum 0.1ms, the 99.9th percentile latency can be reduced even further. However, such change has been shown to cause significant performance overhead to certain CPU-bound benchmarks [46] and may compromise fairness.

5.3 Host network queueing delay

For host network queueing delay, our setup is similar to the testbed experiment in § 4.2. Specifically, we use A1 to `ping` E1 once every 10ms for round-trip time (RTT) measurements and use A2 to saturate B1's access link with `iperf`, and we measure bandwidth. Because E1 and B1 use different access links, there is no switch queueing delay in this experiment. Throughout the experiment, hardware segmentation offload is turned off in order to use BQL and CoDel effectively [45].

Table 4 lists the results for the impact on `ping` RTTs. Compared to the case that applies BQL and CoDel unmodified (Congestion Managed), our solution can yield an additional 50% improvement at both the body and tail of the distribution because CoDel is more effective in scheduling finer-grained packets (small jobs). Meanwhile, the bandwidth loss for breaking down large packets early is negligible.

However, we do trade CPU usage for lower latency. When B1's 1Gbps access link is saturated, our current implementation increases the usage of one CPU core by up to 12% in the host due to earlier software segmentation, and that amount is negligible for guest VMs. Compared to the alternative that forces guest VMs to segment large packets without consolidation, hosts' CPU usage would increase by up to 30% from the overhead of copying small packets from guests, and that for the guest VMs would increase from 13% to 27% for its own software segmentation. Thus, our solution is not only transparent to users, but has less CPU overhead for both host and guest. This is because our choice of software segmentation is early enough to achieve low latency but late enough to avoid excessive CPU overhead.

Again there is still room for improvement. We speculate that the source of the remaining delay is from NICs' transmission queues. Recall that we set that queue to hold up to 37,500 bytes of packets, which translates to a 300μs delay on a 1Gbps link at maximum. To completely eliminate this delay, if necessary, we need higher-end NICs that also support priority queueing. Then, the NICs' drivers can take advantage of the flow tagging we assigned to the latency-sensitive flows and send them before any packets of large flows.

5.4 Switch queueing delay

To evaluate the flow monitoring and tagging module, we leave VMs A2 through A4 idle. For the rest of the VMs, E1 queries A1 and B1 in parallel for small flows, and E2 queries C1 an D1 for large flows to congest the access link shared with E1 and cause switch queueing delay.

Similar to the "Dynamic Flow Experiments" used by Alizadeh *et al.* [4], we set the utilization of the access links to be 20%, 40%, and 60% in three scenarios, respectively, instead of fully saturating them. To do so, VM E2 runs a client that requests large response flows from C1 and D1 in parallel; it varies query rate to control link utilization. For example, this experiment uses 10MB flows as large responses, so we approximate 40% link utilization by using a Poisson process in E2 to send 5 queries per second on average. The case with the switch QoS support enabled is compared against the one without QoS support; QoS enabled switches can prioritize packets tagged as belonging to small flows.

Table 5 summarizes the results. As expected, when QoS support on the switch is enabled to recognize our tags, all small flows enjoy a low latency with an order of magnitude improvement at both the 99th and 99.9th percentiles. As the link utilization increases, their FCT increases only modestly. On the other hand, the *average* throughput loss for large flows is less than 3% In fact, the average throughput is not expected to experience a significant loss under the SRTF policy if the flow size follows a heavy-tail distribution [20]. This is because under such distribution, bottleneck links are not monopolized by high priority small flows and they have spare capacity most of time to service large flows with modest delay.

