Silo: Predictable Message Latency in the Cloud

Keon Jang
Intel Labs
Santa Clara, CA

Justine Sherry*
UC Berkeley
Berkeley, CA

Hitesh Ballani
Microsoft Research
Cambridge, UK

Toby Moncaster*
University of Cambridge
Cambridge, UK

ABSTRACT

Many cloud applications can benefit from guaranteed latency for their network messages, however providing such predictability is hard, especially in multi-tenant datacenters. We identify three key requirements for such predictability: guaranteed network bandwidth, guaranteed packet delay and guaranteed burst allowance. We present Silo, a system that offers these guarantees in multi-tenant datacenters. Silo leverages the tight coupling between bandwidth and delay: controlling tenant bandwidth leads to deterministic bounds on network queuing delay. Silo builds upon network calculus to place tenant VMs with competing requirements such that they can coexist. A novel hypervisor-based policing mechanism achieves packet pacing at sub-microsecond granularity, ensuring tenants do not exceed their allowances. We have implemented a Silo prototype comprising a VM placement manager and a Windows filter driver. Silo does not require any changes to applications, guest OSes or network switches. We show that Silo can ensure predictable message latency for cloud applications while imposing low overhead.

CCS Concepts

•Networks → Data center networks; Cloud computing;

Keywords

Network QoS; Latency SLA; Guaranteed Latency; Traffic Pacing; Network Calculus

1. INTRODUCTION

Many cloud applications are distributed in nature; they comprise services that communicate across the network to generate a response. Often, the slowest service dictates user perceived performance [1,2]. To achieve predictable performance, such applications need guaranteed latency for network messages. However, the consequent network requirements vary with the application. For example, real-time analytics [3,4] and Online Data-Intensive (OLDI) applications like web search and online retail [1,2] generate short messages that are sensitive to *both* packet delay and available network bandwidth. For large message applications like data-parallel jobs [5–7], message latency depends only on network bandwidth.

However, today's public cloud platforms are agnostic to these network demands. Studies of major cloud providers have found that both the bandwidth and packet delay across their network can vary by an order of magnitude [8–12]. As we show in §2.1, this can lead to significant variability in application performance, particularly at the tail. Such performance variability is a key barrier to cloud adoption [13], especially for OLDI applications. For example, moving Netflix to Amazon EC2 required re-architecting their whole system since "AWS networking has more variable latency" [14]. It also complicates application design as programmers are forced to use ad-hoc mitigation techniques [15,16].

We thus try to answer the question: What minimum set of network knobs will allow each tenant to *independently* determine the maximum latency for messages between its virtual machines (VMs)? We identify three key requirements to achieve this, (i) guaranteed bandwidth; (ii) guaranteed packet delay; and (iii) guaranteed burst allowance so that applications can send multi-packet bursts at a higher rate. Of course, some applications may only need some or even none of these guarantees.

These network requirements pose a few challenges. Guaranteeing packet delay is particularly difficult because delay is an additive end-to-end property. We focus on network delay, and argue that in datacenters, queuing is the biggest contributor to it (and its variability). Bounding queuing delay is hard because it requires precise control over how independent flows interact with each other at all network switches along their paths. A further challenge is that guaranteed packet delay is at odds with the guaranteed burst requirement. Allowing some applications to send traffic bursts can increase delay experienced by others; synchronized bursts can even cause packet loss.

*Work done during internship at Microsoft Research

SIGCOMM '15, August 17 - 21, 2015, London, United Kingdom
© 2015 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-3542-3/15/08…$15.00
DOI: http://dx.doi.org/10.1145/2785956.2787479

Unfortunately, none of the existing work on cloud network performance meets all three requirements (§7). Guaranteeing tenant bandwidth alone does not cater to bursty applications with small messages [17–21]. Solutions for low network latency [15,22,23] or flow deadlines [24–26] cannot guarantee message latency in multi-tenant settings.

We present Silo, a system that allows datacenter operators to give tenants guaranteed bandwidth, delay and burst allowances for traffic between their VMs. The key insight behind Silo's design is that end-to-end packet delay can be bounded through fine-grained rate control at end hosts. Cruz's pioneering work [27,28] described a calculus for computing delay bounds for multi-hop networks while Parekh and Gallagher [29,30] developed tighter bounds when switches perform weighted fair queuing.

The tight coupling between guaranteed bandwidth and delay also underlies a lot of research on Internet QoS [31]. However, applying it to cloud settings poses new challenges and opportunities. First, unlike Internet QoS, cloud providers have the added flexibility of controlling the placement of traffic sources. Second, instead of guaranteeing per-flow bandwidth, today's cloud pricing model dictates per-tenant guarantees. Finally, the delay bound analysis relies on strict enforcement of burst and bandwidth limits which is hard to achieve at high link rates. For example, enforcing a 9 Gbps rate limit, on a packet-by-packet basis, for a source sending 1.5KB packets on a 10 Gbps link requires an inter-packet spacing of just $133ns$.

Silo's two components, a VM placement manager and a hypervisor-based packet pacer, address these challenges. Overall, this paper makes the following contributions:

- We identify the network guarantees necessary for predictable message latency, and present the design, implementation and evaluation of Silo.
- We present a novel admission control and VM placement algorithm that uses network calculus to efficiently map tenants' multi-dimensional network guarantees to two simple constraints regarding switch queues.
- We design a software packet pacer for fine-grained rate limiting of VM traffic. To achieve sub-microsecond packet spacing while imposing low CPU overhead, the pacer uses a novel technique called Paced IO Batching that couples IO batching with "void" packets—dummy packets that are forwarded by the NIC but dropped by the first hop switch. This allows the pacer to space out actual data packets without NIC support.

An important feature of Silo's design is ease of deployment. It does not require changes to network switches, NICs, tenant applications or guest OSes. Silo can also leverage switch-based prioritization to accommodate best effort tenants that do not need any guarantees. We have implemented a Silo prototype comprising a VM placement manager and a Windows Hyper-V kernel driver for packet pacing. The network driver is able to saturate 10Gbps links with low CPU overhead while achieving a minimum packet spacing of $68ns$. Through testbed experiments, we show that Silo ensures low and predictable message latency for memcached [32] without hurting bandwidth hungry workloads (§6.1). Packet-level simulations show that Silo improves message latency by 22× at the 99th percentile (and 2.5× at the 95th) as compared to DCTCP [15] and HULL [22] (§6.2).

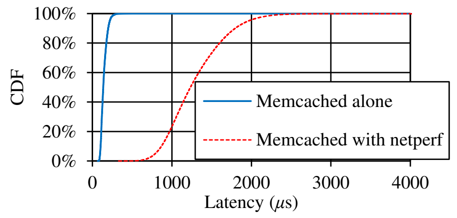


Figure 1: CDF of memcached request latency with and without competing traffic.

A drawback of Silo is its impact on network utilization. Giving delay *and* bandwidth guarantees to tenants can result in up to 11% reduction in network utilization and 4% fewer accepted tenants compared to giving only bandwidth guarantees. Surprisingly, we find that Silo can actually improve both the network and overall cloud utilization than when tenants get no guarantees at all (§6.3).

2. NETWORK REQUIREMENTS

We define a message as a unit of application data, comprising one or more packets, exchanged across the network.[1] Throughout the paper, we use message "latency" as the time to transfer a complete message and "delay" for the packet delay. In this section, we show that network contention can hurt application performance significantly. We also argue why predictable message latency is important for cloud applications and derive the consequent network requirements.

2.1 Motivation

Many recent studies of the internal network in cloud datacenters show that the network bandwidth and packet delay between VMs can vary significantly [8–11]. In recent work, Mogul et al. [12] find that, across two cloud providers, the 99.9th percentile packet delay is typically an order of magnitude more than the median.

To demonstrate the impact of such network-level variability on application performance, we ran a simple experiment with memcached, a popular in-memory key-value store [33]. We use a testbed with five servers, each running six VMs (more details in §6.1). There are two tenants, A and B, each with three VMs per server. Tenant A runs a memcached server on one VM, and clients on all other VMs. Its workload mimics Facebook's ETC workload, and represents general cache usage of multiple cloud applications [33]. Tenant B generates all-to-all TCP traffic using netperf, modeling the shuffle phase of a bandwidth intensive application like MapReduce.

[1]We use the term "message" instead of the more commonly used "flow" since cloud applications often multiplex many messages onto a single flow (i.e. transport connection).

Figure 1 shows the CDF for the latency of Tenant A's memcached requests with and without Tenant B. We find that request latency increases significantly due to network contention; at 99^{th} percentile, it is $270\mu s$ for memcached alone whereas it is $2.3ms$ with interfering traffic. At 99.9^{th} percentile, it suffers from timeouts and latency spikes to $217ms$. Similarly, QJUMP [34] shows that network contention can hurt typical cloud workloads; it is shown to worsen tail performance by 50× for clock synchronization using PTPd and 3× for Naiad, an iterative data-flow application.

2.2 Importance of Predictable Message Latency

For some cloud applications, poor performance directly hurts revenue. More broadly, many cloud applications need to meet service-level latency objectives (SLOs) [12,16]. For example, OLDI applications need to serve end user requests within a time budget, typically 200-300ms [15]. Similarly, users want their data analytics jobs finished in a timely fashion which, given the pay-per-hour cloud pricing model, also ensures predictable job cost [13,35].

