PicNIC: Predictable Virtualized NIC

Praveen Kumar[†] Nandita Dukkipati[*] Nathan Lewis[*] Yi Cui[*] Yaogong Wang[*]
Chonggang Li[*] Valas Valancius[*] Jake Adriaens[*] Steve Gribble[*] Nate Foster[†] Amin Vahdat[*]
[†]*Cornell University* [*]*Google*

Abstract

Network virtualization stacks are the linchpins of public clouds. A key goal is to provide performance isolation so that workloads on one Virtual Machine (VM) do not adversely impact the network experience of another VM. Using data from a major public cloud provider, we systematically characterize how performance isolation can break in current virtualization stacks and find a fundamental tradeoff between isolation and resource multiplexing for efficiency. In order to provide predictable performance, we propose a new system called PicNIC that shares resources efficiently in the common case while rapidly reacting to ensure isolation. PicNIC builds on three constructs to quickly detect isolation breakdown and to enforce it when necessary: CPU-fair weighted fair queues at receivers, receiver-driven congestion control for backpressure, and sender-side admission control with shaping. Based on an extensive evaluation, we show that this combination ensures isolation for VMs at sub-millisecond timescales with negligible overhead.

CCS Concepts

• **Networks** → *Transport protocols*; *Network algorithms*; Network reliability; Cloud computing.

Keywords

Congestion Control, Performance Isolation

ACM Reference Format:
Praveen Kumar, Nandita Dukkipati, Nathan Lewis, Yi Cui, Yaogong Wang, Chonggang Li, Valas Valancius, Jake Adriaens, Steve Gribble, Nate Foster, and Amin Vahdat. 2019. PicNIC: Predictable Virtualized NIC. In *SIGCOMM '19: 2019 Conference of the ACM Special Interest Group on Data Communication, August 19–23, 2019, Beijing, China.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3341302.3342093

1 Introduction

There is a fundamental tension between efficiency and predictable performance in any shared computing platform. On the one hand, providers want to utilize resources efficiently by oversubscribing the infrastructure to achieve economies of scale. On the other hand, the tenants using these platforms want predictable performance without worrying about interference from other tenants. While this issue has been a long-standing problem across a variety of platforms, it is substantially exacerbated in today's public clouds. Whereas in private clouds, resources can be provisioned in a collaborative fashion between providers and tenants to balance the tradeoff between predictability and efficiency, in public clouds the provider typically lacks visibility into tenant applications, so they must compensate with costly overprovisioning to tilt the default in favor of performance isolation [33, 49].

Most prior work in the networking context has focused on sharing bandwidth in the network fabric (Table 1). However, in this work, we find that contention for resources in the host virtualization stack is often a key contributor to unpredictable performance, and that existing mechanisms are insufficient for guaranteeing isolation at end hosts. For instance, if one VM receives small packets at a high rate—e.g., due to a denial of service (DoS) attack—other VMs on the same host may observe spikes in latency or even packet loss due to isolation breakage at the host virtualization stack. Generally speaking, there are two resources on the hosts that must be shared among VMs: (i) the bandwidth (BPS) and (ii) the rate (PPS) at which packets can be transmitted and received. Unfortunately, current stacks do not provide adequate mechanisms for sharing these resources across multiple senders and receivers, which means that the behavior of one VM can adversely impact others.

This paper argues that cloud providers should offer tenants the abstraction of a *Predictable Virtualized NIC*—i.e., performance guarantees formulated as per-VM SLOs in terms of measurable bounds on bandwidth, latency percentiles, and loss rates. This notion of predictable performance generalizes stricter notions of isolation that are impossible to achieve without excessive overprovisioning [49]. In particular, although predictable performance does not mandate strict non-interference at the network level, tenants can still use bounds on performance metrics to reason effectively about the service they will receive.

Of course, achieving predictable performance without sacrificing efficiency is inherently difficult. The fundamental challenge stems from the fact that on end hosts the resources required to process packets depend on the overall traffic mix, which is hard to predict in advance. Overprovisioning for the worst case sacrifices efficiency gains due to multiplexing, while underprovisioning risks violating SLOs. The approach we take is to *initially provision resources for efficient sharing, under the optimistic assumption that VMs will be well-behaved, but monitor the system and rapidly adapt when conditions change, falling back to strict isolation as the safe default.* While this approach lacks some attractive properties—e.g., it is not always work-conserving—and requires a distributed implementation, it does provide predictable performance and uses resources efficiently in the common case.

We realize these ideas in PicNIC, a system that provides the *Predictable Virtualized NIC* abstraction in a shared public cloud environment. With PicNIC, each VM is guaranteed a minimum

SIGCOMM '19, August 19–23, 2019, Beijing, China

ACM ISBN 978-1-4503-5956-6/19/08…$15.00
https://doi.org/10.1145/3341302.3342093

System	Abstraction	Shared capacity	Bandwidth isolation model	Host stack isolation	Predictable latency
SeaWall [69], NetShare [39], FairCloud (PS-L/N) [62]	Virtualized Fabric	Constant	Fair sharing	No	None
Oktopus [7], Hadrian [8], SecondNet [28], Proteus [76], Pulsar [5], CloudMirror [41]	Virtualized Fabric	Constant	Hose-based [18] (VC, TIVC, TAG, pipe)	No	None
Gatekeeper [66], FairCloud (PS-P) [62], EyeQ [35], ElasticSwitch [63], HUG [14]	Virtualized Fabric	Constant	Hose	No	None
Silo [34]	Virtualized Fabric	Constant	Hose	No	Fabric
PicNIC	Virtualized NIC	Variable	Hose	Yes	vNIC

Table 1: Related systems focus on sharing constant fabric bandwidth and cannot provide isolation at the end-hosts. In contrast, PicNIC provides isolation at the end-hosts by sharing the virtualization stack's variable packet-processing capacity and delivers predictable performance in terms of bandwidth, latency distribution and loss rate for each VM.

and maximum bandwidth envelope, bounded latency distribution, and near-zero packet loss within the stack. To achieve these goals, PicNIC leverages two key insights as design principles: (i) *SLO-based resource sharing*: for predictability, packet-processing resources utilized for each VM should be proportional to performance SLOs. In particular, sufficient resources within the virtualization stack (e.g., CPU cycles, NIC and PCIe bandwidth) should be allocated to ensure the minimum guarantees of the SLO (e.g., bandwidth) independent of the behavior of other VMs. (ii) *Backpressure and early drops*: for efficiency, if a packet needs to be dropped (or queued), it should be done as early in the processing pipeline as possible by applying apt backpressure—e.g., packets likely to be dropped at the receiver due to insufficient resources should not be admitted at the source.