		2KB FCT (ms)					10MB FCT (ms)				
		avg.	50th	90th	99th	99.9th	avg.	50th	90th	99th	99.9th
20% Load	QoS Disabled	0.447	0.300	0.450	3.630	5.278	107.402	91.144	158.530	252.537	332.611
	QoS Enabled	0.298	0.298	0.319	0.357	0.499	109.809	91.308	159.768	314.574	431.564
40% Load	QoS Disabled	0.779	0.304	1.845	5.155	5.437	134.294	99.497	217.031	392.592	539.759
	QoS Enabled	0.305	0.301	0.333	0.441	0.533	138.002	104.164	222.907	419.591	639.909
60% Load	QoS Disabled	1.428	0.408	4.618	5.293	5.497	182.512	149.366	325.482	644.225	907.877
	QoS Enabled	0.319	0.305	0.363	0.514	0.551	187.316	153.745	338.976	646.169	908.825

Table 5: The results for tackling switch queueing delay with flow tagging and QoS support on the switch. Both the latency-sensitive flows and bandwidth-bound flows are measured by their FCT.

Benson *et al.* show that the flow size in various cloud data centers does fit heavy-tailed distributions [11].

Meanwhile, the increase in FCT for 10MB flows at the 99th and 99.9th percentiles can be as large as 30%, which is comparable to HULL [4]. This is expected because for 1% or 0.1% of these low-priority flows, there could be high-priority packets that happen to come as a stream longer than average. Because a switch egress queue has to service the high-priority packets first, before any best-effort packets, a few unlucky large flows have to be stuck in the queue substantially longer than average. In fact, the larger the flows are, the more likely they are affected by a continuous stream of high priority packets. On the other hand, larger flows are also less sensitive to queueing delay, and tail FCT only has marginal impact on their average throughput.

For testbed experiments, because the size of both small and large flows is fixed, all flows are correctly tagged. To demonstrate the sensitivity of the parameters, we need to test the solution against dynamically generated flows that follow a heavy-tail distribution, which is discussed in the next subsection.

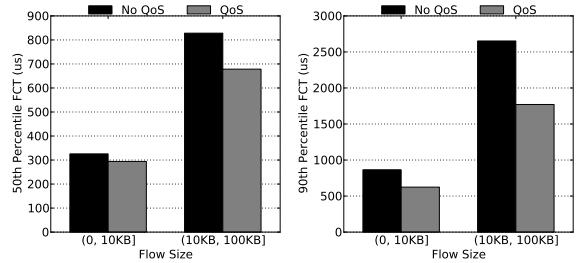
5.5 Large-scale simulation

To explore the scalability and parameter sensitivity of our solution for switch queueing delay, we use ns-3 to simulate a multi-switch data center network. Because only switch queueing delay is evaluated here, we install multiple applications on each simulated host to mimic multiple VMs. All applications have their own sockets that may transmit or receive traffic simultaneously to cause congestion. The flow monitoring and tagging module also uses a 10Mbps `rate` 30KB `burst` token bucket meter per flow to begin with, and we vary them in the parameter sensitivity analysis.

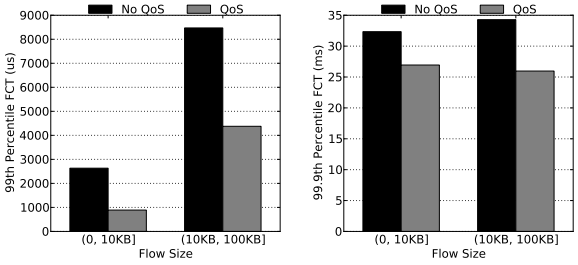
Our simulated network follows the fat-tree topology [2]. There are 128 simulated hosts divided into four pods. In each pod, four 8-port edge switches are connected to the hosts and to four aggregation switches. In the core layer, 8 switches connect the four pods together. The links between hosts and edge switches are 1Gbps, and those between switches are 10Gbps, so that there is no over-subscription in the core network. We set each edge switch to incur a 25μs delay and for upper-level switches, a 14.2μs delay. Thus, the intra-switch RTT is 220μs, the intra-pod RTT is 276.7μs, and the inter-pod RTT is 305.2μs. The routing table of the network is generated statically during simulation setup, and we use flow hashing to do load balancing.