To achieve such predictable performance, cloud applications require *predictable message latency*. This is a consequence of two features common to many applications: they rely on distributed execution and messaging latency is a significant fraction of the latency for application tasks. For example, in Facebook's Hadoop cluster [36], 33% of job completion time is due to messaging.

By contrast, to cope with variable network performance in today's cloud platforms, designers resort to ad-hoc techniques like introducing artificial jitter when sending parallel messages [15], replicating request messages and choosing the fastest response [16], etc. Unpredictable network performance even restricts many applications to dedicated clusters. Providing message latency guarantees, apart from helping such applications migrate to the cloud, can obviate the need for band-aid solutions [34]. It can even simplify application logic — if a web-search task that needs to respond in 20ms knows that the response latency is at most 4ms, it can take 16ms to process the query.

2.3 Deconstructing message latency

An open question is the kind of network guarantees that cloud providers can offer and that tenants will benefit from—message latency guarantees, tail latency SLOs, or lower-level guarantees (like bandwidth and packet delay)? We argue that the latter option is most suitable for Infrastructure-as-a-Service (IaaS) cloud providers, and as explained below, can be used by tenants to determine latency bounds for their messages.

The latency for a message comprises the time to transmit its packets into the network and the time for the last packet to propagate to the destination. This simple model excludes end host stack delay. Thus,

$$\text{Msg. Latency} \approx \text{Transmission delay} + \text{In-network delay}$$

$$\approx \frac{\text{Message size}}{\text{Network bandwidth}} + \text{In-network delay} \qquad (1)$$

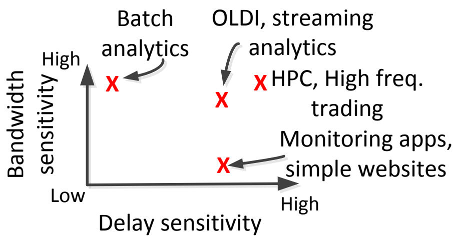


Figure 2: Diverse network demands of cloud apps.

Burst \ Bandwidth	B	$1.4B$	$1.8B$	$2.2B$	$2.6B$	$3B$
M	99	77	55	45	38	33
$3M$	99	22	8	3.6	1.9	1.1
$5M$	99	6.1	0.9	0.2	0.06	0.02
$7M$	99	1.6	0.09	0.01	0	0
$9M$	98	0.4	0.01	0	0	0

Table 1: Percentage of messages whose latency exceeds the guarantee. Messages (size M) have Poisson arrivals and an average bandwidth requirement of B.

For large messages, the latency is dominated by the transmission delay component. This applies to data-parallel applications [5,6] which are thus termed "bandwidth-sensitive". By contrast, applications that send sporadic single packet messages are "delay-sensitive" as the in-network delay component dominates. Examples include monitoring services and simple web services. Between these two extremes are applications that mostly generate small messages but at a high rate. For example, the network traffic for a typical OLDI application like web search is dominated by 1.6–50KB messages [15,24]. For such applications, both components of equation 1 are significant, i.e. the message latency depends *both* on network bandwidth and delay.

Figure 2 summarizes these arguments; some cloud applications are sensitive to available bandwidth, some are sensitive to in-network delay, some to both. Thus, to achieve guaranteed message latency, a general cloud application requires:

R1. *Guaranteed network bandwidth*. This bounds the transmission delay component of message latency.

R2. *Guaranteed packet delay*. This bounds the in-network delay component of message latency.

2.3.1 Burstiness requirement

Requirements R1 and R2, together, are necessary and sufficient to ensure guaranteed message latency. However, this necessitates that an application's bandwidth guarantee match it maximum consumption. This will lead to significant network under-utilization since many cloud applications have bursty workloads, i.e. their maximum bandwidth requirement significantly exceeds the average bandwidth.

To understand the interplay between the amount of guaranteed bandwidth and message latency, we experiment with a synthetic application that generates messages, with Poisson arrivals, between two VMs. Assuming the average bandwidth required is B, the latency for a message of size M

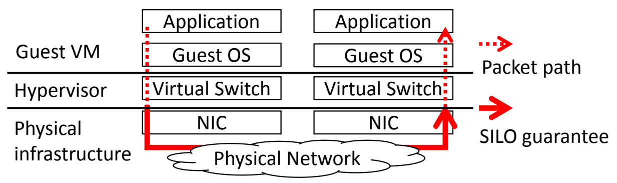


Figure 3: Silo's delay guarantee is from the NIC at the sender to the NIC at the receiver.

should be less than $\frac{M}{B} + d$, where d is the packet delay guarantee (see eq. 1). The first row in Table 1 shows that when the application is guaranteed its average bandwidth requirement, the latency for 99% of messages *exceeds* their guarantee. This is due to exponential inter-arrival times between messages. Increasing the bandwidth guarantee, by itself, is insufficient; even when the guarantee is 3× the average bandwidth, 33% messages are late.

To accommodate such workload burstiness, we propose a burst allowance for VMs. Specifically, a VM that has not been using its guaranteed bandwidth in the past is allowed to send a few messages at a higher rate. Table 1 shows that as we increase both the sending VM's burst allowance and its bandwidth guarantee, the percentage of late messages drops sharply; with a burst of 7 messages and 1.8x the average bandwidth, only 0.09% messages are late. Thus, the third requirement for guaranteed message latency is–

R3. *Guaranteed burst allowance*. Bursty workload applications need to be able to send short traffic bursts.

3. SCOPE AND DESIGN INSIGHTS

Of the three network requirements, guaranteeing packet delay is particularly challenging because all components of the end-to-end path add delay. Below, we define the scope of our delay guarantees and provide intuition regarding the feasibility of guaranteeing packet delay.

Scope. In virtualized datacenters, the end-to-end path of packets comprises many layers. Figure 3 shows this path; at the sender, packets go through the guest OS network stack, the hypervisor and the NIC before being sent onto the wire. Broadly, a packet's total delay comprises two components: end-host delay (shown with dashed line), and network delay (shown with solid line). The latter component includes delay at the NIC and at network switches. Many proposals reduce end-host delay [37,38]. Since we target IaaS cloud settings where tenants can deploy arbitrary OSes, we restrict our focus to the part of the end-to-end path controlled by the cloud provider. Thus, we guarantee network delay, i.e. the delay from the source NIC to the destination NIC.

3.1 Guaranteeing network delay

Network delay comprises the propagation, forwarding and queuing delay across NICs and the network. In datacenters, physical links are short and have high capacity, so the propagation and forwarding delay is negligible, and queuing delay dominates (even for full bisection networks [39,40]).

We build on the observation that *rate limiting the senders in a network ensures a deterministic upper bound for network queuing* [28,30,41]. For example, consider n flows across at a network link. Each flow is guaranteed some bandwidth and is allowed to burst one packet at a time. Assuming the total bandwidth guaranteed across all flows is less than the link capacity, the maximum queue build up at the link is n packets. While Silo shares this insight with Internet QoS architectures like IntServ [31], it further tackles two new challenges posed by cloud settings.

1. Tenant-level guarantees and VM placement. Instead of flow-level guarantees, Silo gives tenants their own private network with tenant-level guarantees. Furthermore, Silo uses the flexibility of VM placement to maximize the number of tenants that can be accommodated. To this end, we extend Cruz's network calculus techniques to derive delay bounds based on tenant-level bandwidth guarantees, and then use them to drive the placement of VMs (§4.2).

2. Fine-grained pacing. Our queuing delay analysis assumes that VM traffic is in strict conformance to its guarantees. Thus, end hosts need to control the rate and burstiness of their traffic at a packet-level timescale. Today's software pacers are typically inaccurate and do not scale with number of flows [42]. The problem is exacerbated by the fact that, to achieve good forwarding performance, network stacks rely on aggressive batching of packets sent to the NIC. This further contravenes the fine-grained pacing requirement.

The obvious solution of pacing at the NIC itself is impractical because NICs only offer small number of rate limited queues, and flows that share the same hardware queue can suffer from head of line blocking. SENIC proposes a hardware and software hybrid approach to achieve scalable and accurate pacing [42]. Instead, we devise a software-only pacing mechanism that uses dummy or "void" packets to precisely control packet gap while retaining I/O batching to handle traffic at 10 Gbps (§4.3.1).

4. DESIGN

Silo couples virtual machines (VMs) with guarantees for network bandwidth, packet delay and burst allowance. It relies on two components: a *VM placement manager* with visibility of the datacenter topology and tenants' guarantees, and a *packet pacer* in the hypervisor at each server. The placement manager admits tenants and places their VMs across the datacenter such that their guarantees can be met (§4.2), and configures the pacers. The pacers coordinate with each other and dynamically determine the rate limit for individual VMs, thus ensuring that VM traffic conforms to their bandwidth and burst allowance (§4.3).