Note that existing approaches are not strong enough to ensure these guarantees for three reasons. First, prior work (Table 1) focuses on apportioning bandwidth in the fabric, whereas PicNIC also offers non-trivial guarantees on latency and packet loss. Second, effectively managing resources in virtualization stacks requires a different approach due to *variable* BPS and PPS capacities for packet processing at end hosts. Third, whereas prior work often relies on rate limiting at sources, this itself can cause a breakdown of isolation at end hosts. Avoiding such breakdowns requires rethinking traffic shaping and extending backpressure mechanisms across a complete chain—the fabric, within the host stack, and between the host stack and the VM.

Contributions. Our main finding is that it is possible to ensure predictable performance by quickly navigating the spectrum between being resource-efficient (work-conserving) and providing strict isolation. Our approach quickly detects risks of SLO violations and tilts the tradeoff towards isolation. To this end, the paper makes the following contributions.

1. Analyzing data from a production environment, we systematically identify key bottlenecks in network virtualization stacks that lead to isolation breakages. We provide insights into new challenges, such as variable packet processing capacity of the stack and limitations of traffic shaping, that make it challenging to ensure predictable performance (§2).
2. We propose an intuitive abstraction for a predictable virtualized NIC which gives a well-defined quantifiable meaning to predictable performance (§4). While it enables high efficiency in the common case, it also prioritizes isolation when there is a risk of SLO violation.
3. To realize this abstraction in a system, we identify key design principles based on SLO-based resource sharing, admission control and backpressure (§3) and present the design and implementation of such a system, PicNIC, using a combination of local and end-to-end constructs (§5).
4. Our evaluation on a large-scale cloud deployment shows that PicNIC can ensure predictable performance without sacrificing efficiency (§6).

Lessons from production. Using data from a major cloud provider, we found that isolation can break down at both sender and receiver host stacks. We found it more efficient and practical to enable parts of PicNIC to react to isolation breakdowns based on signals rather than designing a system that attempts to enforce isolation invariants. Implementing these reactions at receivers turns out to be particularly difficult as it requires coordination with multiple senders. Moreover, such coordination must be done at short timescales, which poses additional practical challenges. While traditional congestion control works by sending acknowledgements, maintaining a similar level of state and generating packets to carry congestion signals would impose significant overheads in a virtualization stack. Ultimately, selecting a design to reduce reaction time to $O(ms)$ without regressing the data path required several iterations. In the same vein, using traffic shaping as a building block required several other features to be implemented in the data path including buffering, backpressure to VMs, and out-of-order completions to avoid head-of-line blocking.

Ethical concerns. This work does not raise any ethical issues.

2 Cause and Cost of Unpredictability

Network virtualization provides the abstraction of a private network to cloud tenants while sharing the underlying physical network [37, 60]. Fig. 1 shows a simplified view of the on-host component of a typical network virtualization stack with egress ② and ingress ⑤ *engines* that process (e.g., encapsulate/decapuslate, apply firewall rules, etc.), buffer, and transport packets between the VMs and the NIC within a host. These components can be realized in various ways on different implementations—e.g., Andromeda uses a modular software switch with fast shared engines and hardware offload [16] while Azure's Virtual Filtering Platform

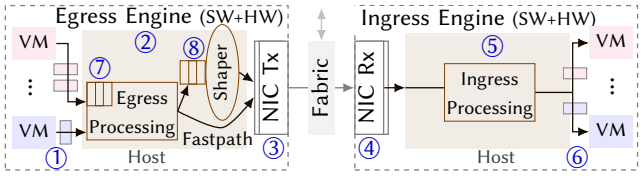


Figure 1: Overview of an on-host network virtualization stack.

uses Generic Flow Tables implemented in hardware and FPGAs to accelerate other network functions [21, 22]. Changing the division of labor between software and hardware can reduce but does not fully eliminate isolation issues—see §8.

Based on incidents observed in production, we find that the severity and frequency of isolation breakages in current virtualization stacks are significant enough to motivate a systematic characterization. Accordingly, in this work, we focus on issues at the end-host and assume that the network fabric is not a bottleneck. Our model for a packet's path is: VM Tx ① → egress engine ② → NIC egress ③ → NIC ingress ④ → ingress engine ⑤ → VM Rx ⑥. At the egress, packets may be buffered for shaping ⑧ based on policies (e.g., WAN bandwidth allocation [38]) or to ensure that transmit completions (§B) are returned to a VM in-order ⑦.

A key factor which makes predictable performance difficult to achieve is that *per-packet processing costs are not constant*, whether in software or hardware. For instance, cache misses affect the number of CPU cycles needed to process a packet in software, while the PCIe and DRAM bandwidth required per packet varies in hardware [45, 53]. These costs also depend on the complexity of operations that must be performed on each packet [55, 58] as well as the overall traffic mix. Thus, the packet processing capacity of a virtualization stack is variable, unlike a link's capacity in the fabric, and makes it difficult to directly extend prior work on fabric bandwidth isolation (Table 1) to this case. While our observations apply to a broad range of implementations (§8), for concreteness, in the rest of this paper we consider a kernel-bypass software virtualization stack with hardware offloads [16].

In general, isolation can break due to contention for any shared resource: egress engine processing capacity ②, buffering in egress engine ⑦ & ⑧, egress NIC buffers ③, ingress NIC buffers ④, and ingress engine processing capacity ⑤. Through careful provisioning, it is possible to largely avoid issues at egress engine ② and NIC buffers ③. For instance, we can throttle the egress engine to ensure that egress NIC buffers do not overflow. However, there remain three key points where isolation can break: egress buffering ⑦ & ⑧, ingress NIC ④ and ingress engine ⑤. We walk through each of these along with examples of isolation breakages from production.

