Silo: Predictable Message Completion Time in the Cloud

Keon Jang[†] Justine Sherry[⋆] Hitesh Ballani[†] Toby Moncaster[‡]

[†]Microsoft Research [⋆]UC Berkeley [‡]University of Cambridge

September, 2013

Technical Report
MSR-TR-2013-95

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
http://www.research.microsoft.com

Abstract

Many cloud applications need predictable completion of application tasks. To achieve this, they require predictable completion time for network messages. We identify three key requirements for such predictability: guaranteed network bandwidth, guaranteed per-packet delay and guaranteed burst allowance. We present Silo, a network architecture for public cloud datacenters that offers these guarantees. Silo leverages the fact that guaranteed bandwidth and delay are tightly coupled: controlling tenant bandwidth yields deterministic bounds on network queuing delay. Silo builds upon network calculus to determine how tenants with competing requirements can coexist, using a novel packet pacing mechanism to ensure the requirements are met.

We have implemented a Silo prototype comprising a VM placement manager and a Windows Hyper-V network driver. Silo does not require any changes to applications, VMs or network switches. We use testbed experiments and large scale simulations to show that Silo can ensure predictable message latency for competing applications in cloud datacenters.

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 predictable tail completion time for network messages. However, the consequent network requirements vary with the application. For example, Online Data-Intensive (OLDI) applications like web search and online retail [1,2], and real-time analytics [3–5] generate small messages whose completion time is sensitive to *both* packet delay and available network bandwidth. For large message applications like data-parallel jobs [6–8], message completion time depends only on network bandwidth.

Today's public cloud platforms are agnostic to these network demands. While tenants can rent virtual machines (VMs) with varying amount of dedicated CPU cores, memory and storage, their network performance is not guaranteed. Studies of major cloud providers have found that both the bandwidth and packet delay across their internal network can vary by an order of magnitude [9–12]. This is recognized as a key barrier to public cloud adoption [13], especially for OLDI applications.

We thus try to answer the question: *What network knobs can the cloud provider expose such that applications can achieve predictable message completion time?* We want any tenant to be able to *independently* answer– "What is the maximum time within which my message will be delivered to its destination?". We identify three key requirements to achieve this, (i) guaranteed bandwidth; (ii) guaranteed per-packet delay; and (iii) guaranteed burst allowance so that applications can send multipacket traffic bursts at a higher rate. Of course, not all applications require all three guarantees; MapReduce jobs only need high bandwidth guarantees with no packet delay bounds and no bursts, while a web-service may need low bandwidth yet low delay bounds and large bursts. Some applications may not need any guarantees.

While many recent efforts target cloud network performance, none meets all these requirements. Guaranteeing tenant network bandwidth [14–20], by itself, does not cater to bursty applications with small messages. Many recent solutions target low network latency by achieving small network queues [21–23] or accounting for flow deadlines [24–26]. However, they are tailored for private data centers where tenants are cooperative and inter-tenant performance isolation is not essential. By contrast, public clouds involve many untrusted and competing tenants, so the assumptions underlying these solutions do not hold. Further, we show that in such multi-tenant settings, low queueing delay is insufficient to ensure predictable message completion. Overall, none of these efforts allow tenants to determine the *end-to-end* completion time of their messages.

Our network requirements pose a few challenges. Guaranteeing per-packet delay is particularly difficult because delay is an end-to-end property; it comprises stack delay at the end hosts and queuing delay inside the network. In both cases, today's design targets high utilization at the expense of delay. Queues at the end host and inside the network ensure good utilization but result in poor tail latency. 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.

In this paper, we present Silo, a network architecture

for public cloud data centers. Silo ensures predictable message completion time by giving tenants guaranteed bandwidth, delay and burst allowances for traffic between their VMs. A simple observation enables these guarantees: traffic that is strictly paced at the rate guaranteed to VMs yields a deterministic upper bound for network queueing. The queuing bound is straightforward for a single hop [27], but is challenging to determine across multiple hops [28–30]. Thus, *bandwidth guarantees, needed for isolating tenant performance anyway, make it easier to bound packet delay at both the end host and inside the network.*

Silo relies on two key mechanisms. First, a *VM placement* algorithm accounts for the network requirements of tenants. It maps the multi-dimensional network guarantees to two simple constraints regarding switch queues. We use techniques from network calculus to check these constraints. Second, we design a *software packet pacing* mechanism that employs per VM-pair rate limiting. This, combined with packet level scheduling in the hypervisor, ensures that the maximum end host queuing for a packet is independent of the number of competing flows. The lack of real-time scheduling at end hosts means our delay guarantees are "soft"; they hold at the 99^{th} percentile, not the maximum.

An important feature of Silo's design is ease of deployment. It does not require changes to network switches, tenant applications or guest OSes. We have implemented a Silo prototype comprising a VM placement manager and a kernel driver in the Windows Hyper-V network stack. The network driver uses "void packets" to pace traffic at sub-microsecond granularity. It can still saturate 10Gbps links with low CPU overhead. Through testbed experiments and simulations, we show that Silo results in low and predictable message completion time even in the presence of competing traffic. It improves message latency by 22x at the 99^{th} percentile (and 2.5x at the 95^{th}) as compared to DCTCP [21] and HULL [22]. We also find that, by carefully placing VMs, Silo can actually improve cloud utilization, thus allowing the provider to accept more tenants.

2 Network Requirements

In this section we argue why predictable message latency is important for cloud applications, and derive key network requirements for such predictability. Note that a message's "latency" is its end-to-end completion time; for packets, we use "delay".

2.1 Predictable message latency

Many cloud applications share two common features. First, they rely on distributed execution. For OLDI applications, answering a single query involves tens or hundreds of workers that exchange small messages across the network [1,2].[1] Data analytics jobs also run on a distributed set of machines that shuffle messages across the cloud network. Messaging latency is a significant fraction of a job's completion time (33% for Facebook's Hadoop cluster [31]). Second, they need timely completion of jobs. For example, OLDI applications need to serve end user requests within a time budget, typically 200-300ms [21]. Similarly, users want their data analytics jobs finished in a timely fashion which, given today's pay-per-hour cloud pricing model, also ensures predictable job cost [13,32].

Thus, to achieve predictable performance, such applications require *predictable message latency*. This holds for OLDI applications [24,26], data analytics [33] and many other cloud applications like HPC, high frequency trading, monitoring services, etc. [34]. Bounded message latency can even simplify application design. For example, in the case of web search, if a worker that has to respond to a query in 20ms knows that the response message will take at most 4ms, it can use 16ms to process the query. Instead, applications today are forced to use conservative latency estimates.

2.2 Deconstructing message latency

The latency for a message comprises the time to transmit the packets into the network and the time for the last packet to propagate to the destination. This simple model assumes an ideal transport protocol and excludes end host stack delay (accounted for in §3.1). Formally,

[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).