The workload we use in the simulation is the same as in a prior study [4]. Specifically, each host opens a permanent TCP connection to all other hosts and chooses destinations at random to send flows to. Therefore, the FCT here is the time to finish a one-way flow instead of a query-response pair. The flows are generated by following a Poisson process with 2ms inter-arrival time on average. In addition, the flow size is generated using a Pareto random variable with the same parameters as in [4]: 1.05 for the shape parameter and 100KB mean flow size. As result, the utilization of access links is approximately 40%. This flow size distribution is picked to match the heavy-tailed workload found in real networks: most flows are small (e.g., less than 100KB), while most bytes are from large flows (e.g., 10MB and larger).

Figure 6 shows the FCT for small flows of range (0, 10KB] or (10KB, 100KB]; the flow size categorization is also based on [4]. We can observe improvement for both types of small flows at all percentiles, and the improvement ranges from 10% to 66%. However, unlike the testbed results, the FCTs with QoS enabled in the simulation result are not consistently low. While the 50th, 90th, and 99th percentile values are all comparable to the testbed results, the 99.9th percentile is an order of magnitude higher, and we only achieve 17% and 24% improvement for both types of small flows, respectively. This is an expected result because the flow sizes vary in the simulation and the flow tagging module may not be able to tag every flow correctly. More importantly, our tagging algorithm has an implicit assumption that a small flow stays small most of time, and a large flow also stays large most of time, which is not unreasonable for real world workload patterns. However, the simulated workload maintains permanent TCP connections for ev-



(a) 50th Percentile **(b)** 90th Percentile



(c) 99th Percentile **(d)** 99.9th Percentile

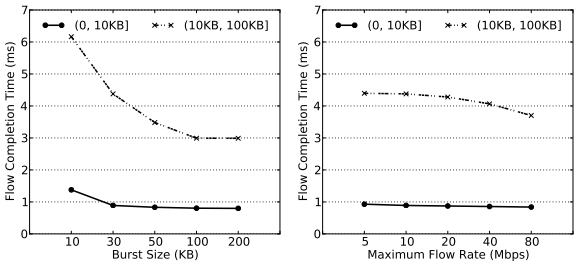
Figure 6: The 50th, 90th, 99th, 99.9th percentile FCT for small flows of range (0, 10KB] and (10KB, 100KB].

Flow Size	No QoS (ms)	QoS (ms)
(100KB, 10MB]	11.878	10.608
(10MB, ∞)	669.618	658.404

Table 6: The average FCTs for large flows.

ery pair of applications (VMs), and the flow size for each connection alone follows the Pareto distribution. As a result, a connection may transmit a random mix of flows with different sizes so that there is no correct label to be assigned to any connection. The flow tagging module is therefore more prone to making mistakes. As discussed in § 6, evaluating with real world flow traces would give us a better understanding of the problem.

Table 6 compares the average FCTs for large flows. To some degree, the FCTs are similar regardless of whether QoS is enabled or not, which is expected because when all the access links are 40% utilized and the core network is not over-subscribed, large flows are not expected to be starved by small flows. On the other hand, however, the numbers with QoS enabled are slightly smaller than that without QoS. This is the *opposite* of the testbed results. There are at least two possible explanations. First, our implementation of priority queueing and buffer management on simulated switches is very primitive compared to that in real Cisco switches. Thus, certain performance-related factors associated with priority queueing may be missing in the simulation. Secondly, with QoS enabled, small and large flows are classified into two separate queues on simulated switches, thus bandwidth-bound flows are protected from packet drops caused by bursts of small flows and hence operate in a more stable environment. A similar speculation is made in the study that applies SRTF to Internet traffic [18]. Harchol-Balter *et al.* also find that for Web requests of static files using a two-class SRTF scheduling can benefit *all* requests [20].



(a) 99th Percentile **(b)** 99th Percentile

Figure 7: The 99th percentile FCT for small flows with varying burst size and flow rates, respectively.