2.1 Egress Buffer Contention

At the egress, contention for shared buffers can break isolation between VMs. Fig. 2 shows a scenario recreated from a production incident, with three VMs on separate hosts. VM1 → VM2 is an unthrottled (or "fastpath") TCP flow. While fastpath packets are expected to pass through the stack without delay, this expectation may be violated in the presence of throttled flows, which must be shaped. For example, at $t = 12s$, VM1 starts a 30s long UDP flow to VM3, which needs to be rate limited to 10 Mbps per policy; the queueing delay for UDP packets is shown as their sojourn

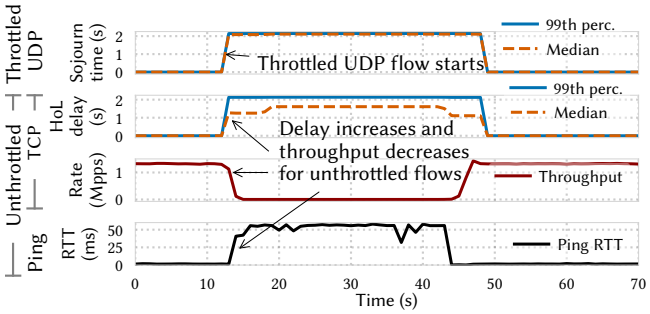
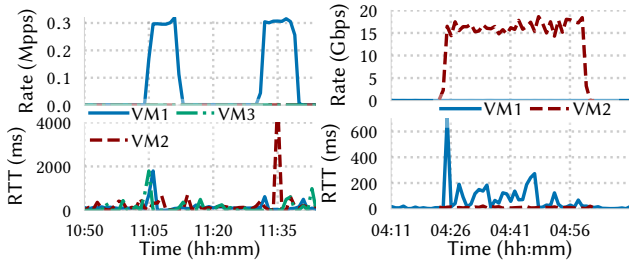


Figure 2: HoL blocking and isolation breakage at egress.



(a) Packet rate overload **(b)** Bandwidth overload

Figure 3: Isolation breakage at ingress due to overloads.

time in the shaper queues ⑧ (top plot). Due to interference in in-order buffers ⑦, the TCP flow experiences Head-of-Line (HoL) blocking (second plot), resulting in decreased throughput (third plot)—packets are held in the in-order buffer waiting for transmit completions to finish in order for all flows, including rate limited ones. RTTs for unthrottled ping from VM1 also increase during this period (bottom plot).

The root cause of this phenomenon is that the egress buffer fills up, either because the shaper queues ⑧ (which hold packet descriptors for throttled flows) or the in-order buffers ⑦ (which hold packet descriptors for both fastpath and throttled flows) are full. In each case, faster flows can be HoL blocked because slower flows monopolize the buffers. Note that increasing buffer size would only *delay* isolation breakage, but would not prevent it. Interestingly, even a single VM with different "flow types" can face this problem (though it can be exacerbated by multiple VMs)—e.g., consider two flows from the same VM, with the faster flow HoL-blocked due to the slower one. Using separate buffers to isolate traffic would require a buffer for each traffic class (rate limit) which is prohibitively expensive. To mitigate such contentions, we need controlled sharing of buffers and admission of packets into the stack.

2.2 Ingress NIC Contention

At the ingress, contention for NIC buffers ④ due to an excessively high packet rate (PPS) can also break isolation. Fig. 3a shows an incident from production where packet bursts to VM1 lasted for a few minutes. While VM1 receives a goodput of 300 kpps (top) at a low bandwidth (2 Mbps) due to small packets, 148 kpps drops occur at the NIC. The RTTs to a few unrelated VMs on the same host increase by more than 100× due to interference (bottom), even though the burst is directed at VM1.

The root cause here is that a storm of small packets overwhelms the packet-processing capacity at the ingress engine ⑤, where

processing costs are *per-packet* rather than per-byte [29, 45]. As there is a limit on the CPU cores available for packet processing, the PPS capacity is bounded (the same is true for HW-based stacks [31, 53]). In particular, the NIC buffers fill up when the engine ⑤ cannot process packets as fast as they arrive—e.g., when a VM is under a DoS attack—saturating the ingress engine with a deluge of small packets that it is unable to drain quickly from NIC queues ④. The overflowing NIC queues in turn leave little room for other traffic, resulting in tail-drops at the ingress NIC and breaking isolation between VMs.

2.3 Ingress Engine Contention

Finally, contention for resources in the rest of the ingress engine due to a large spike of traffic (BPS) can also break isolation. Fig. 3b shows such an incident where VM2 receives a burst of ~16 Gbps from a production service. On the same host, VM1 is receiving at a low BPS of 20 Mbps. In addition to 186 kpps drops in the ingress engine, RTTs for VM1's traffic increase even though its workload has not changed during the period.

Even if the engine drains the NIC queues ④ quickly, packets may encounter bottlenecks in transit to VM queues ⑥. Such bottlenecks may arise for various reasons including cache misses, excessive flow table lookups, and even the inability of VM queues to absorb the spike. These bottlenecks can cause packets to be buffered and eventually dropped after wasting CPU cycles—this harms isolation and is also unfair. So, in addition to arbitrating engine resources, we need admission control to mitigate contention.

2.4 Extent of Isolation Breakages

In summary, even if we assume no contention within the network fabric, there are several ways in which a VM can adversely impact the network experience of other VMs. These are not one-off incidents; we found that their frequency is correlated with packet drop rates at the ingress and HoL blocking latency at the egress. We counted the number of 1-second intervals at each host when the ingress drop rate exceeds 10 kpps in NIC Rx and found the cumulative count over a fleet of servers to be tens of thousands per day. As each such incident on a host can potentially impact the isolation experience of tenants, the problem can become severe since these incidents are not uniformly distributed over the fleet. Even if we can provision resources to make isolation breakages rare in the common case, the lack of isolation mechanisms creates a soft target for disruptive traffic such as during DoS attacks; we want to eliminate this target.