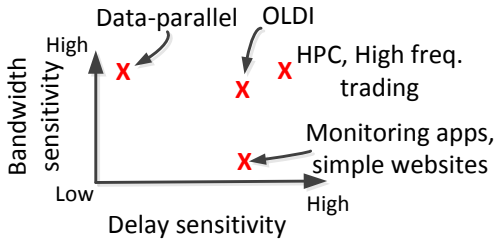


Figure 1: Diverse network demands of cloud applications.

$$\text{Msg. Latency} \approx \text{Transmission delay} + \text{In-network delay}$$
$$\approx \frac{\text{Message size}}{\text{Network bandwidth}} + \text{In-network delay}$$

For large messages, the latency is dominated by the transmission delay component. This applies to data-parallel applications [6,7] 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. For example, the network traffic for a typical OLDI application like web search is dominated by 1.6–50KB messages [21,24]. While traditionally described as delay-sensitive only, the latency for such small messages actually depends *both* on the available bandwidth and the in-network delay. The root cause for this is the small bandwidth delay product in datacenters. For example, on a 10 Gbps link with 40 flows (median concurrent flows for search [21]), the fair share for each flow is 250 Mbps. With a typical datacenter RTT of 200μs, the bandwidth delay product is a mere 6KB! TCP models show that for a TCP flow to reach a congestion window of size w, its size needs to exceed $3w$ [35]. Hence, in the example above, any message exceeding 18 KB will be bandwidth limited. And the situation worsens if more flows compete at the link. Figure 1 summarizes these arguments; some cloud applications are sensitive to available bandwidth, some are sensitive to in-network delay, some to both.

Thus, to achieve predictable message latency, a general cloud application requires: **Requirement 1.** *guaranteed network bandwidth* and **Requirement 2.** *guaranteed per-packet delay*. The former results in a deterministic transmission delay component for a message. The latter bounds the in-network delay component but is only needed by delay-sensitive applications.

2.2.1 Burstiness requirement

For applications that are sensitive to network bandwidth, the instantaneous bandwidth requirement can vary. This is especially true for OLDI applications. For example, let's imagine an application with user requests arriving every a msec. Each request results in a VM generating a single message of size M. Thus, the average bandwidth required by the application is $\frac{M}{a}$. However, the application may actually be required to serve each user request earlier, say within d msec ($d < a$). Hence, the instantaneous bandwidth required by each VM is $\frac{M}{d}$ which ensures that each message finishes by its deadline. Of course, guaranteeing this much bandwidth is possible but inefficient since the average bandwidth required by the application is still $\frac{M}{a}$. This motivates the third requirement for predictable message latency.

Requirement 3. *Burst allowance*. Some applications need to be able to send short traffic bursts. To accommodate this, we borrow the notion of burstiness from token buckets. VMs that have not been using their guaranteed bandwidth should be allowed to burst a few packets at a higher rate.

2.3 Today's public cloud

Many recent studies look at the performance across the internal network in public cloud datacenters. Measurements show that the VM-to-VM network bandwidth can vary by a factor of five or more [9,10,36,37]. Since the underlying network is shared, a VM's bandwidth depends on its placement, the network load and even the nature of competing traffic (UDP/TCP).

Packet delay across the cloud network shows even more variability [11,12,36]. This is because delay is an additive, end-to-end property; the delay for a packet is the sum of delay at the guest OS, the end host hypervisor and NIC, and delay inside the network. Studies show that all these factors contribute to packet delay variability. VM scheduling delay inflates the tail packet delay by 2x [11], hypervisor and NIC delay contributes a 4-6x inflation, while in-network delay increases it by 10x [12].

3 Challenges and Design Insights

Of the three network requirements, guaranteeing per-packet delay is particularly challenging because all com-

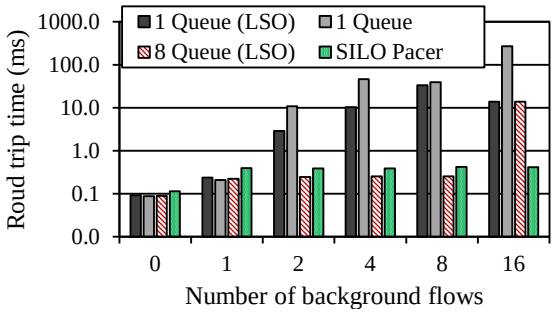


Figure 2: RTT between the hypervisor (vswitch-to-vswitch) with background flows.

ponents of the end-to-end path can add delay. Given our focus on IaaS clouds where tenants can deploy arbitrary OSes, we restrict our delay guarantees to the part of the path the cloud provider controls, i.e. the hypervisor and the network. In this section we highlight the challenges posed by these delay components, and provide intuition regarding the feasibility of guaranteeing packet delay.

3.1 End host delay

Reducing packet delay at the end host is an active area of research. In virtualized settings, such delay can be attributed to two main factors. The first is VM scheduling delay, which is significant when multiple VMs are mapped to the same CPU core. We focus on scenarios without such CPU core oversubscription which avoids VM scheduling delay. This holds for the majority of VM classes offered by cloud providers today; for example, only the micro and small EC2 VMs have CPU oversubscription. Further, recent VM schedulers like vSched [38] and vSlicer [39] can achieve low VM scheduling delay even with CPU oversubscription.

The second factor is queuing delay at the NIC driver. This results from the fact that the driver has one queue corresponding to each hardware queue at the NIC. So if the NIC has a single queue, a small message can get queued behind large messages from other VMs. The use of batching techniques like large send offload (LSO) to reduce CPU load for network I/O exacerbates this by allowing VMs to send large batches of packets to the driver (typically 64KB).

To quantify this problem, we conduct a simple experiment involving two physical servers; each has 10 Gbps Ethernet interfaces and two hexa core CPUs (Intel E5-2665, 2.66Ghz). Figure 2 shows the 99^{th} percentile RTT for 1KB ping-pong messages between the hypervisors on each server. When there is no competing traffic, the RTT is $100\mu s$. However, when background flows increase, the RTT grows beyond 10ms (labeled "1 Queue (LSO)"). This is due to queuing at the NIC driver which increases with the number of competing flows. When the number of NIC queues is increased to 8, the RTT is stable until 8 background flows and then increases as multiple flows use the same queue. As NICs typically support a limited number of hardware queues, simply ensuring that flows are mapped to separate queues is not feasible. And disabling LSO does not solve the problem either. We measured the packet delay with LSO disabled ("1 Queue"). This increases the CPU utilization such that we cannot even saturate the 10Gbps link which, in turn, increases the queuing delay as packets are drained slower.

To reduce driver queuing delay, we use per VM-pair queues at the hypervisor. Since each VM has a bandwidth guarantee, each queue is serviced at a guaranteed rate, and the amount of time any bandwidth-compliant packet spends in this hypervisor queue is bounded. We also ensure that a fixed number of packets are outstanding at the driver, thus bounding the delay at the driver queue. *These two mechanisms ensure that the amount of queuing a packet can incur at the end host is independent of the number of competing flows*. While we detail this pacing technique in §4.3.1, Figure 2 shows that it ensures the RTT is always less than $400\mu s$. Hence, it is feasible to offer one-way packet delay guarantees as low as 200 μs, which is reasonable for OLDI applications. Note that due to the lack of real-time network scheduling support, such delay guarantees are only statistical guarantees based on empirical analysis; thus, we bound the 99^{th} percentile packet delay rather than the worst case.