4.1 Silo's network guarantees

With Silo, tenants can imagine their VMs as being connected by a private "virtual" network, as shown in Figure 4. A virtual link of capacity B and propagation delay $\frac{d}{2}$ connects each VM to a virtual switch. Each VM's traffic is shaped by a virtual token bucket with average bandwidth B and size S. The network capabilities of a VM are thus captured using three parameters, $\{B, S, d\}$:

(i). a VM can send and receive traffic at a maximum rate of B Mbps,

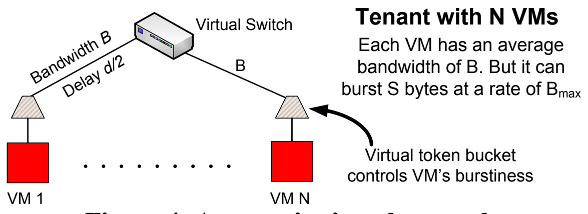

Virtual Switch
Tenant with N VMs
Each VM has an average bandwidth of B. But it can burst S bytes at a rate of B_{max}
Bandwidth B
Delay d/2
B
Virtual token bucket controls VM's burstiness
VM 1
VM N

Figure 4: A tenant's virtual network.

(ii). a VM that has under-utilized its bandwidth guarantee is allowed to send a burst of at most S bytes,

(iii). a bandwidth-compliant packet is guaranteed to be delivered, from the sending to receiving NIC, within $d\ \mu s$.

Just as today's cloud providers offer a few classes of VMs (small, medium, etc.), we expect providers will offer a few classes of network guarantees. Tools like Cicada [43] allow tenants to automatically determine their bandwidth guarantees. Inferring delay requirements automatically may be feasible too [12]. Some tenants may only need bandwidth guarantees; for example, a tenant running a data-analytics job. In §4.4, we show that Silo can leverage switch-based prioritization to accommodate tenants without *any* network guarantees and ensure they co-exist with tenants with guarantees.

Guarantee semantics. We have chosen our guarantees to make them useful for tenants yet practical for providers. As with past proposals [18,19,44,45], a VM's bandwidth guarantee follows the *hose model*, i.e. the bandwidth for a flow is limited by the guarantee of *both* the sender and receiver VM. So if a tenant's VMs are guaranteed bandwidth B, and N VMs send traffic to the same destination VM, each sender would achieve a bandwidth of ^{B}

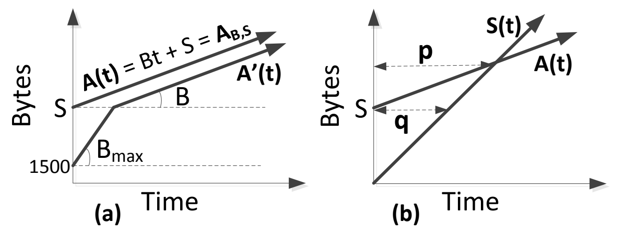


Figure 6: (a) Two arrival curves. (b) An arrival curve, A(t) and a switch's service curve, S(t).

lay guarantees lead to the second queuing constraint. These constraints then dictate VM placement.

In the following sections, we detail our placement algorithm. We assume a multi-rooted tree network topology prevalent in today's datacenters. Such topologies are hierarchical; servers are arranged in racks that are, in turn, grouped into pods. Each server has a few slots where VMs can be placed.

4.2.2 Queue bounds

We begin by describing how we use basic network calculus concepts [28,41,46] to determine the queue bounds for network switches. We then take advantage of Silo's tenant-level bandwidth guarantees to tighten the delay bounds, and use this as a building block for our VM placement algorithm.

Source Characterization. Traffic from a VM with bandwidth guarantee B and burst size S is described by a *arrival curve* $A(t) = Bt + S$, which provides an upper bound for traffic generated over a period of time. We will refer to this curve as $A_{B,S}$. This arrival curve is shown in figure 6(a) and assumes that the VM can send a burst of S bytes instantaneously. While we use this simple function for exposition, our implementation uses a more involved arrival curve (labelled A' in the figure) that captures the fact that a VM's burst rate is limited to B_{max}.

Calculating queue bound. Arrival curves can be used to determine queue bounds for network switches. Just as traffic arriving at a switch is characterized by its arrival curve, each switch port is associated with a *service curve* that characterizes the rate at which it can serve traffic. Figure 6(b) illustrates how these two functions can be used to calculate the maximum queuing at the port or its queue bound. Initially the arrival curve $A(t)$ is larger than the service curve $S(t)$ because of initial burst of traffic; so the queue starts to build up. However, as time reaches $t = p$, the aggregate traffic that the switch can serve exceeds the aggregate traffic that can arrive. This means that at some point during the interval $(0, p]$ the queue must have emptied at least once.

The horizontal distance between the curves is the time for which packets are queued. Hence, the maximum queuing delay experienced by a packet at the switch is q, the maximum horizontal distance between the curves (i.e., the largest q such that $S(t) = A(t - q)$). This is the port's queue bound.

The analysis above allows us to calculate the queue bound at a switch directly receiving traffic from a VM. Next we describe how arrival curves can be added (when traffic from VMs merges at a switch) and propagated across switches to determine the queuing across a multi-hop network.

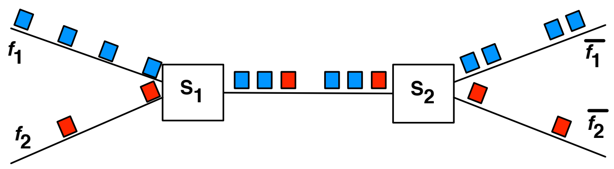


Figure 7: Switch S_1 causes packet bunching for flow f_1.

Adding arrival curves. Arrival curves for VMs can be combined to generate an aggregate arrival curve. For example, adding arrival curves $A_{B1,S1}$ and $A_{B2,S2}$ yields $A_{B1+B2,S1+S2}$. However, as explained below, the semantics of our guarantees allow us to generate a tighter arrival curve when adding curves for VMs belonging to the same tenant.

Consider a tenant with N VMs, each with an average bandwidth B and burst allowance S. The arrival curve for each VM's traffic is $A_{B,S}$. Imagine a network link that connects the tenant's VMs such that m VMs are on the left of the link and the remaining $(N - m)$ are on the right. We want to add the m arrival curves for the VMs on the left to generate an aggregate curve for all traffic traversing the link from left to right. Our choice of bandwidth guarantees using hose-model implies that the total bandwidth guaranteed for the tenant across the link is min$(m, N - m)$*B [18]. By contrast, burst allowances are not destination limited, so the maximum traffic burst across the link from left to right is $m{*}S$ bytes. Thus, instead of $A_{mB,\ mS}$, we can derive a tighter (i.e. more accurate) aggregate arrival curve: $A_{min(m,N-m)*B,\ mS}$.

Propagating arrival curves. After traffic egresses a switch, it may no longer be shaped according to the properties it arrived at the switch with. For example, consider Figure 7: flow f_1 has a rate of $C/2$ (link capacity is C), and flow f_2 has a rate of $C/4$. Both have a burst size of one packet; so f_1's arrival curve is $A_{\frac{C}{2},1}$ and f_2's is $A_{\frac{C}{4},1}$. However, as shown in the figure, f_1's burst size is doubled as it egresses the switch. Its arrival curve changes to $A_{\frac{C}{2},2}$. Note that a flow's average bandwidth cannot change through a switch, only the burst size is impacted.

To compute the arrival curve for a flow after it egresses a switch, we use Kurose's analysis [41] which shows that the upper-bound on the burst introduced by a switch port depends on its p value—the maximum interval over which its queue must be emptied at least once (see Fig. 6). However, this analysis means that the arrival curve for egress traffic depends on a port's p value which, in turn, depends on other flows using the port. To make the egress arrival curve independent of competing traffic, we use the port's queue capacity c as a looser bound on the egress burst size (to avoid packet loss, $p \leq q$). A port's queue capacity is a static value given by the size of its packet buffer, but can be set to a lower value too. In the worst case, every packet sent by a VM over the interval $[0, c]$ may be forwarded as one burst. Since a VM with arrival curve $A_{B,S}$ can send at most $B.c + S$ bytes in time c, the egress traffic's arrival curve is $A_{B,(B.c+S)}$.

4.2.3 Placement algorithm

We have designed a placement algorithm that uses a greedy first-fit heuristic to place VMs on servers. We first describe how we map a new tenant's network guarantees to two simple queuing constraints at switches. These constraints thus characterize a valid VM placement and guide the algorithm.

Valid placement. For the tenant's bandwidth guarantees to be met, we must ensure that network links carrying its traffic have sufficient capacity. Further, VMs can send traffic bursts that may temporarily exceed link capacities. Switch buffers need to absorb this excess traffic, and we must ensure that switch buffers never overflow. In combination, these restrictions imply that for each switch port between the tenant's VMs, the maximum queue buildup (queue bound) should be less than the buffer size (queue capacity). Formally, if V is the set of VMs being placed and $Path(i, j)$ is the set of ports between VMs i and j, the first constraint is

$$\text{Q-bound}_p \;\leq\; \text{Q-capacity}_p, \quad \forall p \in Path(i,j),\; i,j \in V$$

For packet delay guarantees, we must ensure that for each pair of VMs belonging to the new tenant, the sum of queue bounds across the path between them should be less than the delay guarantee. However, a port's queue bound changes as tenants are added and removed which complicates the placement. Instead, we use a port's queue capacity, which always exceeds its queue bound, to check delay guarantees. Thus, for a tenant with delay guarantee d, the second constraint is

$$\sum_{p \in Path(i,j)} \text{Q-capacity}_p \;\leq\; d, \quad \forall i,j \in V$$

Finding valid placements. A request can have many valid placements. Given the oversubscribed nature of typical datacenter networks, we adopt the following optimization goal: *find the placement that minimizes the "height" of network links that may carry the tenant's traffic*, thus preserving network capacity for future tenants. Servers represent the lowest height of network hierarchy, followed by racks and pods. We place a tenant's VMs while greedily optimizing this goal. First, we attempt to place all requested VMs in the same server. If the number of VMs exceeds the number of VM slots per server, we attempt to place all VMs in the same rack—for each server inside the rack, we use the queuing constraints on the server's uplink switch port to determine the number of VMs that can be placed at the server. If all requested VMs can be accommodated at servers within the rack, the request is accepted. Otherwise we consider the next rack and so on. If the request cannot be placed in a single rack, we attempt to place it in a pod and finally across pods.