3 Design Principles

Based on our analysis (§2), we come away with two guiding principles for providing predictable performance to VMs:

$\mathcal{P}1$. *SLO-based resource sharing*: for predictability, packet-processing resources utilized for each VM should be proportional to performance SLOs. In particular, sufficient resources within the virtualization stack (e.g., CPU cycles, NIC and PCIe bandwidth, shared buffers at shaper and NIC ingress) should be allocated to ensure the minimum guarantees of the SLO (e.g., bandwidth) independent of the behavior of other VMs. More generally, we prioritize SLO-compliance over work-conserving behavior.

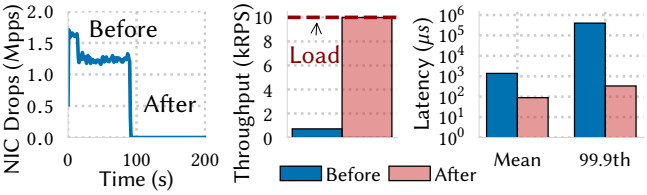


Figure 4: Example: PicNIC ensures predictable performance.

$\mathcal{P}2$. *Backpressure and early drops*: for efficiency, if a packet needs to be dropped or queued, it should be done as early in the processing pipeline as possible by applying apt backpressure—e.g., packets likely to be dropped at the receiver due to insufficient resources should not be admitted at the source.

By using SLO-based resource allocation per VM ($\mathcal{P}1$), an overloading VM quickly builds up its own queue or has its own packets dropped before affecting others; this facilitates early detection of SLO violations. While $\mathcal{P}1$ is necessary, it is *not sufficient* to ensure predictable performance SLOs (§4) as it still allows excessive delay and losses—e.g., in an incast scenario, packets destined for a VM may be admitted to the source egress engines in excess of the receiver VM's SLO-based ingress capacity. These excess packets consume resources at the egress, in the fabric, and at the ingress, only to be dropped eventually. Such wasted work elevates the latency and packet drops for colocated VMs as seen in §2.2 and also hurts efficiency. Thus, admission control with backpressure all the way to the source and early drops ($\mathcal{P}2$) are necessary to ensure isolation and efficiency [17, 50, 66]. We find that a combination of these principles is *sufficient* to ensure predictable SLOs with high efficiency.

We show how PicNIC (§5), a system based on these principles, ensures predictable performance in a simple three-host setup. We have three client-server pairs of VMs: one pair runs a latency-sensitive request-response workload, memcached [46], and the other two run a resource-intensive UDP job. We place all server VMs on the same host. The memcached client, colocated with another UDP client, generates a load of 10k requests/sec (kRPS) to the server, while the UDP clients generate 100B packets at a high PPS. Fig. 4 shows that without PicNIC, there are $1.2{\times}10^6$ drops/sec at the receiver host NIC (left), resulting in elevated latency (right) and low throughput in terms of completed RPS for memcached (middle). PicNIC increases throughput by an order of magnitude to 100% and lowers the tail latency by three orders of magnitude (close to the memcached job existing by itself) while ensuring *predictably high goodput* even for the intensive UDP flows (details in §6).

4 Predictable Virtualized NICs

The designer of any network virtualization stack must consider the performance and isolation guarantees they wish to provide [4, 16, 22]—these properties cannot be "bolted on" as an afterthought. Ideally, every VM should have the illusion of having its own dedicated NIC with known performance characteristics, regardless of the unpredictable behavior of other VMs. Our approach is to provide an abstraction of a *predictable virtualized NIC*, defined by target SLOs, to each VM. A physical NIC is characterized by certain properties such as bandwidth, delay and loss rate [31]. Drawing an analogy, we define a predictable vNIC along the same three SLO

Metric	Predictable vNIC SLO
Bandwidth	*min.* and *max.* envelope (hose model)
Delay	Low; predictable distribution*
Loss rate	No drops for cooperating traffic*

*for well-behaved traffic in bandwidth envelope

Table 2: Abstraction: Predictable vNIC SLO metrics

dimensions of offered bandwidth, delay and loss rate. This provides a quantifiable meaning to predictable performance and extends the purely functional behavior of virtual NICs with SLOs. Table 2 summarizes our abstraction. While we do not include other possible dimensions—e.g., message and packet rates—for intuitiveness, one may extend this abstraction as needed.

By providing SLOs that apply to traffic at sources as well as destinations, this abstraction provides the illusion of a dedicated NIC to each VM. However, to fully understand the guarantees ensured by this abstraction, it is important to consider the overall mix of traffic and patterns of communication between VMs. We aim to offer predictability for traffic mixes with a combination of shaped and unshaped flows, as well as under ingress fan-in communication patterns. Hence, some of the techniques proposed in PicNIC, e.g. for sender coordination, would also apply to standalone NICs and not just to virtualized NICs.

Bandwidth. We define bandwidth (BPS) using the *hose model* [18], a natural fit for a vNIC. The abstraction is in terms of an envelope with a minimum guarantee (`MIN_BPS`) for *performance* and a maximum cap (`MAX_BPS`) for *predictability*. `MIN_BPS` is based on provisioning and is not oversubscribed—i.e., $\sum$ `MIN_BPS` for VMs on a host $\leq$ host NIC line rate. `MAX_BPS` is a cap that is not exceeded even if the VM is the sole occupant of the host. `MAX_BPS` can be oversubscribed for higher multiplexing and efficiency. If multiple VMs contend for bandwidth, then after allocating each VM its `MIN_BPS`, any residual bandwidth is shared per policy. The hose model along with BPS envelope allows expressing a range of policies. Non-work-conserving policies, such as `MAX_BPS` = `MIN_BPS`, can provide strict latency and loss properties, while work-conserving policies, such as `MAX_BPS` per VM = NIC line rate, can provide high utilization.

BPS SLOs are defined for traffic adhering to "standard" packet size distributions, such as Internet Mix (IMIX) [10, 51, 75], as is common for NICs, switches, and middleboxes [11, 68, 73]. Such devices usually have a PPS limit and support line rate only if the average packet size is above a certain limit—e.g., Intel XL710 40GbE NIC has a PPS limit of ~37 Mpps, and needs packets larger than 160B to achieve line rate [31].