3.2 In-network delay

In-network delay comprises the propagation, forwarding and queuing delay across the network. In datacenters, the propagation and forwarding delay is negligible, and queuing delay dominates. TCP-like protocols drive the network to congestion by filling up queues. This leads to high and variable in-network delay. To circumvent this, HULL [22] operates the network below capacity and achieves tiny queues. While an elegant solution, it does not provide packet delay and bandwidth guarantees.

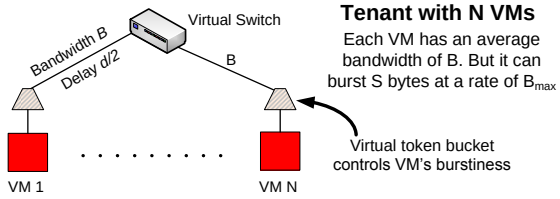


Figure 3: A tenant's virtual network captures its network guarantees. It combines the hose model with per-VM virtual token buckets.

We adopt a different tact to bound in-network delay. As shown in §2.2, cloud applications require guaranteed network bandwidth for predictable message latency. *Ensuring that a VM's traffic is paced at the rate guaranteed to it, in turn yields a deterministic upper bound for network queuing* [28,30]. For example, consider n flows bottlenecked at a network link. Each flow is guaranteed some bandwidth and can only 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. This happens when a packet for each flow arrives at the link at exactly the same time. In §4.2, we build upon this simple observation and use network calculus to quantify the maximum queuing across a multi-hop network.

Overall, by controlling both the end host and in-network delay, we can ensure a tenant's end-to-end packet delay guarantees can be met.

4 Design

We present Silo, an architecture for multi-tenant datacenters that ensures tenants achieve predictable message latency. To this end, VMs are coupled with guaranteed bandwidth, packet delay and burst allowance.

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 3. A virtual link of capacity B and propagation delay $\frac{d}{2}$ connects each VM to a virtual switch. Further, each VM's traffic is shaped by a virtual token bucket with average bandwidth B and size S. Thus, VMs get the following network guarantees–

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

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

(iii). any bandwidth-compliant packet is guaranteed to be delivered from the source hypervisor to the destination hypervisor within d μs.

The network capabilities of a VM are thus captured using three parameters– $\{B, S, d\}$. Just as today's cloud providers offer a limited classes of VMs (small, medium, etc.) with varying compute and memory capacity, we expect providers will offer a few classes with varying network guarantees. Some tenants may only need the bandwidth guarantees. In §4.3.2, we show that Silo can also accommodate tenants without any network guarantees.

The precise network guarantees represent a trade-off between how useful they are for tenants and how practical they are for providers. We have chosen the semantics of the guarantees to balance this trade-off. As with past proposals [16,18,40], our VM 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 all send traffic to the same destination VM, each sender would achieve a bandwidth of $\frac{B}{N}$ (since the destination VM becomes the bottleneck). By contrast, each VM gets an *aggregate* burst guarantee. With such a guarantee, all N VMs are allowed to send a simulataneous burst of S bytes to the same destination. This is motivated by the observation that many OLDI-like applications employ a partition aggregate workflow that results in all-to-one traffic pattern [21].

However, allowing VMs to send traffic bursts can result in high and variable packet delay for VMs of other tenants. Synchronized bursts can even overflow switch buffers and cause packet loss. While Silo carefully places tenants VMs to ensure switch buffers can absorb the bursts, we also control the maximum bandwidth at which a burst is sent. This maximum rate (B_{max}) is exposed to the tenant. In our experiments, B_{max} is 1-2Gbps.

Overall, these network knobs allow tenants to determine their message latency. Consider a VM, that has not used up its burst quota, sends a message of size M ($\leq S$) bytes. The message is guaranteed to be delivered to its destination in less than $(\frac{M}{B_{max}} + d)$ μs. If $M > S$, message latency is less than $(\frac{S}{B_{max}} + \frac{M-S}{B} + d)$. To achieve these guarantees, Silo relies on two key mecha-

nisms: *network-aware VM placement* and *packet pacing* at end host hypervisors. We detail these mechanisms below.

4.2 VM Placement

Silo includes a placement manager that, given a tenant request, places its VMs at servers in the datacenter such their network guarantees requirements can be met. If the guarantees cannot be met, the request is rejected.

4.2.1 Placement overview

Placement of VMs in today's datacenters typically focusses on non-network resources like CPU cores and memory. Recent efforts propose algorithms to place VMs such that their bandwidth guarantees can also be met [15,16]. Silo expands VM network guarantees to include per-packet delay and burst allowance which is critical for OLDI-like applications. Thus, the placement algorithm needs to account for multiple network constraints. The main insight behind our approach is that each VM's bandwidth guarantee yields an upper bound for the rate at which it can send traffic. This allows us to quantify the **queue bound** for any switch port, i.e. the maximum queuing delay that can occur at the port. Further, we can also determine a port's **queue capacity**, the maximum possible queue delay before packets are dropped. For example, a 10Gbps port with a 312KB buffer has a $\approx$250 μs queue capacity.

The key novelty in the placement algorithm is the mapping of multi-dimensional network constraints to two simple queueing constraints at intervening switches. These constraints then dictate the placement of VMs. To ensure the network has enough capacity to accommodate the bandwidth guarantees of VMs *and* absorb all bursts, we need to ensure that at all switch ports, the queue bound does not exceed the queue capacity. This is the first constraint. As we explain later, the packet delay guarantees lead to the second queuing constraint.

In the following sections, we detail our placement algorithm. We assume a multi-rooted tree-like 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. We also assume that if the topology offers multiple paths between VMs, the underlying routing protocol load balances traffic across them. This assumption holds for fat-tree topologies [41,42] that use multi-pathing mechanisms like ECMP, VLB [41] and Hedera [43].

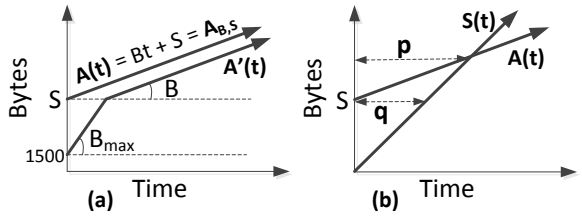


Figure 4: (a) Two arrival curves. (b) An arrival curve, A(t) and a switch's service curve, S(t).

4.2.2 Queue bounds

We begin by describing how we use basic network calculus concepts [28,44] to determine the queue bounds for network switches. This serves as a building block for Silo's 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 4(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. We now show how 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 4(b) illustrates how these two functions can be used to calculate the maximum queuing at the port or its queue bound. We start at $t = 0$ when the switch's queue is empty: 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 port's queue bound is q, the maximum horizontal distance between the curves

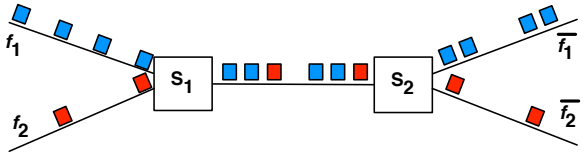


Figure 5: Switch S_1 causes packet bunching for flow f_1.

.