Finally, we evaluate the sensitivity of `burst` and `rate` for flow monitoring and tagging. The choices for these parameters depend on the flow characteristics of the target data center environment. According to recent data center measurements [11, 3, 44], any value in the range of [10KB, 1MB] may be reasonable for `burst` as the size of a single message. Once `burst` is determined, `rate` determines how many query-response pairs can be exchanged per second before a flow is classified as low priority. To demonstrate the sensitivity, we first fix `burst` to be 10Mbps and vary `burst` to be {10, 30, 50, 100, 200}KB. Then, we fix `burst` to be 30KB and use {5, 10, 20, 40, 80}Mbps as `rate` values.

Figure 7 shows the 99th percentile FCT for small flows with varying burst sizes or flow rates. Overall, the FCT improves as burst size increases until 100KB. It is the mean size of all flows, and over 90% of the flows are smaller than that. Given the workload distribution, this is a good threshold to use to distinguish between small and large flows. Meanwhile, increasing maximum flow rate improves the FCT modestly because more flows are tagged as high priority under a larger maximum flow rate. However, in practice, a large maximum flow rate may be abused by cloud guests to block the head-of-line of switch priority queues. A similar trend is observed for the 99.9th percentile, but to a lesser extent. Meanwhile, the impact of these parameters on the FCT of small flows at lower percentiles is even smaller, and the average FCT of large flows does not exhibit any clear pattern. Thus, both results are omitted here.

6 Limitations

Gaming the SRTF policies Our new policy for reducing host network queueing delay (§ 4.2) does not open

new windows for gaming since it only requires software segmentation to be conducted earlier in the stack. However, it is possible to game our new policies for the VM scheduler (§ 4.1) and data center switches (§ 4.3), although doing so would either be very difficult or only have limited impact.

There are two potential ways to exploit our new VM scheduling policy (§ 4.1). First, a greedy VM may want to monopolize the shared physical CPUs by accumulating credits as it does with Xen's default policy. However, our new policy is at least as fair as the default policy, and it makes such exploitation more difficult because the new policy allows `BOOST`ed VMs to preempt each other: Every VM has the same chance of getting the physical CPU, but other `BOOST`ed VMs can easily take the CPU back. The second issue is that malicious VMs may force frequent VM preemption to hurt overall system performance. Fortunately, by default, Xen's `rate limit` ensures that preemption cannot happen more often than once per millisecond. Moreover, § 5.2 shows the extra preemption overhead introduced by our new policy is rather limited because Xen's `BOOST` mechanism already allows fairly frequent VM preemption and our change only affects the tail cases.

For the two-class priority queueing policy (§ 4.3), greedy guests may break a large flow into smaller ones to gain a higher priority. However, because we *define a flow as a collection of any packets from a source VM to a destination VM*, creating N high priority flows would require N different source-destination VM pairs. Using parallel TCP connections to increase resource sharing on bottleneck links is a known problem [19] regardless of applying SRTF or not. Our new policy would only be exploited when multiple pairs of VMs are colluding together, which is not necessarily unfair because the customers have to pay for more guest VMs in this case. In fact, the fairness consideration in this scenario is a research problem by itself [40, 35, 8], which is out of the scope of this paper.

10Gbps and 40Gbps networks While we impose no requirement on the core cloud network, our solution assumes 1Gbps access links. We segment large packets earlier in software to reduce host queueing delay. However, using software segmentation is not a limitation introduced by our solution; it is suggested to turn off hardware segmentation to use BQL and CoDel effectively [45]. Host queueing delay may become less of an issue with 10Gbps and 40Gbps access links because transmitting a 64KB packet only takes about 52.4μs at 10Gbps and 13.1μs at 40Gbps, in which case software segmentation may no longer be necessary.

Hardware-based solutions Hardware-based solutions may be needed to cope with higher bandwidth. As discussed, Intel's VMDq [25] may reduce host network queueing delay without any modification to the host kernel. In addition, monitoring and tagging of flows and transmission queue scheduling can also be implemented in the NIC hardware. Because these algorithms rely on standard constructs (e.g., token bucket), and some of them are specifically designed to be hardware friendly [32], the added complexity may not be prohibitive.