Delay. vNIC delay for fastpath flows (§2) is the time elapsed from the instant a packet exits the sender's guest OS Tx queue, e.g. `virtio` Tx [36], to being received at the Rx queue of receiver's guest OS, excluding the delay in fabric. For throttled flows, it also excludes any delay necessary for shaping. vNIC delay has two parts: (i) egress delay from guest OS Tx queue to the wire and (ii) ingress delay from the wire to guest OS Rx queue. The ingress delay is further split into two parts: (i) delay in NIC hardware queues and (ii) delay in engine. We specify predictable vNIC delay as a distribution rather than a fixed value—e.g., median delay $\leq 10\mu s$ and 99^{th} perc. delay $\leq 50\mu s$. We ensure delay guarantees for traffic that adheres to its BPS envelope.

Loss rate. Like delay, a predictable vNIC should have low loss rate for traffic within its BPS envelope. Particularly, both ingress and egress drops should be zero for traffic responding to backpressure. Uncooperative traffic that keeps discharging packets at a high rate may experience high drops in order to ensure isolation for others. Drops at the ingress hurt efficiency, and tail-drops at the NIC are unfair; a predictable vNIC must avoid such drops at ingress.

To summarize, we define the abstraction of a predictable vNIC in terms of bandwidth, delay and loss-rate SLOs for well-behaved traffic that adheres to a standard packet size distribution. Delay and loss-rate SLOs are for cooperating traffic within its BPS envelope. Even if a VM does not cooperate, we ensure predictable performance for other well-behaved VMs. While we want to utilize resources at shared contention points efficiently (e.g., in a work-conserving manner) in the common case, we rapidly switch to strict isolation when the goal of efficiency is too far counter to isolation—e.g., when per-VM target SLOs may be violated. We build PicNIC with three mechanisms to quickly detect such cases and tilt the tradeoff towards isolation in sub-*ms* response time and with negligible overhead to the dataplane.

5 Design and Implementation

In this section, we introduce the constructs of PicNIC and discuss their combined role in achieving the predictable vNIC abstraction (§5.1), followed by the details of design and implementation of each construct (§5.2 to §5.4).

5.1 Constructs and Guarantees

Abstractly, instantiating the design principles (§3) in either hardware or software requires a combination of local and end-to-end constructs for: (i) sharing resources at ingress based on SLOs ($\mathcal{P}1$), (ii) admission control to manage contention at ingress ($\mathcal{P}2$), and (iii) sharing egress resources per SLOs with apt backpressure to guest ($\mathcal{P}1, \mathcal{P}2$). Concretely, for a software-based virtualization stack, we identify the following three design constructs. Fig. 5 shows their system-level view.

Ingress CPU-fair Weighted Fair Queues (§5.2). At the ingress, PicNIC implements per-VM CPU-fair weighted fair queues (CWFQs) to share the engine's processing capacity ($\mathcal{P}1$) by apportioning the engine's CPU cycles in proportion to VM weights. A VM i's weight (w_i) is based on its SLO—e.g., $w_i \propto$ `MAX_BPS`$_i$.

Receiver-driven congestion control (§5.3). PicNIC provides SLO-based shares of NIC bandwidth and engine capacity to VMs following the hose model. To meet delay and loss-rate SLOs at ingress, it applies backpressure to sources ($\mathcal{P}2$) by implementing a receiver-driven hypervisor-level congestion control called PicNIC Congestion Control (PCC). PCC computes ingress rate limits per VM and shares these limits among senders.

Sender-side admission control (§5.4). At the egress, PicNIC implements admission control using a traffic shaper based on Carousel [67] to enforce the rate limits computed by PCC. Additionally, it creates backpressure to guest VM transport ($\mathcal{P}2$) and enforces per-VM packet limits in the shaper buffer for isolation ($\mathcal{P}1$).

	Constructs	CPU-fair Weighted Fair Queues (CWFQs, at ingress)	PicNIC Congestion Control (PCC, end-to-end)	PicNIC Sender-side Admission Control (at egress)
Guarantees	Bandwidth		Compute SLO-based NIC BW share per receiver; backpressure to egress host	Enforce VM-VM BPS limits and per-VM MAX_BPS; backpressure to guest OS stack
	Delay	Share engine CPU based on SLO; provide signals to PCC	Compute PPS rate limit for each receiver; backpressure to egress host	Enforce VM-VM PPS limits; backpressure to guest OS stack; egress isolation
	Loss rate	Avoid unfair drops at NIC	Rate limit senders to avoid drops	Backpressure to guest OS stack

Table 3: PicNIC constructs. Role of each PicNIC construct in achieving the predictable virtualized NIC abstraction.

Achieving a predictable vNIC abstraction. We provide insights into our choice of these constructs and outline how they work together to provide bandwidth, delay and loss-rate SLOs. Table 3 summarizes the role of each construct.

Bandwidth envelope. PCC computes the SLO-based ingress BPS limit per receiver VM by first allocating the minimum bandwidth SLO (MIN_BPS) for each VM and then dividing the residual host NIC bandwidth based on the maximum bandwidth SLOs (MAX_BPS). Per the hose model, PCC rate limits traffic at senders to meet the SLO-based ingress BPS limit for each receiver VM. It also imposes a total egress rate limit of MAX_BPS for each VM.

Low vNIC delay. At the ingress, isolation breaks when a high-rate flow overloads the engine (§2.2). CWFQs share engine capacity among VMs per SLO and drop offending traffic fairly to protect others. This still wastes resources to pull and classify extra packets from the NIC before dropping. NIC Rx queueing also increases in such cases. To avoid such ingress overloads, PCC computes and applies a SLO-based PPS rate limit for responsible traffic. At the egress, PicNIC applies backpressure to guest OS stack when shaping flows. To avoid HoL blocking due to slow flows exhausting buffer resources (§2.1), PicNIC's out-of-order completions (§5.4.3), along with Linux NAPI-TX [44], offers per-TCP-flow level backpressure. Per-VM packet accounting (§5.4.2) limits the number of outstanding packets per VM in the engine, thus protecting other VMs regardless of the number of flows for each VM.