(i.e., the largest q such that $S(t) = A(t-q)$).

This allows us to calculate the queue bound at a switch directly receiving traffic from a VM. Below we describe how arrival curves can be added (when traffic from different VMs merges at a switch) and propagated across switches to determine the queuing at any network switch.

Adding arrival curves. Arrival curves for VMs can be added 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 hose-model bandwidth guarantees implies that the total bandwidth guaranteed for the tenant across the link is min(m, $N-m$)*B [16]. By contrast, each VM is allowed to send an aggregate burst of S bytes, even if all VMs send to the same destination. So the maximum burst of tenant traffic across the link from left to right is $m * S$ bytes. Thus, instead of $A_{mB,\ mS}$, the aggregate arrival curve is actually $A_{min(m,N-m)*B,\ mS}$.

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 5: flow f_1 has a sending rate of $C/2$ (link capacity is C), and flow f_2 has a sending rate of $C/4$. Both have a burst size of one packet; so f_1's arrival function is $A_{\frac{C}{2},1}$ and f_2's is $A_{\frac{C}{4},1}$. At switch S_1, the first packet of both f_1 and f_2 arrive simultaneously; the packet from f_2 is served first followed by the packet from f_1. Immediately after this, a packet from f_1 arrives and is served. This sequence then repeats itself. Thus, f_1's packets are bunched due to queueing at switch S_1 such that after leaving the switch, f_1's arrival function is $A_{\frac{C}{2},2}$. Note that a flow's average bandwidth cannot change with queueing, only the burst size is impacted.

Kurose [28] proved an upper bound for the burst size of traffic egressing a switch. Consider the value p from Fig. 4– the maximum interval over which the queue must be emptied at least once. In the worst case, every packet sent by a VM over the interval $[0, p]$ may be bunched together and forwarded as one burst. However, this analysis makes placement decisions computationally untenable; the arrival curve for egress traffic depends on the port's p value which, in turn, depends on other flows using the port. To ensure that the new tenant's VMs do not cause the guarantees of existing tenants to be violated, we potentially would need to recalculate the queue bound for every switch port in the datacenter for every new placement.

Instead, we ensure that the p value on a port can never exceed its queue capacity c and then use the queue capacity to determine a (looser) upper bound on the egress traffic's burst size. 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 $A_{B,(B.c+S)}$. Since a port's queue capacity is a static value, a valid addition of new VMs cannot cause the guarantees of already existing VMs to be violated.

4.2.3 Placement algorithm

We have designed a placement algorithm that uses a greedy first-fit heuristic to place VMs on servers. It can accommodate tenants that need all network guarantees as well as those that only need guaranteed bandwidth. We first describe how we map the network guarantees to two simple queuing constraints at switches. These constraints thus characterize a valid VM placement and guide the algorithm.

Valid placement. Past placement algorithms that account for VM bandwidth guarantees focus on ensuring that the bandwidth for all traffic that may traverse a network link is less than the link's capacity. With Silo, we further need to account for the fact that VMs can send traffic bursts at a rate that may temporarily exceed a link's ca-

pacity. The buffer at switches needs to absorb this excess traffic. To ensure there is enough capacity to accommodate the bandwidth guarantees of VMs and absorb bursts, we must make sure that switch buffers never overflow. Thus, 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 tenant, the sum of queue bounds across the path between them should be less than the 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. Consider a tenant whose packets, after accounting for end host delay, should be delayed by at most d'. Thus, the second constraint is

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

Given that today's switches are shallow buffered with low queue capacity, this approximation is not too conservative. It also makes the placement of delay sensitive tenants much simpler. For example, say delay-sensitive tenants get a 1ms packet delay guarantee and each switch port has a 250μs queue capacity. Assuming end host delay is at most 200μs, the in-network delay for any packet should not exceed 800μs. Thus, VMs for any such tenant cannot be placed more than three hops apart.

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 "level" of network links that may carry the tenant's traffic*, thus preserving network capacity for future tenants. Servers represent the lowest level of network hierarchy, followed by racks, pods and the cluster.

The algorithm psuedo code is shown in Algorithm 4.1. We first attempt to place all requested VMs at the same server (lines 4–6). If the number of VMs exceeds the empty VM slots on the server, we attempt to place all VMs in the same rack. To do this, for each server inside the rack, we use the queuing constraints on the

Algorithm 4.1 Silo's Placement Algorithm

```
1: Ensures: Placement for request with N VMs and {B, S, d} network guarantees.
2: Requires: Topology tree T with pods, racks and hosts. Pre-calculated state includes delayQuota[d, l] which is the maximum packet delay at each link for a request with e2e delay guarantee d that is allocated at level l. We also have serverUpDelay and serverDownDelay for servers, and similarly for racks and pods.
3:
4: if N < VMSlotsPerServer then
5:     return AllocOnServer(request)
6: end if
7:
8: for each l ∈ [0, T.height - 1] do
9:     for each p ∈ T.pods do
10:        vmsPerPod = 0
11:        for each r ∈ p.racks do
12:            vmsPerRack = 0
13:            for each s ∈ r.servers do
14:                v = CalcValidAllocations(s.emptySlots, N, request,
15:                    s.upLink, serverUpDelay[d, l], serverDownDelay[d, l],
16:                    delayQuota[d, l])
17:                vmsPerRack += v
18:                if vmsPerRack >= N and l==0 then
19:                    return AllocOnRack(r, request)
20:                end if
21:            end for
22:            if l > 0 then
23:                v = CalcValidAllocations(vmsPerRack, N, request,
24:                    r.UpLink, rackUpDelay[d, l], rackDownDelay[d, l],
25:                    delayQuota[d, l])
26:                vmsPerPod += v
27:                if v >= N and l==1 then
28:                    AllocOnPod(p, request)
29:                end if
30:            end if
31:        end for
32:        if l > 1 then
33:            v = CalcValidAllocations(vmsPerPod, N, request,
34:                p.UpLink, podUpDelay[d, l], podDownDelay[d, l],
35:                delayQuota[d, l])
36:            if v >= N and l==2 then AllocOnCluster(request)
37:            end if
38:        end if
39:    end for
40: end for
41: function CALCVALIDALLOCATIONS(k, N, request, uplink, updelay, downdelay, delay)
42:    for each m ∈ [k,1] do
43:        if (uplink.GetMaxDelay(request, (N-m), N, updelay) < delay and uplink.reverse.GetMaxDelay(request, m, N, downdelay) < delay) then
44:            return m
45:        end if
46:    end for
47: end function
```

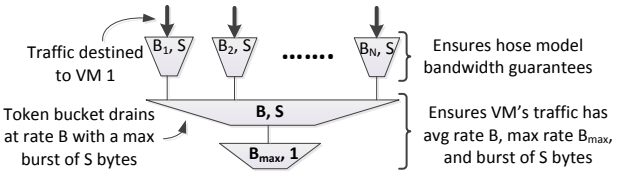


Figure 6: VM traffic is paced using a hierarchy of token buckets to ensure it conforms to network guarantees.