Other constraints. An important concern when placing VMs in today's datacenters is fault tolerance. Our placement algorithm can ensure that a tenant's VMs are placed across some number of fault domains. For example, if each server is treated as a fault domain, we will place the VMs across two or more servers. Beyond this, VM placement may need to account for other goals such as ease of maintenance, reducing VM migrations, etc. Commercial placement managers like Microsoft's Virtual Machine Manager model these as constraints and use multi-dimensional bin packing heuristics to place VMs [47]. Our queuing constraints could be added to these systems; we defer an exploration to future work.

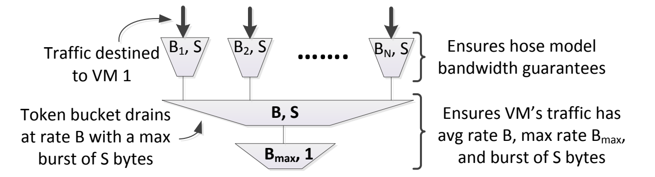

Traffic destined to VM 1
B_1, S
B_2, S
....
B_N, S
Ensures hose model bandwidth guarantees
Token bucket drains at rate B with a max burst of S bytes
B, S
Ensures VM's traffic has avg rate B, max rate B_{max}, and burst of S bytes
B_{max}, 1

Figure 8: VM traffic is paced using a hierarchy of token buckets to ensure it conforms to network guarantees, i.e. its bandwidth guarantee (B) and burst allowance (S).

4.3 End host pacing

Silo's VM placement relies on tenant traffic conforming to their bandwidth and burstiness specifications. To achieve this, a *pacer* at the end host hypervisor paces traffic sent by each VM. Figure 8 shows the hierarchy of token buckets used by the pacer to enforce traffic conformance. The bottom most token bucket ensures a VM's traffic rate can never exceed B_{max}, even when sending a burst. The middle token bucket ensures the average traffic rate is limited to B and the maximum burst size is S bytes. At the top is a set of token buckets, one each for traffic destined to each of the other VMs belonging to the same tenant. These are needed to enforce the hose model semantics of guaranteed bandwidth; i.e. the actual bandwidth guaranteed for traffic between a pair of VMs is constrained by both the sender and the destination. To enforce the hose model, the pacers at the source and destination hypervisor communicate with each other like EyeQ [20]. This coordination determines the rate B_i for the top token buckets in Figure 8 such that $\sum B_i \leq B$.

4.3.1 Paced IO Batching

The placement of VMs assumes their traffic strictly conforms to their arrival curve. Thus, the token buckets should ideally be serviced at a per-packet granularity with accurate spacing between the packets. For example, if a VM with a 1 Gbps rate limit sends 1.5KB packets on a 10 Gbps link, the packets should be separated by $10.8\mu s$. This precludes the use of I/O batching techniques since today's NICs transmit an entire batch of packets back-to-back [22]. With a typical batch size of 64KB used by offloading techniques like TSO, all packets for our example VM would be sent at line rate. This increases the amount of network queuing and can even result in packet loss. However, disabling IO batching results in high CPU overhead and reduces throughput; in experiments with batching disabled, we cannot even saturate a 10Gbps link. One solution is to implement pacing at the NIC itself [22,42]. However, this requires hardware support. For ease of deployment, we design a software solution.

We propose Paced IO batching, a technique that allows us to reduce the IO overhead while still pacing packets at sub-microsecond timescales. To achieve this, we use "**void packets**" to control the spacing between data packets forwarded by the NIC. A void packet is a packet that will be forwarded by the NIC but discarded by the first switch it encounters. This can be achieved, for example, by setting the packet's destination MAC address the same as the source MAC. Such void packets do not increase the power consumption at the NIC and switch because most of power is

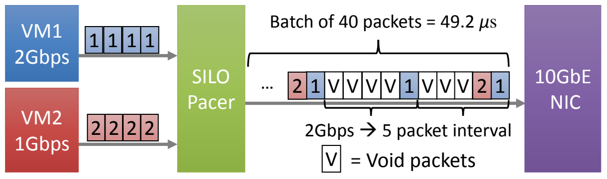

VM1
2Gbps
1 1 1 1
VM2
1Gbps
2 2 2 2
SILO
Pacer
Batch of 40 packets = 49.2 μs
... 2 1 V V V V 1 V V V 2 1
2Gbps → 5 packet interval
V = Void packets
10GbE
NIC

Figure 9: VM1's guarantee is 2Gbps, VM2's is 1Gbps. Use of void packets achieves packet level pacing in the face of NIC burstiness.

consumed for keeping the link active [48]. We note that injecting void packets requires disabling TCP Segmentation Offload (TSO), however VMs can still send large segments to the hypervisor to minimize the overhead for VM-to-hypervisor communication.

Figure 9 illustrates how we use void packets. The link capacity is 10Gbps and VM1 is guaranteed 2Gbps, so every fifth packet sent to the NIC belongs to VM1. In every batch of 40 packets sent to the NIC, twelve are actual data packets, while the rest 28 are void packets. While the NIC forwards the entire batch of packets as is, all void packets are dropped by the first hop switch, thus generating a correctly paced packet stream. The minimum size of a void packet, including the Ethernet frame, is 84 bytes. So, at 10Gbps, we can achieve an inter-packet spacing as low as $68ns$.

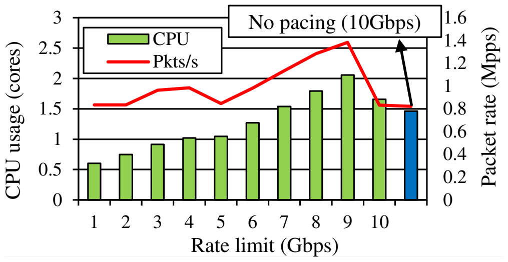
4.4 Tenants without guarantees

Silo relies on rate limiting tenants to give packet delay guarantees. However, this can hurt network utilization since tenants cannot use spare network capacity. We evaluate this in §6.3. We note that some cloud applications are indeed not network-limited, so they do not need any network guarantees. Such "best-effort tenants" can be used to improve network utilization. Silo leverages 802.1q priority forwarding in switches to support best-effort tenants. Traffic from such tenants is marked by our pacer as low priority while traffic from tenants with guarantees is higher priority. Thus, such tenants share the residual network capacity. It is also possible for tenant with guarantees to be able to send low priority traffic to achieve higher throughput by using schemes like RC3 [49]. We leave this for future work.

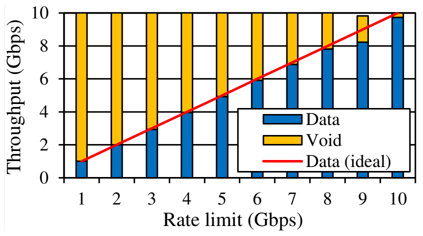
5. IMPLEMENTATION

We have implemented a Silo prototype comprising a VM placement manager and a software pacer implemented as a Windows NDIS filter driver. The pacer driver sits between the virtual switch (vswitch) and the NIC driver, so we do not require any modification to the NIC driver, applications or the guest OS. It implements token buckets and supports token bucket chaining. We use virtual token buckets, i.e. packets are not drained at an absolute time, rather we timestamp when each packet needs to be sent out. This requires an extra eight bytes in each packet's metadata. The overhead is negligible in comparison to the size of the packet buffer structure: 160 bytes in Windows NET BUFFER and 208 bytes in Linux skb [50,51].

At high link rates, I/O batching is essential to keep the CPU overhead low. For accurate rate limiting with I/O batching, we need two key properties. The first is to keep the precise gap between packets within a batch; we achieve this using void packets. The second is to schedule the next batch of packets before the NIC starts to idle. This is essential to guarantee burst allowance but challenging since we want to keep the batch size small so that NIC queuing delay is limited. We borrow the idea of soft-timers [52] and reuse existing interrupts as a timer source. Our pacer does not use a separate timer, but triggers sending the next batch of packets upon receiving a DMA (Direct Memory Access) completion interrupt for transmit packets. We use a batch size of 50 μs when pulling out packets from the token buckets.