Low loss rate. At the ingress, CWFQs move packets from NIC queues to per-VM queues with high priority to avoid any unfair NIC drops. To avoid both NIC and CWFQ drops at ingress, PCC enforces rate limits at the egress. Sender-side admission control applies backpressure to guest OS stack to prevent drops at egress.

5.2 Ingress CPU-Fair WFQs

The goal of CWFQs is to reduce unfair NIC drops and share ingress engine's CPU based on SLOs to provide isolation at short timescales. They also aid in early detection of overloads.

The capacity of the packet-processing engine on the end host is generally not constant and depends on various factors including packet sizes, complexity of network functions, cache misses etc. During overloads, the engine may not be able to process packets as fast as they arrive at the NIC, leading to queueing and eventually tail-drops in the NIC Rx queues—both are unfair as a high-rate flow can impact other flows.

Sharing engine capacity is challenging using conventional approaches such as allocating SLO-weighted ingress BPS per VM. BPS does not reflect the true resource usage—e.g., a flow with 64B packets at a modest 512 Mbps translates to 1 Mpps, which may consume significantly more CPU cycles relative to another flow with 1500B packets at a higher BPS of 2.4 Gbps but lower PPS of 200 kpps. Engine CPU usage relates more directly to PPS instead of BPS. However, even using PPS to track resource usage is tricky as some flows need more complex processing, e.g. encryption, and hence more CPU cycles per packet compared to others. Thus, we must account CPU cycles used by the engine for each VM.

To track engine CPU usage per VM, PicNIC classifies packets early by pulling them from NIC Rx, classifying them by destination VM, and pushing them to the corresponding per-VM CWFQs. All resource-intensive processing happens after CWFQs. By draining NIC queues with high priority, PicNIC mitigates unfair tail-drops and delays in the NIC. From per-VM queues, PicNIC dequeues packets for processing while sharing engine's CPU cycles fairly. For this, PicNIC records the CPU time spent to process each VM's packets and maintains the moving average (EWMA) giving more weight to recent CPU usage. Using EWMA CPU time per-VM, PicNIC assigns each VM a dynamic priority that governs how frequently the VM's queue is scheduled. PicNIC recomputes and decreases a VM's priority whenever its queue is serviced, while skipping any empty queues when scheduling. Hence, PicNIC's work-conserving scheduler ensures that each VM is allocated engine CPU in a weighted fair manner based on its demand and weight ($\propto$ SLO).

5.3 PicNIC Congestion Control

PCC plays a central role in achieving all three SLOs by applying apt backpressure across the network and coordinating among multiple senders and receivers (Table 3). Note that while CWFQs are needed only when multiple VMs exist per host, PCC is required even when VMs exist in isolation on each host.

PCC implements receiver-driven hypervisor-level admission control. It has two parts: (i) PCCB, which sets BPS limits to ensure bandwidth envelopes, and (ii) PCCP, which sets PPS limits to ensure low vNIC delay and loss rate; only one of these limits is dominant at a given time for each flow [14, 25]. With just BPS limits, we can provide bandwidth SLOs and yet not meet the delay and loss-rate SLOs. Similarly, with just PPS limits, it is difficult to ensure bandwidth envelope as packet sizes vary. Hence, we require both.

5.3.1 Bandwidth envelope.

At the ingress, PCCB apportions the host NIC's BPS capacity (C) among VMs per their (MIN_BPS, MAX_BPS) envelopes. To avoid drops, sources should not send, in aggregate, more than the receiver VM's apportioned ingress BPS. PCCB ensures that each VM gets an ingress BPS $\in$ [MIN_BPS, MAX_BPS]. Note that this allocation may not always be work-conserving because of the MAX_BPS limit per VM.

PCCB monitors the rate, r_j^{in}, at which traffic for a VM j is received at the host. To compute the fair ingress capacity c_j for each

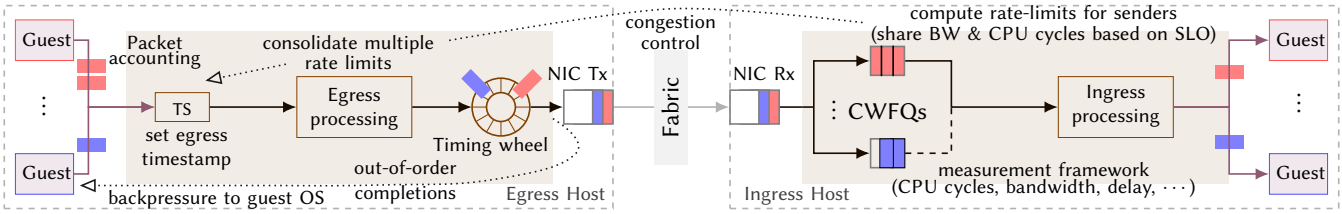


Figure 5: PicNIC architecture. Local constructs (ingress CWFQs and egress sender-side admission control) working in coherence with end-to-end receiver-driven congestion control to achieve the predictable vNIC abstraction.

VM j, it first allocates the MIN_BPS: $\forall j, c_j \leftarrow \texttt{MIN\_BPS}_j$. Then, it allocates any spare bandwidth ($C - \sum c_j$) to each active VM (i.e., with $r_j^{in} > 0$) in proportion to the VM's MAX_BPS and subject to $c_j \leq \texttt{MAX\_BPS}_j$. To ensure that the ingress rate (r_j^{in}) for each VM converges to its fair capacity (c_j), provided there is enough traffic, PCCB computes a BPS value r_j^{lim} to rate limit senders. We experimented with various control algorithms, each showing different convergence properties, such as time to converge to the correct rate and stability. For concreteness and ease of comparison with prior work, we present an implementation based on RCP and EyeQ [19, 35]:

$$\text{at every epoch,} \quad r_j^{lim} \leftarrow r_j^{lim} \cdot \left(1 - \alpha \cdot \frac{r_j^{in} - c_j}{c_j}\right)$$

PCCB computes r_j^{lim} at fixed epochs of length ϵ and notifies r_j^{lim} to sender hosts. At the egress, PicNIC rate limits $VM_i \rightarrow VM_j$ traffic to $w_i \cdot r_j^{lim}$ so that c_j is shared among senders in proportion to their SLO (e.g. $w \propto$ MAX_BPS). This approach is independent of the number of senders so that r_j^{in} converges to c_j while r_j^{lim} scales automatically to a stable fixed point. In addition, PicNIC also enforces a total egress rate limit of MAX_BPS for each sender VM.