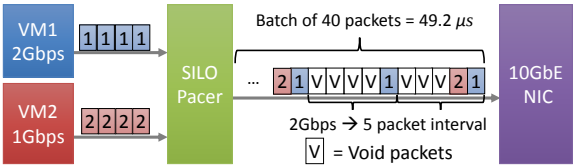


Figure 7: Use of void packets achieves packet level pacing in the face of NIC burstiness

switch port connected to the server to determine the number of VMs that can be placed at the server. The $CalcValidAllocations$ function implements this logic. This, in turn, relies on the $GetMaxDelay$ function that uses the traffic arrival and service curves to determine the maximum queuing delay at a specific network port if the requested VMs are placed a certain way. If all requested VMs can be accommodated at servers within the rack, the request is accepted (lines 18–20). 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 requirements regarding other resources and 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 heuristics for multi-dimensional bin packing to place VMs [45]. Our queuing constraints could be added to the set of constraints used by such systems, though we defer an exploration to future work.

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 6 shows the hierarchy of token buckets used by the pacer to enforce traffic conformance. 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. They ensure that traffic between them achieves the max-min fair bandwidth it would have achieved across the virtual topology shown in Figure 3. The pacer at the destination divides its guaranteed bandwidth amongst all VMs sending traffic to it and communicates a rate back to the sender. The pacer at the source uses this to determine the rate at which it can send traffic to any given VM such that the aggregate traffic being sent by the VM does not exceed its guarantee; i.e., it determines the rate B_i for the top token buckets in Figure 6 such that $\sum B_i \leq B$. The rate B_i is also communicated to the destination hypervisor.

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.

4.3.1 Packet level pacing

Ideally, the token buckets should be serviced at a per-packet granularity. However, this precludes the use of I/O batching techniques which results in high CPU overhead and reduces throughput (as shown in Figure 2). One solution is to implement the pacing at the NIC itself [22,46]. However, this requires hardware support. For ease of deployment, we prefer a software solution.

To ensure acceptable CPU overhead for network I/O, we must leverage I/O batching. In our prototype, we batch 50μs of data in transferring packets from the hypervisor to the NIC. However, most NICs are bursty and will forward the entire batch of packets back-to-back [22]. While our placement analysis can model such behavior in the traffic arrival curve, it can generate better placements when the arrival curve is less bursty and packets conform to their

guarantees even at small time scales; for example, for a VM with a 1Gbps bandwidth guarantee, full-sized packets (1.5KB) should only be released every $12\mu s$.

In order to retain the high throughput and low overhead offered by I/O batching while still pacing packets at sub-microsecond timescales, we use a novel technique called "**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. Figure 7 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. 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 $68ns$.

4.3.2 Tenants without guarantees

Silo leverages two level priority forwarding in switches to support tenants without any network guarantees. "Best effort traffic" from such tenants is marked as low priority while traffic from tenants with guarantees is high priority. As long as switches ensure high priority traffic gets precedence in both forwarding and buffer occupancy, our guarantees hold. Tenants without guarantees share the residual network capacity. The provider can ensure reasonable performance for such tenants by limiting the fraction of network bandwidth reserved for tenants with guarantees.

While Silo's bandwidth guarantees mean that high network utilization is not a primary design goal, such best effort traffic can improve utilization. Further, many cloud applications are indeed not network-limited, so they do not need any network guarantees.

5 Implementation

Our software pacer is implemented as a Windows NDIS filter driver [47]. The pacer driver sits between the vswitch and the NIC driver, so we do not require any modification to the NIC driver, applications or the guest OS. At a high-level, the pacer has the following workflow. When applications send packets, they are passed to our pacer by the vswitch. The pacer looks up the source and destination addresses for the packets, and classifies them into appropriate token buckets. We use virtual token buckets, i.e. packets are not drained at the exact moment, rather we timestamp when each packet needs to be sent out. The pacer regularly pulls out packets from the token bucket and sends them to the NIC driver.

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. First is to keep the gap between packets within a batch. We achieve this using void packets (§4.3.1). Second is to schedule the next batch of packets before NIC starts to idle. This is non-trivial, especially since we want to keep the batch size small so that packet queuing delay is limited. We borrow the idea of soft-timers [48] and reuse existing interrupts as a timer source. Our pacer does not use any separate timer, but triggers sending the next batch of packets upon receiving a DMA (Direct Memory Access) completion interrupt for transmit packets.

We also implemented the placement algorithm described in §4.2. To evaluate its performance, we measured the time to place tenant requests in a datacenter with 100K machines. Over 100K representative requests, the maximum time to place VMs is less than a second.

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). We verify that our prototype can offer bandwidth, delay and burstiness guarantees that, in turn, ensures predictable message latency. Through microbenchmarks, we show that Silo's pacer is able to saturate 10Gbps links with low CPU overhead.

(ii). Through NS-2 simulations, we show that Silo improves message latency as compared to state-of-the-art solutions like DCTCP [21], HULL [22], and Oktopus [16]. Unlike Silo, none of these solutions can ensure predictable message completion time.

(iii). We characterize the performance of our VM placement algorithm and show that, as compared to locality-aware VM placement, it can actually improve cloud utilization and thus, accept more tenant requests.

Evaluation set-up. We model two classes of tenants.

	Class I	Class II
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 1: Tenant classes and their average network requirements.

Class-I contains delay-sensitive tenants that run a small message application, and require bandwidth, delay and burst guarantees. Each class-I 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 [21] and distributed storage [49]. *Class-II* contains bandwidth-sensitive tenants that run a large message application and only require bandwidth guarantees. Such tenants have an all-to-all communication pattern, as is common for data parallel applications. Unless otherwise specified, we generate the bandwidth and burst requirements from an exponential distribution with the parameters in table 6. For simulations, we use a multi-rooted tree topology for the cloud network. The capacity of each network link is 10Gbps, the network has an oversubscription of 1:5. We model commonly used shallow buffered switches [50] with 312KB buffering per port (queue capacity is 250μs). Each physical server has 8 VM slots.

6.1 Pacer microbenchmarks

We first evaluate our pacer implementation in terms of throughput it can sustain and the CPU overhead. We use physical servers equipped with one Intel X520 10GbE NIC, and two Intel Xeon E5-2665 CPU (8 cores, 2.4Ghz). There are two configuration that affects the pacer performance (i) packet batch size and (ii) amount of outstanding packets. After testing with various loads, we find that a batch size of 50 μs, and 150 μs worth of outstanding packets are enough to saturate the link capacity. This means that bandwidth-compliant packets cannot be queued at the end host for more than 150μs.

Figure 8 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. The red line represents the number of transmitted packets per second, including void packets. We observe that to generate only void packets at 10 Gbps, the pacer consumes about 0.6 cores. 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 1/10th 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. In Figure 9, we show the throughput for both void packets and data packets. Except for 9 Gbps, we saturate 100% of the link capacity, and achieve actual data rate at more than 98% of the ideal rate.

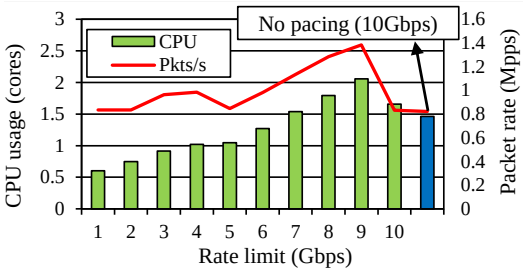


Figure 8: CPU usage of Silo pacer.