(a) CPU usage

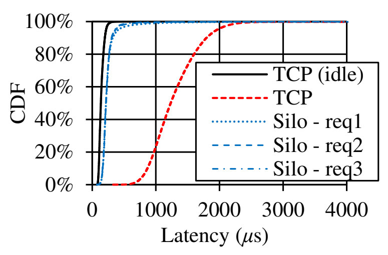


(b) Throughput

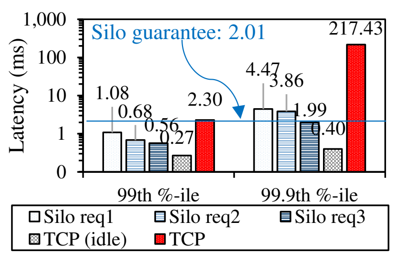
Figure 10: Microbenchmarks show Silo's pacer is able to saturate 10Gbps links with low CPU overhead.

Pacer microbenchmarks. We evaluate our pacer implementation in terms of throughput and the CPU overhead. We use physical servers equipped with one Intel X520 10GbE NICs, and two Intel Xeon E5-2665 CPUs (8 cores, 2.4Ghz). Overall, we find that the pacer is able to saturate 10Gbps links with low CPU overhead.

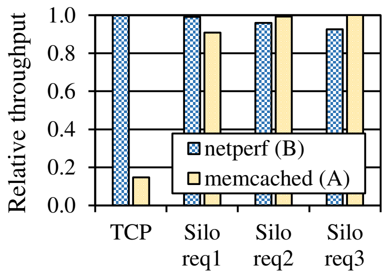
Figure 10(a) shows the CPU usage of the entire system by varying the rate limit imposed by the pacer. The right most bar is CPU usage when the pacer is disabled. LSO is disabled in all cases. The red solid line represents the number of transmitted packets per second, including void packets. The pacer consumes 0.6 cores to generate only void packets at 10 Gbps. As the actual data rate increases, the overall CPU utilization goes up to ≈2.1 cores worth of CPU cycles at 9 Gbps. The reason is that at 9 Gbps, the pacer needs to put $1/10^{th}$ of MTU sized packets (150 bytes) between all the data packets, which results in high packet rate. The graph shows that the overall CPU usage is proportional to the packet rate shown in the red line. At full line-rate of 10 Gbps, our pacer incurs less than 0.2 cores worth of extra CPU cycles as compared to no pacing. Since void packets are generated only when there is another packet waiting to be sent, the pacer does not incur any extra CPU overhead when the network is idle.



(a) Memcached latency CDF



(b) Memcached tail latency



(c) Throughput

Figure 11: Testbed experiments involving two tenants, A and B, with 15 VMs each. Silo achieves lower and predictable message latency for memcached, even with a competing bandwidth-hungry workload generated using netperf.

	Req 1		Req 2		Req 3	
Tenant	A	B	A	B	A	B
Bandwidth ($BMbps$)	210	3,123	315	3,018	420	2,913
Burst (S)	1.5KB	1.5KB	1.5KB	1.5KB	1.5KB	1.5KB
Packet Delay (d)	1ms	N/A	1ms	N/A	1ms	N/A
Burst rate (B_{max})	1Gbps	N/A	1Gbps	N/A	1Gbps	N/A

Table 2: Tenant network guarantees for the testbed experiments. We set combined bandwidth at each host ($3 \times (B_A + B_B)$) to be link capacity (10Gbps).

In Figure 10(b), we show the throughput for both void packets and data packets. Except at 9 Gbps, the pacer sustains 100% of the link capacity, and achieves actual data rate at more than 98% of the ideal rate.

Placement microbenchmarks. To evaluate the scalability of the placement manager, we measured the time to place tenants in a simulated datacenter with 100K hosts with an average tenant requesting 49 VMs (as in [18,45]). Over 100K requests, the maximum placement time is 1.15s.

6. EVALUATION

We evaluate Silo across three platforms: a small scale prototype deployment, a medium scale packet-level simulator, and a datacenter scale flow-level simulator. The key findings are as follows:

(i). Through testbed experiments with memcached, we verify that our prototype can offer bandwidth, delay and burst guarantees. This ensures improved and predictable performance for memcached without impacting bandwidth sensitive workloads (who get predictable performance too).

(ii). Through *ns2* simulations, we show that Silo significantly improves tail message latency as compared to state-of-the-art solutions like DCTCP [15], HULL [22], and Oktopus [18]; none of these achieve predictable message latency.

(iii). Through flow-level simulations, we evaluate our VM placement algorithm, and find that it can result in a small reduction in accepted tenants and network utilization as compared to when tenants only get bandwidth guarantees. Surprisingly, as compared to the status quo, Silo can improve both network and overall cloud utilization.

6.1 Testbed experiments

We deployed our prototype across five physical servers connected to a 10GbE switch. Each server is equipped with either two Intel Xeon E5-2630 (6 cores, 2.6Ghz) or two E5-2650 (8 cores, 2.6Ghz) processors, and an Intel X520 10GbE NIC. Our testbed experiments comprise two tenants, A and B, each with 15 VMs. We launch six VMs per server, and assign three to each tenant. Tenant A runs the ETC trace of Facebook workloads [33] using memcached. We generate value sizes and inter arrival times using generalized pareto distribution with parameters from the trace [33]. Tenant B runs netperf to generate TCP traffic emulating the shuffle phase in mapreduce.

Guarantees. To determine Tenant A's network requirements, we measured the bandwidth and packet sizes by running memcached in isolation. The average packet size is around 400 B, and the average value size in our workload is 300 B; the maximum value size is 1 KB. A burst allowance of $1.5KB$ is thus enough to handle a burst of three to four requests on average. The average bandwidth requirement (B_{avg}) is 210 $Mbps$. Since the instantaneous bandwidth required can vary, we experimented with varying bandwidth guarantees for tenant A. This allows us to evaluate the tradeoff between memcached request latency and overall utilization. The tenants' guarantees are shown in Table 2. Requirements 1, 2, and 3 in Table 2 allocate B_{avg}, $1.5 \times B_{avg}$, and $2 \times B_{avg}$ respectively for Tenant A, and the remaining bandwidth for Tenant B.

Memcached latency. We plot the CDF of latency for memcached in Figure 11(a). Figure 11(b) shows the 99^{th} and 99.9^{th} percentile latency. We measure latency for five different scenarios: (a) Tenant A and B running simultaneously with the three different configurations for Silo in Table 2 (Silo req 1–3); (b) Tenant A running memcached in isolation (labelled as "TCP (idle)"); and (c) Tenant A and B running simultaneously without using Silo (labelled as "TCP"). TCP incurs very high latency compared to memcached running in isolation. Figure 11(b) shows it even suffers timeouts at 99.9^{th} percentile resulting in over 200 ms latency.

In these experiments, the message latency guarantee for memcached with Silo is 2.01 ms. Silo stays well within this guarantee even at 99^{th} percentile. At 99.9^{th} percentile, Silo stays within the message latency guarantee when tenant A is guaranteed twice its average bandwidth requirement (req 3). However, it exceeds the guaranteed latency by a little when the bandwidth guarantee is lower (req 1–2). This is due to the bursty nature of the memcached workload. We also experimented with a higher burst allowance of 3 KB, and

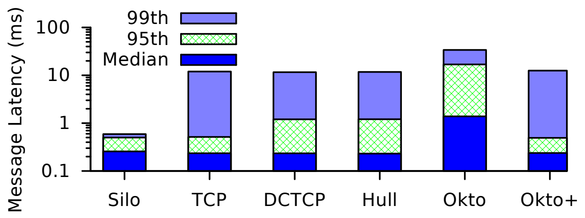


Figure 12: Message latency for class-A tenants.

	Class A	Class B
Communication pattern	All to 1	All to all
Bandwidth (B)	0.25Gbps	2Gbps
Burst length (S)	15KB	1.5KB
Delay guarantee (d)	1000μs	N/A
Burst rate (B_{max})	1Gbps	N/A

Table 3: Tenant classes and their average network requirements.

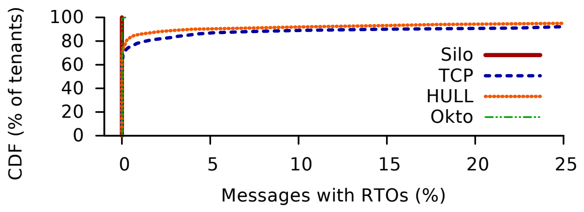


Figure 13: Class-A tenants whose messages incur RTOs.

	Silo	TCP	DCTCP	Hull	Okto	Okto+
Outliers-1x	0	23.1	47.3	46.9	91	20.2
Outliers-2x	0	21.7	17.4	15.6	80.7	19.1
Outliers-8x	0	21.5	14.4	14.4	36.5	19

Table 4: Outlier tenants, i.e. tenants whose 99th percentile message latency exceeds the latency estimate.

found that it lowers memcached's 99.9^{th} percentile latency to 4.3 ms, 2.5 ms, and 1.8 ms for req-1–3 respectively.

Throughput. We measured the throughput of memcached and netperf to evaluate whether Silo's use of strict bandwidth reservation impacts network utilization. In Figure 11(c), we plot the relative throughput of both Silo and TCP against when Tenant A and Tenant B are running alone with standard TCP. For TCP, the netperf traffic gets 94% of the link capacity while memcached transaction rate drops by seven times. This is because memcached does synchronous transactions with limited concurrency, and inflated latency leads to low throughput even when network bandwidth is available. When Silo is used, Tenant A's throughput increases as we increase its reservation, and reaches 100% for req-3. However, at the same time, Tenant B can still use 92% to 99% of bandwidth achieved by TCP alone.