Real-world workloads in public clouds To systematically determine the optimal settings for flow tagging, a detailed flow level measurement in the target public cloud is needed. It would be ideal to collect a set of flow level traces from real-world public clouds and make it available to the public so that research results may become more comparable and easier to reproduce. In lieu of such authentic workloads or traces, our synthetic workloads must suffice.

7 Conclusion

In this paper, we explore network latency problems in virtualized multi-tenant clouds. Using EC2 measurement and testbed experiments, we explain why existing solutions designed for dedicated data centers are rendered impractical by virtualization and multi-tenancy. To address these challenges, we design a host-centric solution that extends the classic shortest remaining time first scheduling policy from the virtualization layer, through the host network stack, to the network switches without requiring or trusting guest cooperation. With testbed evaluation and simulation, we show that our solution can reduce median latency of small flows by 40%, with improvements in the tail of almost 90%, while reducing throughput of large flows by less than 3%.

8 Acknowledgments

This work was supported in part by the Department of Homeland Security Science and Technology Directorate under contract numbers D08PC75388, FA8750-12-2-0314, and FA8750-12-2-0235; the National Science Foundation (NSF) under contract numbers CNS 1111699, CNS 091639, CNS 08311174, CNS 0751116, CNS 1330142, and CNS 1255153; and the Department of the Navy under contract N000.14-09-1-1042. This material was based on work supported by the National Science Foundation, while working at the Foundation. Any opinion, finding, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

References

[1] Code patch for this paper. `http://goo.gl/yYjU9`.

[2] M. Al-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity Data Center Network Architecture. In *Proceedings of the ACM SIGCOMM 2008 conference*, Seattle, WA, USA, August 2008.

[3] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data Center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM 2010 conference*, New Delhi, India, August 2010.

[4] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda. Less is More: Trading a little Bandwidth for Ultra-Low Latency in the Data Center. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation (NSDI'12)*, San Jose, CA, USA, April 2012.

[5] M. Alizadeh, S. Yang, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. Deconstructing Datacenter Packet Transport. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks (HotNets'12)*, Redmond, WA, USA, October 2012.

[6] Amazon Web Services LLC. Amazon Elastic Compute Cloud (EC2). `http://aws.amazon.com/ec2/`.

[7] G. Appenzeller, I. Keslassy, and N. McKeown. Sizing Router Buffers. In *Proceedings of the ACM SIGCOMM 2004 conference*, Portland, OR, USA, August 2004.

[8] H. Ballani, K. Jang, T. Karagiannis, C. Kim, D. Gunawardena, and G. O'Shea. Chatty Tenants and the Cloud Network Sharing Problem. In *Proceedings of the 10th USENIX Symposium on Networked System Design and Implementation (NSDI'13)*, Lombard, IL, April 2013.

[9] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*, Bolton Landing, NY, USA, October 2003.

[10] S. K. Barker and P. Shenoy. Empirical Evaluation of Latency-sensitive Application Performance in the Cloud. In *Proceedings of the 1st annual ACM SIGMM conference on Multimedia systems (MMSys'10)*, Scottsdale, AZ, USA, February 2010.

[11] T. Benson, A. Akella, and D. A. Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *Proceedings of the 2010 Internet Measurement Conference (IMC'10)*, Melbourne, Australia, November 2010.

[12] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An Analysis of Linux Scalability to Many Cores. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI'10)*, Vancouver, BC, Canada, October 2010.

[13] P. Chen and B. Noble. When Virtual Is Better Than Real. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS'01)*, Washington, DC, USA, May 2001.

[14] L. Cheng and C.-L. Wang. vBalance: Using Interrupt Load Balance to Improve I/O Performance for SMP Virtual Machines. In *Proceedings of ACM Symposium on Cloud Computing 2012 (SoCC'12)*, San Jose, CA, USA, October 2012.

[15] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-Value Store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP'07)*, Stevenson, WA, USA, October 2007.

[16] G. W. Dunlap. Scheduler Development Update. In *Xen Summit Asia 2009*, Shanghai, China, November 2009.