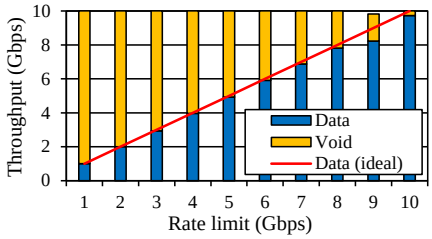


Figure 9: Throughput achieved by Silo pacer.

6.2 Testbed experiments

We begin with a simple testbed experiment to understand the benefits of Silo. We use six physical servers connected to a single switch. The switch has 9 MB of buffering. There are two tenants, A and B, running a delay-sensitive and a bandwidth-sensitive application respectively. Each tenant has 40 workers across 5 servers (8 on each). On the last server, each tenant runs an aggregator that requests messages from the workers. The tenant A aggregator requests a 12 KB response from all 40 workers simultaneously at 20 ms interval. The tenant B aggregator requests for 1 GB messages. For Silo, ten-

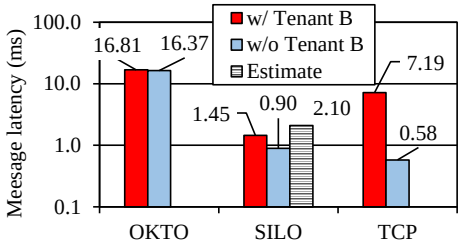


Figure 10: Message latency for delay-sensitive application (with competing bandwidth-sensitive application).

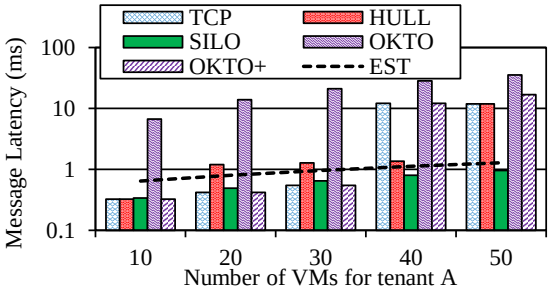


Figure 11: 95th percentile 20KB message latency for tenant A with varying number of its VMs (i.e. flows).

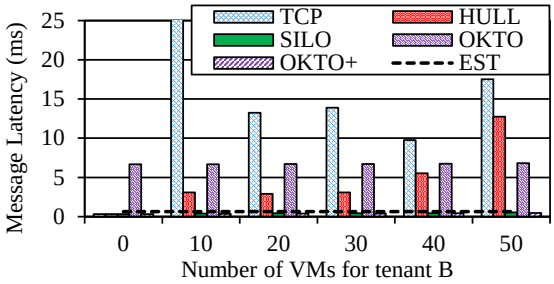


Figure 12: 95th percentile 20KB message latency for tenant A with varying tenant B workload.

ant A is guaranteed $\{B$=0.25Gbps, S=12 KB, d=1000$\mu s\}$ with $B_{max} = 1Gbps$, and tenant B is guaranteed $\{B$ = 9.75Gbps, S=1.5 KB, d=1000$\mu s\}$.

Figure 10 shows the message latency at 99th percentile for tenant A with and without tenant B. We compare against TCP and Oktopus. Oktopus uses our pacer but without bursts. For Silo, the tenant's guarantees can be used to determine an estimate for maximum message latency. The figure shows that the 99^{th} percentile message latency with Silo is within the estimate, even when there is tenant B traffic. Further, the aggregate throughput for tenant B is 99.6% of its bandwidth guarantee. With Oktopus, tenant A gets a bandwidth guarantee but is not allowed to burst, so the message latency is higher. The messages take 16ms; this confirms that our pacer is fine-grained in that it spreads out a 9 packet message (12 KB) over 16 ms. TCP, however suffers from large queueing at the switch because of tenant B traffic. The latency is as high as 7 ms, which is the amount of buffer in the switch. Overall, this shows that the Silo prototype can ensure predictable end-to-end message latency.

6.3 Packet level simulations

We use NS-2 simulations to compare Silo against existing solutions, and show that it delivers predictable message latency. We begin with a couple of simple scenarios. The simulation involves two tenants. Tenant A is a class-I tenant running a small message application with an all-to-one communication pattern such that all VMs simultaneously send a 20KB message to the same receiver. The messages are generated such that the receiving VM's average bandwidth requirement is 200 Mbps. Thus, the tenant is guaranteed: $\{B$=200Mbps, S=20KB, d=500$\mu s\}$. Tenant B is a class-II tenant with all-to-all communication. The tenant only needs bandwidth guarantees with no delay and burst requirements, so its guarantees are: $\{B$=1250Mbps, S=1.5KB, d=N/A$\}$ with B_{max}=B.

We start with a simple setup involving only tenant A. Figure 11 shows the 95^{th} percentile message latency with varying number of tenant A VMs. The "EST" lines show the maximum latency estimate given the tenant's guarantees. We find that Silo ensures messages complete before the estimate, even in the worst case (not shown in figure). With Oktopus [16], tenant A is guaranteed bandwidth but cannot burst traffic, so the latency is much higher. We also extend Oktopus so that VMs can burst at line rate (labelled OKTO+). However, with a lot of VMs, the OKTO+ latency is also high and variable. Silo avoids this by controlling the VM burst rate. HULL also results in high and variable message latency; the workload results in severe incast and even HULL's phantom queues cannot avoid buffer overflows. DCTCP performs slightly worse than HULL, so we omit it in the graph.

Next we introduce tenant B. Tenant A has 10 VMs, and we vary the number of VMs for tenant B, thus increas-

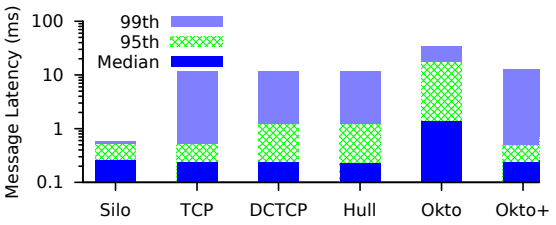


Figure 13: Message latency for class-I tenants.

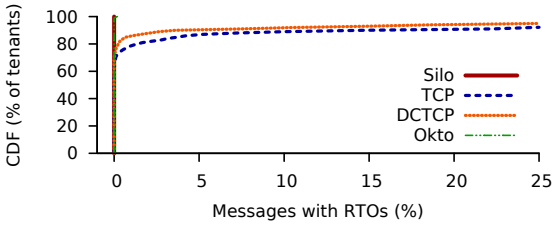


Figure 14: Class-I tenants whose messages incur RTOs.

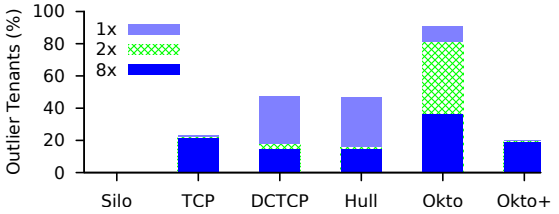


Figure 15: Outlier tenants, i.e. tenants whose 99^{th} percentile message latency exceeds the latency estimate.