Overall, we find that Silo delivers guarantees for packet delay, burst and bandwidth. This significantly improves the performance of a latency sensitive application like memcached without hurting bandwidth hungry applications. Moreover, both sets of applications get predictable performance.

6.2 Packet level simulations

We use *ns2* to compare Silo against state-of-the-art solutions. Instead of using two specific tenants, we model two classes of tenants. *Class-A* contains delay-sensitive tenants that run a small message application, and require bandwidth, delay and burst guarantees. Each class-A tenant has an all-to-one communication pattern such that all VMs simultaneously send a message to the same receiver. This coarsely models the workload for OLDI applications [15]. *Class-B* tenants only require bandwidth guarantees. Such tenants have an all-to-all communication pattern, as is common for data parallel applications. The bandwidth and burst requirements of tenants are generated from an exponential distribution with the parameters in table 3.

Simulation setup. We model 10 racks, each with 40 servers and 8 VMs per server, resulting in 3200 VMs. We use a multi-rooted tree topology for the cloud network. The capacity of each network link is 10Gbps and the network has an oversubscription of 1:5, as is the norm in datacenters [53]. We model commonly used shallow buffered switches [54] with 312KB buffering per port (queue capacity is 250μs). The number of tenants is such that 90% of VM slots are occupied. For Silo, VMs are placed using its placement algorithm. For Oktopus, we use its bandwidth-aware algorithm [18]. For other solutions, we use a locality-aware algorithm that greedily places VMs close to each other.

Class-A tenants. Figure 12 shows the message latency for class-A tenants across 50 runs. Silo ensures low message latency even at the 99th percentile while all other approaches have high tail latency. With Oktopus, VMs cannot burst, so the message latency is high, both at the median and at the tail. At 99th percentile, message latency is 60× higher with Oktopus as compared to Silo. "Okto+" is an Oktopus extension with burst allowance. Hence, it reduces the median latency but still suffers at the tail. This is because it does not account for VM bursts when placing VMs which can lead to switch buffer overflows (see example in §4.2). We note that TCP is used as the transport protocol for Silo, Okto, and Okto+ on top of their rate enforcement. Using DCTCP with Okto+ still suffers from packet losses and TCP timeouts as the placement of VMs does not ensure switch buffers can actually absorb traffic bursts.

With DCTCP and HULL, message latency is higher by 22x at the 99th percentile (and 2.5x at the 95th). Two factors lead to poor tail latency for these approaches. First, class-A tenants have an all-to-one traffic pattern that leads to contention at the destination. Second, none of these approaches isolate performance across tenants by guaranteeing bandwidth, so class-A small messages compete with large messages from class-B tenants. This leads to high tail latency and losses for small messages. Figure 13 shows that with TCP, for 21% of class-A tenants, more than 1% of messages suffer retransmission timeout events (RTOs). With HULL, this happens for 14% of tenants. Thus, *by itself, neither low queuing (ensured by DCTCP and HULL) nor guaranteed bandwidth (ensured by Oktopus) is sufficient to ensure predictable message latency.*

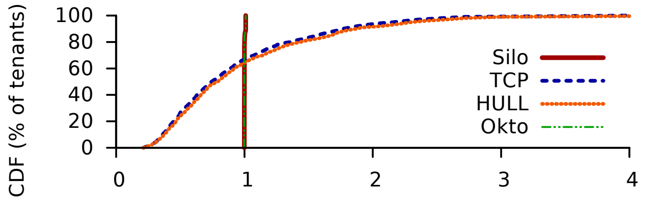


Figure 14: Message latency for class-B tenants.

We also look at *outlier tenants*, i.e. class-A tenants whose 99^{th} percentile message latency is more than the latency estimate. Table 4 shows that Silo results in no outliers. Since we observe outliers with other approaches, we mark the fraction of outliers whose latency exceeds the estimate by 1x, 2x or 8x. With DCTCP and HULL, 14.4% tenants are 8x outliers.

Class-B tenants. Since Silo does not let tenants exceed their bandwidth guarantee, it can impact the performance of class-B tenants with large messages whose completion is dictated by the bandwidth they achieve. Figure 14 shows the average message latency for class-B tenants, normalized to the message latency estimate. For clarity, we omit the DCTCP (similar to HULL) and Okto+ lines. With Silo and Oktopus, tenant bandwidth is guaranteed, so all large messages finish by the estimated time. With TCP and HULL, the message latency varies. 65% of tenants achieve higher bandwidth with HULL as compared to Silo but there is a long tail with many tenants getting very poor network bandwidth. This shows how Silo trades-off best-case performance for predictability.

6.3 Large-scale simulations

We use a flow-level simulator to evaluate Silo's VM placement algorithm and its impact on network utilization at scale.

Simulation setup. We model a public cloud datacenter with 32K servers connected using three tier network topology with 1:5 oversubscription at each level. The arrival of tenant requests is a Poisson process. By varying the average arrival rate, we can control the average datacenter occupancy. Each tenant runs a job that transfers a given amount of data between its VMs. Each job also has a minimum compute time. A job is said to finish when all its flows finish and the compute time has expired. 50% tenants belong to class-A while the rest belong to class-B. We compare Silo's placement against two approaches: *Oktopus* placement that guarantees VM bandwidth only and locality-aware placement (*Locality*) that greedily places a tenant's VM close to each other. For Silo and Oktopus, a tenant's flows are assigned bandwidth based on its guarantee with no bandwidth sharing across tenants, while for greedy placement, we emulate ideal TCP behavior by sharing bandwidth fairly between all flows.

Workload. As before, Class-A tenants have an all-to-one workload. To study the impact of different traffic patterns, Class-B tenants have a "Permutation-x" workload—each VM has flows to x randomly chosen other VMs. By varying x, we control the tenant's traffic pattern. Note that for a tenant with N VMs, Permutation-N is an all-to-all traffic pattern.

Admittance ratio. We first use a Permutation-1 workload for class-B tenants. Figure 15(a) shows the fraction of tenants admitted with 75% datacenter occupancy. Silo rejects 4.5% of tenants while locality-aware placement accepts all of them and Oktopus rejects 0.3%. This is because Silo ensures that both the delay and bandwidth requirements of tenants are met, and can reject tenants even if there are empty VM slots. With Silo, the rejection ratio is higher for Class-A tenants as their delay requirements are harder to meet.

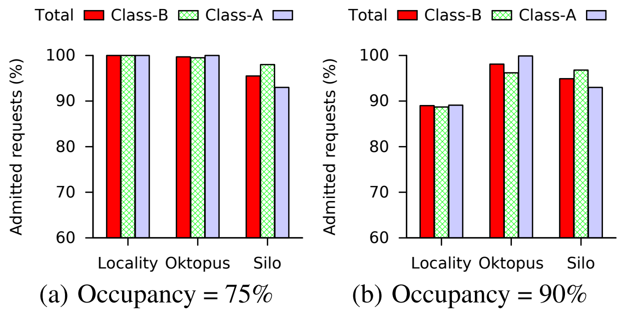


Figure 15: Admitted requests with Permutation-1 workload for class-B tenants.

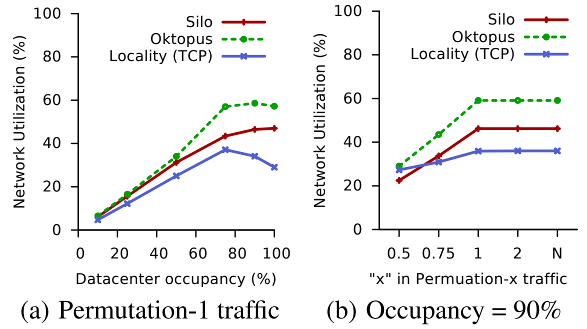


Figure 16: Average network utilization with varying datacenter occupancy and varying traffic patterns.

However, as the datacenter occupancy increases and tenants arrive faster, the admittance ratio of the locality-aware placement drops. Figure 15(b) shows that at 90% occupancy, it rejects 11% of tenants as compared to 5.1% rejects by Silo. This result is counter-intuitive. Locality-aware placement will only reject requests if there are insufficient VM slots. By contrast, Silo can reject a request, even when there are empty VM slots, if the request's network guarantees cannot be met. The root cause is that locality-aware placement does not account for the bandwidth demands of tenants. So it can place VMs of tenants with high bandwidth requirements far apart. Such tenants get poor network performance and their jobs get delayed. These outlier tenants reduce the *overall* cloud throughput, causing subsequent requests to be rejected. With Silo, tenants get guaranteed bandwidth, so no tenant suffers from poor network performance.

Network utilization. With Silo, tenants cannot exceed their bandwidth guarantees which can result in network under-utilization. We study Silo's impact on network utilization with varying occupancy and traffic patterns, and find that, surprisingly, Silo can actually match and even improve utilization as compared to the status quo.