[17] S. Govindan, A. R. Nath, A. Das, B. Urgaonkar, and A. Sivasubramaniam. Xen and Co.: Communication-Aware CPU Scheduling for Consolidated Xen-based Hosting Platforms. In *Proceedings of the 3rd international conference on Virtual execution environments (VEE'07)*, San Diego, CA, 2007, June 2007.

[18] L. Guo and I. Matta. The War Between Mice and Elephants. In *Proceedings of the Ninth International Conference on Network Protocols (ICNP'01)*, Riverside, CA, USA, November 2001.

[19] T. J. Hacker, B. D. Noble, and B. D. Athey. Improving Throughput and Maintaining Fairness Using Parallel TCP. In *Proceedings of the 23rd conference on Information communications (INFOCOM'04)*, Hong Kong, China, March 2004.

[20] M. Harchol-Balter, B. Schroeder, N. Bansal, and M. Agrawal. Size-Based Scheduling to Improve Web Performance. *ACM Transactions on Computer Systems*, 21(2):207–233, May 2003.

[21] J. L. Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34, September 2006.

[22] T. Herbert. bql: Byte Queue Limits. `http://lwn.net/Articles/454378/`.

[23] C.-Y. Hong, M. Caesar, and P. B. Godfrey. Finishing Flows Quickly with Preemptive Scheduling. In *Proceedings of the ACM SIGCOMM 2012 conference*, Helsinki, Finland, August 2012.

[24] Y. Hu, X. Long, J. Zhang, J. He, and L. Xia. I/O Scheduling Model of Virtual Machine Based on Multi-core Dynamic Partitioning. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC'10)*, hicago, IL, USA, June 2010.

[25] Intel LAN Access Division. Intel VMDq Technology. Technical report, Intel, March 2008.

[26] V. Jeyakumar, M. Alizadeh, D. Mazières, B. Prabhakar, C. Kim, and A. Greenberg. EyeQ: Practical Network Performance Isolation at the Edge. In *Proceedings of the 10th USENIX Symposium on Networked System Design and Implementation (NSDI'13)*, Lombard, IL, April 2013.

[27] R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, and A. Vahdat. Chronos: Predictable Low Latency for Data Center Applications. In *Proceedings of ACM Symposium on Cloud Computing 2012 (SoCC'12)*, San Jose, CA, USA, October 2012.

[28] H. Kim, H. Lim, J. Jeong, H. Jo, and J. Lee. Task-aware Virtual Machine Scheduling for I/O Performance. In *Proceedings of the 5th international conference on virtual execution environments (VEE'09)*, Washington, DC, USA, March 2009.

[29] M. Lee, A. S. Krishnakumar, P. Krishnan, N. Singh, and S. Yajnik. Supporting Soft Real-Time Tasks in the Xen Hypervisor. In *Proceedings of the 6th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'10)*, Pittsburgh, PA, USA, March 2010.

[30] A. Li, X. Yang, S. Kandula, and M. Zhang. CloudCmp: Comparing Public Cloud Providers. In *Proceedings of the 2010 Internet Measurement Conference (IMC'10)*, Melbourne, Australia, November 2010.

[31] B. Lin and P. A. Dinda. VSched: Mixing Batch And Interactive Virtual Machines Using Periodic Real-time Scheduling. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing (SC'05)*, Seattle, WA, November 2005.

[32] K. Nichols and V. Jacobson. Controlling Queue Delay. *Queue*, 10(5):20:20–20:34, May 2012.

[33] NS-3. `http://www.nsnam.org/`.

[34] D. A. Patterson. Latency Lags Bandwidth. *Communication of ACM*, 47(10):71–75, Oct 2004.

[35] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica. FairCloud: Sharing the Network in Cloud Computing. In *Proceedings of the ACM SIGCOMM 2012 conference*, Helsinki, Finland, August 2012.

[36] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, You, Get Off of My Cloud! Exploring Information Leakage in Third-Party Compute Clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS'09)*, Chicago, IL, Nov. 2009.