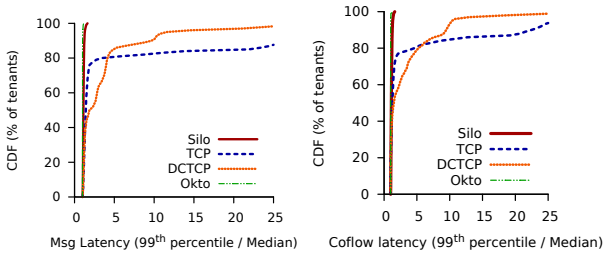
ing the amount of network interference from a competing application. Figure 12 shows that Silo, unlike other approaches, still ensures that message latency for tenant A is predictable. HULL does not isolate competing workloads, leading to high message latency for tenant A when there is a lot of competing traffic. Thus, low queuing, by itself, does not ensure predictable message latency in multi-tenant settings.

Non uniform workload: With Silo's guarantees, a tenant can independently determine the worst-case latency for its messages, irrespective of other tenant workloads. The use of controlled bursts ensures the latency is low too. However, real applications often involve a single VM sending many messages close together, in which case the burst allowance does not benefit the latter messages. This can be addressed by over provisioning the tenant's bandwidth and burst guarantees. For example, we repeated the experiment above such that tenant A's messages, instead of arriving uniformly, had exponential inter-arrival times with the same average rate as before. By guaranteeing the tenant 40% more average bandwidth (280 Mbps) and doubling its burst size (40KB), we ensure that 99% of messages can use the burst allowance. This preliminary result suggests that Silo's guarantees, beyond predictability, can also ensure very low message latency for realistic workloads. We are investigating this further with application traffic traces.

Simulations with many tenants. We now present results from simulations involving both class-I and class-II tenants across 10 racks, each with 40 servers and 8 VMs per server, resulting in 3200 VMs. The number of tenants is such that 90% of VM slots are occupied. For each run, Silo places VMs using its placement algorithm. For Oktopus, we use the bandwidth-aware algorithm in [16]. For other solutions, we use a locality-aware algorithm that places VMs as close to each other as possible. Figure 13 shows the latency for all small messages across 50 runs. Silo ensures low message latency even at the 99^{th} percentile while all other approaches have high tail latency. With Oktopus, VMs cannot burst, so the message latency is high, both at the average and at the tail. At 99^{th} percentile, message latency is 60x higher with Oktopus compared to Silo. Okto+ is Oktopus's bandwidth guarantee with the burst allowance. It reduces the average latency but still suffers at the tail due to switch buffer overflow. With DCTCP and HULL, message latency is higher by 22x at the 99^{th} percentile (and 2.5x at the 95^{th}).

Two factors lead to poor tail latency for TCP, DCTCP and HULL. First, class-I 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-I small messages compete with large messages from class-II tenants. This leads to high latency and even losses for small messages. Figure 14 shows that with TCP, for 21% of class-I tenants, more than 1% of messages suffer retransmission timeout events (RTOs). With DCTCP and HULL, this happens for 14% of tenants. Thus, *by itself, neither low queuing (ensured by DCTP and HULL) nor guaranteed bandwidth (ensured by Oktopus) is sufficient to ensure predictable message latency*.

We also look at *outlier tenants*, i.e. class-I tenants whose 99^{th} percentile message latency is more than the latency estimate. Figure 15 shows that Silo results in no outliers. Since we observe outliers with other approaches,



(a) Messages (99^{th}/median) (b) Coflows (99^{th}/median)

Figure 16: Per-tenant message and coflow latency for class-I tenants.

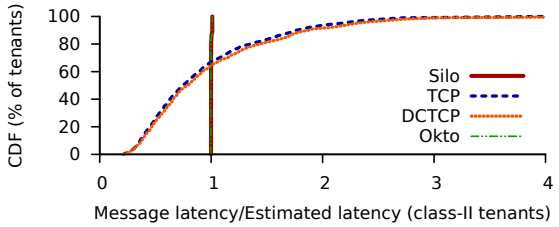


Figure 17: Message latency for class-II tenants.

we mark the fraction of outliers whose latency exceeds the estimate by 1x, 2x or 8x. With DCTCP and HULL, 15% tenants are 8x outliers. For clarity, we omit the HULL (results similar to DCTCP) and Okto+ lines in subsequent graphs.

The same trend is evident in absolute message latency for each class-I tenant. Figure 16(a) shows that with Silo and Oktopus, the 99^{th} percentile latency is very close the median. This is because both offer bandwidth guarantees. However, with Oktopus, the median latency itself is much higher. With DCTCP, the 99^{th} percentile latency is more than twice the median for more than 50% of the tenants. Figure 16(b) shows this variability is higher when we measure coflow completion time [51], i.e. the time it takes for all messages of a given tenant to finish. The performance of OLDI-like applications is dictated by the completion time of coflows and such variability can be detrimental to user-perceived performance.

However, since Silo does not let tenants exceed their bandwidth guarantee, it can hurt network utilization. In our experiments, the average network utilization with Silo was 35% lower relative to TCP and DCTCP. This is partly because Silo results in better VM placements which reduces network traffic. However, this can impact the performance of class-II tenants with large messages whose completion is dictated by the bandwidth achieved. Figure 17 shows the average message latency for class-II tenants, normalized to the message latency estimate. With both Silo and Oktopus, tenant bandwidth is guaranteed, so all messages finish on time. With TCP and DCTCP, the message latency varies. 65% of tenants achieve higher bandwidth with DCTCP as compared to Silo but there is a long tail with many tenants getting very poor network bandwidth with DCTCP. In the next section, we show that such outlier tenants can actually drag down total cloud throughput. Overall, this shows how Silo, without best effort traffic, trades-off network utilization for predictability.

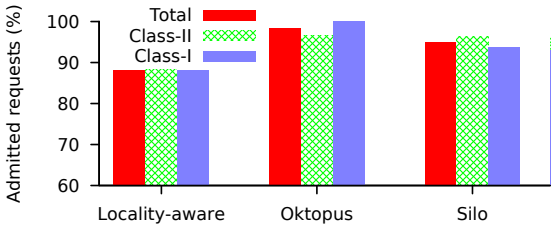


Figure 18: Admitted requests with various placement approaches.

6.4 VM placement simulations

To evaluate Silo's VM placement algorithm, we developed a flow-level simulator that models a public cloud datacenter. The datacenter has 32K servers with a three tier network topology. The arrival of tenant requests is a Poisson process with the average arrival rate such that the datacenter is 90% full. We compare Silo's placement against two approaches: *Oktopus* placement that guarantees VM bandwidth only and a *locality-aware* placement that greedily places VMs of a tenant close to each other. With the latter approach, bandwidth is fairly shared between flows, thus modeling ideal TCP behavior.

Figure 18 shows the fraction of tenants admitted. Silo admits 3% fewer requests than Oktopus. While Silo admits almost the same number of class-II requests, it admits 6% fewer class-I requests because it accounts for their delay and burstiness guarantees. However, both Silo and Oktopus can accept more requests than locality-aware placement. Silo can accept 7% more requests than this greedy approach.