5.3.2 Ensuring low delay in vNIC. While PCCB ensures BPS SLO for VMs, delay and loss-rate SLOs do not follow automatically—e.g., delay and drops can be high if a colocated VM receiving at a high PPS, but within its BPS limits, overflows ingress queues (§2.2). PCCP handles such cases.

PCCP's goal is to keep the vNIC ingress delay within a given distribution. Its activation is gated by per-VM CWFQ occupancy, and it remains *off* for VMs whose traffic do not contribute to overload. Thus, in the absence of engine overload, PCCP is off. When PCCP is activated for a VM due to high CWFQ occupancy or drops, it starts with a rate estimate. The VM's maximum loss-free receive rate [50] or CWFQ goodput acts as a good estimate because it is close to the desired ingress PPS rate for the VM.

PCCP is delay-based. The ingress delay ($delay_{in}$) consists of delays in the NIC and CWFQs. After CWFQs, packets are processed and delivered to VMs without further queueing. As PicNIC ensures that each VM gets an SLO-based share of engine CPU for packet processing, an overloading VM with a resource-hungry flow gets lower scheduling priority for its CWFQ, making its packets wait longer in CWFQs compared to others. So, CWFQ delay automatically captures the engine's packet processing cost for each VM.

$$delay_{in} = T_{\text{dequeue from per-VM queue}} - T_{\text{received at host NIC HW}}$$

PCCP uses $delay_{in}$, measured using accurate NIC hardware timestamps, as congestion signal to compute the VM's ingress PPS capacity. We use PPS here as delay depends on engine CPU usage, which relates more directly to PPS than BPS. Bounding $delay_{in}$ for an overloading VM also decreases the shared NIC queuing part; this, in turn, ensures low $delay_{in}$ for well-behaved VMs. Alg. 1 describes PCCP's rate control algorithm, which is based on these observations and inspired by prior approaches [1, 48, 78]. In a nutshell, it keeps $delay_{in}$ close to a specified *threshold*. Multiplicative decrease reduces rate based on the extent by which delay exceeds *threshold*, while fast recovery attempts to quickly increase the rate to the value before the last decrease. Additive increase probes for higher rates when delay is within the threshold, and after a few cycles, it enters hyper-active increase to accelerate. Like PCCB, this also runs at fixed epochs per VM.

Algorithm 1: PCCP RATE CONTROL

Input: $delay\_in$
$delay \leftarrow EWMA(delay, delay\_in)$
if $delay > threshold$ **then** ▷ multiplicative decrease (MD)
  $rate \leftarrow (1 - \beta \cdot (1 - \frac{threshold}{delay})) \cdot rate$
  $counter \leftarrow 0$ ▷ enter fast recovery (FR)
  $target\_rate \leftarrow rate$
else ▷ default: fast recovery (FR)
  if $N_{AI} < counter \leq N_{HAI}$ **then** ▷ additive increase (AI)
    $target\_rate \leftarrow target\_rate + \delta$
  if $counter > N_{HAI}$ **then** ▷ hyper-active increase (HAI)
    $target\_rate \leftarrow target\_rate + (counter - N_{HAI}) \cdot \delta$
  $rate \leftarrow \frac{rate + target\_rate}{2}$
  $counter \leftarrow counter + 1$
Output: $rate$ ▷ initial value = goodput / approx. #senders

PicNIC tracks the approximate count of sources with lightweight cardinality estimation methods—e.g., HyperLogLog [23]—and uses it to set a per-sender limit instantly on activation (Alg. 1). This ensures low vNIC delay even while PCCP's rate limit converges.

Feedback mechanism. For both PCCB and PCCP, we explored two choices: (i) the in-datapath approach either generates special packets in the datapath or uses encap headers of packets in the reverse direction to piggyback rate limits and (ii) the control-thread approach uses a control thread to scrape statistics and sampled sources cached in the datapath, compute rate limits and notify source hosts via a control channel, e.g. RPC. While the in-datapath approach is more responsive, it also incurs overheads in the critical datapath.

5.4 Sender-side Admission Control

To enforce PCC's rates limits, PicNIC implements a traffic shaper building on Carousel [67] at the egress (§5.4.1). PicNIC avoids egress isolation breakages (as in §2.1) by adding two techniques: (i) packet

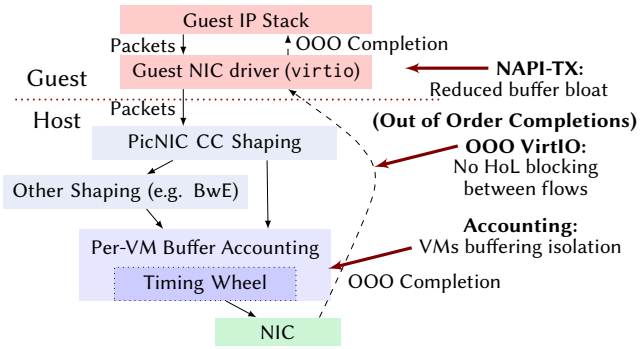


Figure 6: PicNIC's sender-side admission control.

accounting (§5.4.2) and (ii) backpressure to guest OS stack (§5.4.3). Fig. 6 illustrates PicNIC's sender-side admission control.

5.4.1 Enforcing Rate Limits. PicNIC implements shaping with a single time-indexed queue—Timing Wheel (TW) [74]. It supports a flow being shaped by multiple policies [14, 25]. In our implementation, the rate limits applicable to a packet are consolidated into a single timestamp based on the slowest rate. This timestamp serves as the earliest departure time for when the packet is released from the TW for transmission. VM-to-VM BPS and PPS rate limits computed by PCC are also consolidated into these timestamps.