[37] S. M. Rumble, D. Ongaro, R. Stutsman, M. Rosenblum, and J. K. Ousterhout. It's Time for Low Latency. In *Proceedings of the 13th Workshop on Hot Topics in Operating Systems (HotOS XIII)*, Napa, CA, USA, May 2011.

[38] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz. Runtime Measurements in the Cloud: Observing, Analyzing, and Reducing Variance. In *Proceedings of the 36th International Conference on Very Large Data Bases (VLDB'10)*, Singapore, September 2010.

[39] L. E. Schrage and L. W. Miller. The Queue M/G/1 with the Shortest Remaining Processing Time Discipline. *Operation Research*, 14(4):670–684, July–August 1966.

[40] A. Shieh, S. Kandula, A. Greenberg, C. Kim, and B. Saha. Sharing the Data Center Network. In *Proceedings of the 8th USENIX conference on Networked Systems Design and Implementation (NSDI'11)*, Boston, MA, USA, March 2011.

[41] B. Vamanan, J. Hasan, and T. N. Vijaykumar. Deadline-Aware Datacenter TCP (D^2TCP). In *Proceedings of the ACM SIGCOMM 2012 conference*, Helsinki, Finland, August 2012.

[42] V. Varadarajan, T. Kooburat, B. Farley, T. Ristenpart, and M. M. Swift. Resource-Freeing Attacks: Improve Your Cloud Performance (at Your Neighbor's Expense). In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS'12)*, Raleigh, NC, USA, October 2012.

[43] G. Wang and T. S. E. Ng. The Impact of Virtualization on Network Performance of Amazon EC2 Data Center. In *Proceedings of the 29th conference on Information communications (INFOCOM'10)*, San Diego, CA, USA, March 2010.

[44] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron. Better Never than Late: Meeting Deadlines in Datacenter Networks. In *Proceedings of the ACM SIGCOMM 2011 conference*, Toronto, ON, CA, August 2011.

[45] www.bufferbloat.net. Best Practices for Benchmarking CoDel and FQ CoDel. `http://goo.gl/2RhwY`.

[46] xen.org. Xen Credit Scheduler. `http://wiki.xen.org/wiki/Credit_Scheduler`.

[47] S. Xi, J. Wilson, C. Lu, and C. Gill. RT-Xen: Towards Real-time Hypervisor Scheduling in Xen. In *Proceedings of the 11th International Conference on Embedded Software (EMSOFT'11)*, Taipei, Taiwan, October 2011.

[48] C. Xu, S. Gamage, H. Lu, R. Kompella, and D. Xu. vTurbo: Accelerating Virtual Machine I/O Processing Using Designated Turbo-Sliced Core. In *Proceedings of the USENIX 2013 Annual Technical Conference (ATC'13)*, San Jose, CA, USA, June 2013.

[49] C. Xu, S. Gamage, P. N. Rao, A. Kangarlou, R. Kompella, and D. Xu. vSlicer: Latency-Aware Virtual Machine Scheduling via Differentiated-Frequency CPU Slicing. In *Proceedings of the 21st ACM International Symposium on High Performance Distributed Computing (HPDC'12)*, Delft, The Netherlands, June 2012.

[50] Y. Xu, M. Bailey, F. Jahanian, K. Joshi, M. Hiltunen, and R. Schlichting. An Exploration of L2 Cache Covert Channels in Virtualized Environments. In *Proceedings of the 3rd ACM Cloud Computing Security Workshop (CCSW'11)*, Chicago, IL, USA, October 2011.

[51] Y. Xu, Z. Musgrave, B. Noble, and M. Bailey. Bobtail: Avoiding Long Tails in the Cloud. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI'13)*, Lombard, IL, April 2013.

[52] D. Zats, T. Das, P. Mohan, D. Borthakur, and R. Katz. DeTail: Reducing the Flow Completion Time Tail in Datacenter Networks. In *Proceedings of the ACM SIGCOMM 2012 conference*, Helsinki, Finland, August 2012.