Figure 16(a) shows the average network utilization in our experiments with varying datacenter occupancy. As low oc-

		Unmodified			Guarantee				Cloud support		Work conservation		Fine-grained pacing
		H/W	OS	App	Bandwidth	Packet delay	Burst	Message latency	Placement	Tenant Isolation	Intra tenant	Inter tenant	
	Silo	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗[†]	✓
Packet delay guarantee	QJUMP [34]	✓	✓	✓	✓	✓	✗	✓	✗	✗	✓	✓	✗
	Fastpass [23][♯]	✓	✓	✓	✓	✗	✗	✗	✗	✗	✓	✓	✓
	TDMA Ethernet [55]	✓	✗	✓	✓	✓	✗	✓	✗	✗	✓	✓	✗
Internet QoS	ATM [56]	✗	✗	✗	✓	✓	✓	✓	✗	✓[⋆]	✓	✓	✗[‡]
	IntServ [31]	✗	✗	✗	✓	✓	✓	✓	✗	✓[⋆]	✗	✗	✗
	Stop and Go [57]	✗	✗	✗	✓	✓	✓	✓	✗	✓[⋆]	✗	✗	✗[‡]
Bandwidth guarantee	Oktopus [18]	✗	✗	✓	✓	✗	✗	✗	✓	✓	✓	✗	✗
	Hadrian [21]	✗	✗	✓	✓	✗	✗	✗	✓	✓	✓	✓	✗
	EyeQ [20]	✓	✓	✓	✓	✗	✗	✗	✓	✓	✓	✓	✗
Flow deadline	D^3 [24]	✗	✗	✗	✗	✗	✓	✓	✗	✗	✓	✓	✗
	PDQ [26]	✗	✗	✗	✗	✗	✓	✓	✗	✗	✓	✓	✗
	D^2TCP [25]	✓	✗	✗	✗	✗	✓	✓	✗	✗	✓	✓	✗
Low latency	pFabric [58]	✗	✗	✓	✗	✗	✓	✗	✗	✗	✓	✓	✗[‡]
	DCTCP [15]	✓	✗	✓	✗	✗	✗	✗	✗	✗	✓	✓	✗
	HULL [22]	✗	✗	✓	✗	✗	✗	✗	✗	✗	✓	✓	✓

[†]Silo's guarantee is not work-conserving across tenants, however Silo can use switch priorities to support tenants without guarantees and achieve work-conservation. [‡]These do not require nor benefit from fine-grained pacing at the end host. [⋆]Although these systems are not designed for the cloud, their support for traffic classes can achieve tenant isolation. [♯]Fastpass's centralized packet scheduling can support any policy in theory. However, it does not provide scheduling mechanisms for guarantees, and its scalability to datacenters is unknown.

Table 5: Comparison of related work in terms of design choices.

cupancy, there is not much difference between the approaches. At an occupancy of 75%, network utilization with Silo is actually 6% higher than with locality-aware placement (which uses an ideal-TCP for bandwidth sharing). This results from two factors. First, locality-aware placement naturally minimizes network traffic by co-locating VMs close to each other while Silo may be forced to place VMs across servers to meet their guarantees. Second, as mentioned above, a small fraction of outlier tenants that get poor network performance with the locality approach also hurt network utilization. As compared to Oktopus, Silo's network utilization is lower by 9-11% at high occupancy. *This is the price we pay for accommodating delay requirements of class-A tenants as it causes Silo to accept fewer requests than Oktopus and reduces network utilization.*

Figure 16(b) shows the utilization with varying traffic patterns for class-B tenants. When such tenants have a sparse traffic matrix (eg., Permutation-0.5 traffic), Silo can result in $\sim$4% drop in utilization. However, as the tenant traffic matrix becomes denser, network utilization with Silo surpasses locality-aware placement.

Other simulation parameters. We repeated these experiments while varying other simulation parameters. We omit the results but highlight a couple of trends. The requests admitted by Silo when datacenter occupancy is lower than 75% are similar to the ones shown in Figure 15(a). As we increase the size of switch buffers or reduce network oversubscription, Silo's ability to accept tenants increases since the network is better provisioned.

7. RELATED WORK

Silo adds to a rich literature on network guarantees and optimization. We briefly discuss the work most relevant to Silo, and summarize the key differences in Table 5.

Many recent efforts look at cloud network sharing. They propose different sharing policies, including fixed guarantees [17,18], minimum bandwidth guarantees [19,20], and per-tenant fairness [59]. However, these proposals focus solely on bandwidth, thus catering to bandwidth-sensitive applications. They do not provide bounded packet delay and the ability to burst.

QJUMP [34] is the closest to Silo among recent works; it uses rate limiting and priorities, and allows different traffic classes at different points of the trade off between network latency and throughput. However, it only provides a latency guarantee for very low bandwidth traffic; higher bandwidth traffic can incur variable message latency. By contrast, Silo provides an intuitive abstraction for tenants described by bandwidth, delay, and burst guarantees. It ensures isolation between tenants, and a tenant can independently determine it message latency. In terms of mechanisms, Silo additionally uses a novel placement algorithm to maximize the number of tenants that can be accommodated.

Numerous solutions achieve low latency in private datacenters by ensuring small network queues [15,22], by accounting for flow deadlines [24–26], through network prioritization [34,58] or centralized rate allocation [23]. Silo targets predictable message latency in multi-tenant datacenters and three key factors differentiate our work. First, these proposals do not isolate performance across tenants; i.e., they do not guarantee a tenant's bandwidth nor control total burstiness on network links. The former hurts the latency of both small and large messages while the latter hurts small messages. In §6.2, we show that DCTCP and HULL can suffer from high and variable message latency, especially when there is competing tenant traffic. Similarly, with D^3 [24], PDQ [26] and D^2TCP [25], there is no guarantee for a message's latency as it depends on the deadlines of messages of other tenants. And a tenant can declare tight deadlines for its messages, thus hurting other tenants.

Second, with public clouds, we cannot rely on tenant cooperation. Tenants can use the transport protocol of their choice and cannot be expected to pass application hints like deadlines. Thus, a hypervisor-only solution is needed. Finally, apart from HULL [22], none of these solutions look at queuing delay at the NIC itself. Overall, while these solutions work well for the class of applications or environment they were designed for, they do not guarantee end-to-end message latency for diverse cloud applications.

The early 1990's saw substantial work on providing network performance guarantees to Internet clients. ATM [56] and its predecessors [60] provided different kinds performance guarantees. Stop and Go [57] is the proposal most similar to Silo in terms of architecture. However, it introduces a new queueing discipline at switches while Silo does not require any switch changes. More broadly, much of the work from this era focuses on novel queueing disciplines at switches [61–63]; Zhang and Keshav [64] provide a comparison of these proposals. We chose to avoid this in favor of better deployability. Silo works with commodity Ethernet. Further, Silo leverages the flexibility of placing VMs, an opportunity unique to datacenters. Finally, Silo pushes to much higher performance in terms pacing traffic; by using void packets, Silo can achieve sub-microsecond granularity pacing with very low CPU overhead. However, Silo can still benefit from using hardware offloading like SENIC [42] which eliminates the overhead of pacing and disabling TSO.

8. CONCLUSION

In this paper we target predictable message latency for cloud applications. We argue that to achieve this, a general cloud application needs guarantees for its network bandwidth, packet delay and burstiness. We show how guaranteed network bandwidth makes it easier to guarantee packet delay. Leveraging this idea, Silo enables these guarantees without any network and application changes, relying only on VM placement and end host packet pacing. Our prototype can achieve fine grained packet pacing with low CPU overhead. Evaluation shows that Silo can ensure predictable message completion time for both small and large messages in multi-tenant datacenters.

Acknowledgements. We thank Richard Black and Austin Donnelly for the suggestion regarding the packet pacing method. We thank the anonymous reviewers and our shepherd Teemu Koponen for their helpful comments. We are grateful to Dongsu Han, Sangjin Han, Thomas Karagiannis, Keunhong Lee, Sue Moon, Aurojit Panda, Greg O'Shea, and Andy Slowey for discussions and help with experiments.

9. REFERENCES

[1] V. Jalaparti, P. Bodik, S. Kandula, I. Menache, M. Rybalkin, and C. Yan. Speeding up Distributed Request-Response Workflows. In *Proc. of ACM SIGCOMM*, 2013.

[2] 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. *ACM SIGOPS*, 41(6), 2007.

[3] C. Engle, A. Lupher, R. Xin, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: Fast Data Analysis Using Coarse-grained Distributed Memory. In *Proc. of ACM SIGMOD*, 2012.

[4] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: interactive analysis of web-scale datasets. In *Proc. of VLDB*, 2010.

[5] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proc. USENIX OSDI*, 2004.

[6] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *Proc. EuroSys*, 2007.

[7] R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: easy and efficient parallel processing of massive data sets. In *Proc. VLDB*, 2008.

[8] A. Li, X. Yang, S. Kandula, and M. Zhang. CloudCmp: comparing public cloud providers. In *Proc. ACM IMC*, 2010.

[9] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz. Runtime measurements in the cloud: observing, analyzing, and reducing variance. In *Proc. VLDB*, 2010.

[10] Y. Xu, Z. Musgrave, B. Noble, and M. Bailey. Bobtail: Avoiding Long Tails in the Cloud. In *Proc. of USENIX NSDI*, 2013.