This result is counter-intuitive. Locality-aware place-

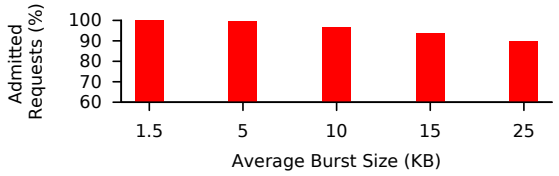


Figure 19: Admitted requests with varying burst size.

ment will only reject requests if there are insufficient VM slots. By contrast, Silo can underutilize the network and can reject a request, even when there are sufficient 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 tenants do not suffer from poor network performance.

Figure 19 shows that the percentage of requests accepted by Silo reduces as we increase the average burst size requested by tenants. With an average burst size of 25KB, which is larger than messages for typical OLDI applications like web search [21,24], Silo accepts 93.3% of requests. We also repeated the experiments while varying other simulation parameters. We omit the results for brevity but highlight the main trends. We find that Silo can accept more requests than locality aware placement when the datacenter occupancy is 70% or more. Cloud operators like Amazon EC2 target an average occupancy of 70-80% [52]. Silo's ability to accept tenants increases with larger switch buffers and lesser network oversubscription.

7 Related Work

Silo adds to a rich literature on network performance guarantees and optimization. We briefly summarize the work most relevant to Silo.

Many recent efforts look at cloud network sharing. They propose different sharing policies, including fixed guarantees [15,16], time-varying guarantees [17], minimum bandwidth guarantees [14,18,19], per-source fairness [53] and per-tenant fairness [54]. However, these proposals focus solely on bandwidth allocation, thus catering to bandwidth-sensitive applications like data analytics. They do not cater to applications that need bounded packet delay and ability to burst.

Many solutions achieve low latency in private datacenters by ensuring small network queues [21–23] or by accounting for flow deadlines [24–26]. Silo targets predictable message latency in multi-tenant public clouds 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 do the control the total burstiness on network links. The former hurts the latency of both small and large messages while the latter hurts small messages. In §6.3, we show that DCTCP and HULL can suffer from high and variable message latency, especially when there is competing traffic from other tenants. 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 pass application hints like deadlines. Thus, a hypervisor-only solution is needed.

Finally, apart from HULL [22], none of the delay-centric (and even the bandwidth-centric) solutions look at the host part of the end-to-end network path. Overall, while these solutions work well for the class of applications or environment they were designed for, they do not ensure predictable end-to-end message latency for diverse cloud applications.

The early 1990's saw substantial work on providing network performance guarantees to Internet clients. Stop and Go [55] is the proposal most similar to Silo in terms of architecture, in that it paces packets to minimize queueing delay and to ensure conformity to bandwidth provisioning. However, it introduces a new queueing discipline at switches while Silo does not require any switch changes. Further, Silo leverages the flexibility of placing VMs, an opportunity unique to datacenters. More generally, much of the work from this era focuses on novel queueing disciplines at switches – an approach we chose to avoid in favor of better deployability. Zhang and Keshav [56] provide a comparison of work on queueing disciplines from this era, including Delay-Earliest-Due-Date [57], Jitter Earliest-Due-Date [58], and Hierarchical

Round Robin [59].

Stochastic network calculus [60] allows a distribution describing queue length to be calculated. Traditionally, these bounds are computed on an arrival distribution over flows. However, in datacenters, where flows arrive in job-generated bursts, these stochastic techniques would have to be extended to account for the bursty arrival of correlated flows. Because of these correlated arrivals, we chose to focus on worst-case bounds.

ATM [61] and its predecessors [62] provided varying types of performance guarantees. Silo works with commodity Ethernet. However, it is not as flexible as ATM; it provides only three classes of network guarantees (bandwidth, bandwidth + delay + burst, and best-effort). Nevertheless, to the best of our knowledge, Silo is the first to take delay bounds and use them to work backwards for VM placement. Furthermore, Silo pushes to much higher performance in terms of generating paced traffic: by using void packets, Silo can achieve sub-microsecond granularity pacing with very low CPU overhead.

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.

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] Distributed query execution engine using apache hdfs. `https://github.com/cloudera/impala`.

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

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

[8] 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 *VLDB*, 2008.

[9] A. Li, X. Yang, S. Kandula, and M. Zhang. CloudCmp: comparing public cloud providers. In *Proc. of conference on Internet measurement (IMC)*, 2010.

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

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

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

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

[14] P. Soares, J. Santos, N. Tolia, and D. Guedes. Gatekeeper: Distributed Rate Control for Virtualized Datacenters. Technical Report HP-2010-151, HP Labs, 2010.

[15] 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*, 2010.

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

[17] 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.

[18] 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.

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

[20] 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.

[21] 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.

[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] 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.

[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] 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.

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

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

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

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

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

[33] G. Ananthanarayanan, S. Kandula, A. G. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in map-reduce clusters using mantri. In *Proc. USENIX OSDI*, 2010.

[34] High Performance Computing (HPC) on AWS. `http://aws.amazon.com/en/hpc-applications/`.

[35] N. Cardwell, S. Savage, and T. Anderson. Modeling TCP latency. In *Proc. IEEE INFOCOM*, 2000.

[36] G. Wang and T. S. E. Ng. The Impact of Virtualization on Network Performance of Amazon EC2 Data Center. In *IEEE Infocom*, 2010.

[37] Measuring EC2 performance. `http://tech.mangot.com/roller/dave/entry/ec2_variability_the_numbers_revealed`.

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

[39] 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*.

[40] 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. Proc. SIGCOMM '99.

[41] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sen-

gupta. VL2: a scalable and flexible data center network. In *Proc. of ACM SIGCOMM*, 2009.

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

[43] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *Proc. of Usenix NSDI*, 2010.

[44] 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.

[45] 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.

[46] S. Radhakrishnan, V. Jeyakumar, A. Kabbani, G. Porter, and A. Vahdat. Nicpic: Scalce and accurate end-host rate limiting. In *Proc. USENIX HotCloud'13*.

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

[48] 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.

[49] A. Phanishayee, E. Krevat, V. Vasudevan, D. G. Andersen, G. R. Ganger, G. A. Gibson, and S. Seshan. Measurement and analysis of tcp throughput collapse in cluster-based storage systems. In *FAST '08*.

[50] 10GE ToR port buffers. `http://www.gossamer-threads.com/lists/nanog/users/149189`.

[51] M. Chowdhury and I. Stoica. Coflow: A Networking Abstraction for Cluster Applications. In *Proc. of ACM HotNets*, 2012.

[52] Amazon's EC2 Generating 220M. `http://cloudscaling.com/blog/cloud-computing/amazons-ec2-generating-220m-annually`.

[53] A. Shieh, S. Kandula, A. Greenberg, and C. Kim. Sharing the Datacenter Network. In *Proc. of Usenix NSDI*, 2011.

[54] 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.

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

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

[57] 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.

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

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

[60] Y. Jian and Y. Liu. *Stochastic Network Calculus*. Springer-Verlag, 2008.

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

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