[11] Y. Xu, M. Bailey, B. Noble, and F. Jahanian. Small is Better: Avoiding Latency Traps in Virtualized Data Centers. In *Proc. of SoCC*, 2013.

[12] J. C. Mogul and R. R. Kompella. Inferring the Network Latency Requirements of Cloud Tenants. In *Proc. of USENIX HotOS*, 2015.

[13] Michael Armburst et al. Above the Clouds: A Berkeley View of Cloud Computing. Technical report, University of California, Berkeley, 2009.

[14] 5 Lessons We've Learned Using AWS. http://techblog.netflix.com/2010/12/5-lessons-weve-learned-using-aws.html.

[15] M. Alizadeh, A. G. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center TCP (DCTCP). In *Proc. ACM SIGCOMM*, 2010.

[16] J. Dean and L. A. Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, Feb. 2013.

[17] C. Guo, G. Lu, H. J. Wang, S. Yang, C. Kong, P. Sun, W. Wu, and Y. Zhang. SecondNet: A Data Center Network Virtualization Architecture with Bandwidth Guarantees. In *Proc. ACM CoNext'10*.

[18] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Towards Predicable Datacenter Networks. In *Proc. ACM SIGCOMM*, 2011.

[19] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica. FairCloud: Sharing the Network In Cloud Computing. In *Proc. ACM SIGCOMM*, 2012.

[20] V. Jeyakumar, M. Alizadeh, D. MaziÃÍres, B. Prabhakar, and C. Kim. EyeQ: Practical Network Performance Isolation at the Edge. In *Proc. USENIX NSDI*, 2013.

[21] H. Ballani, K. Jang, T. Karagiannis, C. Kim, D. Gunawaradena, and G. O'Shea. Chatty Tenants and the Cloud Network Sharing Problem. In *Proc. USENIX NSDI*, 2013.

[22] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vadhat, and M. Yasuda. Less is More: Trading a little Bandwidth for Ultra-Low Latency in the Data Center. In *Proc. USENIX NSDI*, 2012.

[23] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal. Fastpass: A Centralized Zero-Queue Datacenter Network. In *Proc. of ACM SIGCOMM*, 2014.

[24] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowstron. Better Never than Late: Meeting Deadlines in Datacenter Networks. In *Proc. ACM SIGCOMM*, 2011.

[25] B. Vamana, J. Hasan, and T. N. Vijaykumar. Deadline-Aware Datacenter TCP (D^2TCP). In *Proc. ACM SIGCOMM*, 2012.

[26] C.-Y. Hong, M. Caesar, and P. B. Godfrey. Finishing Flows Quickly with Preemptive Scheduling. In *Proc. ACM SIGCOMM*, 2012.

[27] R. Cruz. A Calculus for Network Delay, Part I: Network Elements in Isolation. *IEEE Transactions on Information Theory*, 37(1):114–131, January 1991.

[28] R. Cruz. A Calculus for Network Delay, Part II: Network Analysis. *IEEE Transactions on Information Theory*, 37(1):132–141, 1991.

[29] A. Parekh and R. Gallager. A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks: The Single-Node Case. *IEEE/ACM Transactions on Networking (ToN)*, 1, June 1993.

[30] A. Parekh and R. Gallager. A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks: The Multiple -Node Case. *IEEE/ACM Transactions on Networking (ToN)*, 1, April 1994.

[31] D. Clark, S. Shenker, and L. Zhang. Supporting real-time applications in an integrated services packet network: Architecture and mechanism. In *Proc. ACM SIGCOMM*, 1992.

[32] P. Saab. Scaling memcached at Facebook, 2008. http://www.facebook.com/note.php?note_id=39391378919.

[33] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload Analysis of a Large-scale Key-value Store. In *Proc. ACM SIGMETRICS*, 2012.

[34] M. P. Grosvenor, M. Schwarzkopf, I. Gog, R. N. M. Watson, A. W. Moore, S. Hand, and J. Crowcroft. Queues don't matter when you can JUMP them! In *Proc. USENIX NSDI*, 2015.

[35] H. Herodotou, F. Dong, and S. Babu. No One (Cluster) Size Fits All: Automatic Cluster Sizing for Data-intensive Analytics. In *ACM SOCC*, 2011.

[36] M. Chowdhury, M. Zaharia, J. Ma, M. Jordan, and I. Stoica. Managing data transfers in computer clusters with orchestra. In *Proc. ACM SIGCOMM*, 2011.

[37] B. Lin and P. A. Dinda. Vsched: Mixing batch and interactive virtual machines using periodic real-time scheduling. In *Proc. ACM/IEEE Supercomputing*, 2005.

[38] C. Xu, S. Gamage, P. N. Rao, A. Kangarlou, R. R. Kompella, and D. Xu. vSlicer: latency-aware virtual machine scheduling via differentiated-frequency CPU slicing. In *Proc. HPDC'12*.

[39] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: a scalable and flexible data center network. In *Proc. of ACM SIGCOMM*, 2009.

[40] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *Proc. of ACM SIGCOMM*, 2008.

[41] J. Kurose. On Computing Per-Session Performance Bounds in High-Speed Multi-Hop Computer Networks. In *Proc. ACM SIGMETRICS*, 1992.

[42] S. Radhakrishnan, Y. Geng, V. Jeyakumar, A. Kabbani, G. Porter, and A. Vahdat. Senic: Scalable nic for end-host rate limiting. In *Proc. USENIX NSDI*, 2014.

[43] K. LaCurts, J. C. Mogul, H. Balakrishnan, and Y. Turner. Cicada: Introducing Predictive Guarantees for Cloud Networks. In *Proc. USENIX HotCloud*, 2014.

[44] N. G. Duffield, P. Goyal, A. Greenberg, P. Mishra, K. K. Ramakrishnan, and J. E. van der Merive. A flexible model for resource management in virtual private networks. In *Proc. ACM SIGCOMM*, 1999.

[45] D. Xie, N. Ding, Y. C. Hu, and R. Kompella. The Only Constant is Change: Incorporating Time-Varying Network Reservations in Data Centers. In *Proc. ACM SIGCOMM*, 2012.

[46] J.-Y. Le Boudec and P. Thiran. *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*. Lecture Notes in Computer Science. Springer-Verlag, 2001.

[47] S. Lee, R. Panigrahy, V. Prabhakaran, V. Ramasubramanian, K. Talwar, L. Uyeda, and U. Wieder. Validating Heuristics for Virtual Machines Consolidation. Technical Report MSR-TR-2011-9, MSR, 2011.

[48] Energy Efficient Switches , 2009. http://www.cisco.com/c/dam/en/us/products/collateral/switches/catalyst-2960-series-switches/cisco_catalyst_switches_green.pdf.

[49] R. Mittal, J. Sherry, S. Ratnasamy, and S. Shenker. Recursively cautious congestion control. In *Proc. USENIX NSDI*, 2014.

[50] NDIS Filter Driver. http://msdn.microsoft.com/en-us/library/windows/hardware/ff556030(v=vs.85).aspx.

[51] Linux SKB. http://lxr.free-electrons.com/source/include/linux/skbuff.h.

[52] M. Aron and P. Druschel. Soft timers: efficient microsecond software timer support for network processing. *ACM Transactions on Computer Systems (TOCS)*, 18(3):197–228, 2000.

[53] Luiz Andrr Barroso and Urs Holzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan & Claypool Publishers, 2009.

[54] 10GE ToR port buffers. http://www.gossamer-threads.com/lists/nanog/users/149189.

[55] B. C. Vattikonda, G. Porter, A. Vahdat, and A. C. Snoeren. Practical tdma for datacenter ethernet. In *Proc. EuroSys*, 2012.

[56] S. Minzer. Broadband isdn and asynchronous transfer mode (atm). *Communications Magazine, IEEE*, 27(9):17–24, 1989.

[57] S. J. Golestani. A Stop-and-Go Queueing Framework for Congestion Management. In *Proc. ACM SIGCOMM*, 1990.

[58] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. pFabric: Minimal Near-Optimal Datacenter Transport. In *ACM SIGCOMM*, 2013.

[59] T. Lam, S. Radhakrishnan, A. Vahdat, and G. Varghese. NetShare: Virtualizing Data Center Networks across Services. Technical Report CS2010-0957, University of California, San Diego, May 2010.

[60] G. Finn. RELIABLE ASYNCHRONOUS TRANSFER PROTOCOL (RATP). RFC 916.

[61] D. Ferrari and D. Verma. A Scheme for Real-Time Channel Establishment in Wide-Area Networks. *IEEE Journal on Selected Areas in Communications*, 8(3):368–379, April 1990.

[62] D. Verma, H. Zhang, and D. Ferrari. Guaranteeing Delat Jitter Bounds in Packet Switching Networks. In *Proc. IEEE Conference on Communications Software (TriComm)*, 1991.

[63] C. R. Kalmanek, H. Kanakia, and S. Keshav. Rate Controlled Servers for Very High-Speed Networks. In *Proc. IEEE Global Telecommunications Conference*, 1989.

[64] H. Zhang and S. Keshav. Comparison of Rate-Based Service Disciplines. In *Proc. ACM SIGCOMM*, 1991.