5.4.2 Packet Accounting. To avoid costly dynamic memory allocation, the engine uses a statically-allocated shared packet descriptor pool. We buffer packets for throttled flows in the TW and release the corresponding descriptor when a packet leaves the TW and enters the NIC. A non-cooperating VM can exhaust the entire descriptor pool by discharging packets at a high rate for a throttled flow, leading to drops for other VMs (§2.1). To handle this, PicNIC introduces *Packet Accounting* to track the number of packets buffered in the TW from each VM. It sets a per-VM cap and a total shared cap on the number of buffered packets, and does not admit packets that exceed either limit. Thus, even if a VM sends excessive packets for throttled flows, it cannot exhaust the descriptor pool, and hence would not impact other VMs' flows.

5.4.3 Backpressure to Guest OS. Egress shaping must be accompanied by complete backpressure to the guest networking stack (§2.1 and §3). Every link in this chain of backpressure is critical for isolation. PicNIC implements this by combining out-of-order completions [67] for backpressure from the engine to the guest NIC driver, and NAPI-TX [44] along with TCP Small Queues (TSQ) [15] for backpressure from the guest NIC driver to the guest IP stack.

The first link in this chain is from the egress engine to the guest NIC driver. To create backpressure here, the engine must hold the completion event for each packet until the packet actually leaves the engine. On dequeuing a packet from guest Tx queues, instead of marking it as Tx complete immediately, PicNIC sends the Tx completion to the guest only when the packet is delivered to the host NIC (or dropped). By limiting the number of descriptors per VM, we prevent a VM from sending a deluge of packets into the engine. Implementing such deferred completions needs support from the guest driver—e.g., `virtio` [36] works in two completion modes, in-order and out-of-order (OOO). In the in-order mode, completion events must be received in the same order as the transmission sequence—e.g, if a VM has two flows and only one is throttled, the unthrottled flow's descriptors cannot be freed until the throttled flow's descriptors are freed; this causes HoL blocking. Eventually, both flows become throttled. To solve this, PicNIC enables OOO completions in `virtio` so that the flows' descriptors can be freed independently. This relies on packet accounting to limit the number of buffered packets per flow and ensuring that this limit is lower than the total number of descriptors in the guest NIC.

The second link in this chain is from the guest NIC driver to the guest stack. We combine NAPI-TX with TSQ to ensure that each flow can queue only a limited amount of data for Tx in the guest stack. NAPI-TX is a Linux kernel feature that makes `virtio` call the SKB destructor after a packet is actually "out"—i.e., at Tx completion interrupt, instead of immediately on enqueue to `virtio`. This provides socket backpressure and is needed for TSQ. Enabling NAPI-TX in guests is critical for complete backpressure all the way up to the guest applications as depicted in Fig. 6. This prevents bufferbloat and avoids associated long latencies and isolation breakage among flows for a VM as shown in §6 (Table 4).

5.5 Practical Considerations

We need to overcome multiple challenges to make PicNIC practical. The key challenges stem from our goal of sub-*ms* isolation enforcement without sacrificing datapath performance.

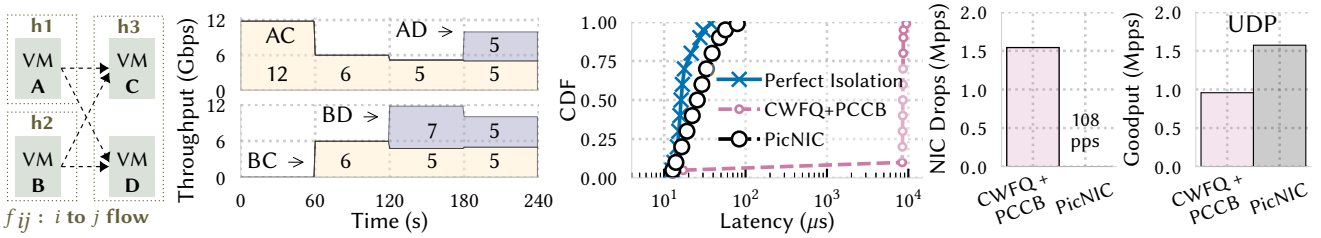
Responsiveness. We need early signals of overloads. We engineer CWFQs to provide basic isolation at short timescales and aid PCCP with signals for rapid overload detection. Notifying rate limits to sources quickly is challenging as maintaining a list of all senders is expensive; PicNIC uses lightweight sampling to identify heavy-hitters. To make PCCP more responsive, we start it with an initial rate estimate to apply immediately on detecting isolation issues.

Performance. To achieve line rate, we have $O(100ns)$ to process each packet; so every per-packet operation needs to be optimized to minimize overhead. We address this by running PCC at fixed epochs instead of per-packet while deriving congestion signals from existing metrics. We gate PCCP on CWFQ occupancy; this turns off PCCP to reduce overheads when there are no isolation issues. To keep the datapath minimal and fast, we explored a control-thread based approach that moves PCC out of the datapath (§5.3).

Sometimes, the source of offending traffic could be the Internet, and not other VMs in the same datacenter—e.g., in the case of DoS attacks. To handle such cases, PicNIC considers the load-balancers [20, 56] as traffic sources and sends rate feedback to load-balancers, which impose throttles on DoS traffic. As there can be multiple load-balancers, we also explored a centralized approach for rate dissemination where a central server collects all feedbacks and distributes them based on source weights. While a distributed approach may take longer to notify all sources, a centralized approach can disseminate rate limits quickly.

6 Evaluation

We evaluate PicNIC in production of a major cloud provider. We start with microbenchmarks (§6.1) to show that PicNIC implements the predictable vNIC abstraction with low overheads (§6.2). Then, we quantify the benefits to applications (§6.3) and end-users (§6.4).



(a) Setup **(b)** VM-VM throughput within envelope **(c)** Low latency **(d)** Near-zero drops **(e)** Better goodput

Figure 7: Microbenchmarks: PicNIC provides a predictable vNIC abstraction to VMs. In addition to providing predictable bandwidth envelope, low latency and near-zero drops, PicNIC also improves the goodput for the PPS-intensive traffic.

6.1 Microbenchmarks

First, we show that PicNIC realizes the predictable vNIC abstraction with a small setup (§6.1.1), and then we evaluate each construct of PicNIC in more detail (§6.1.2).

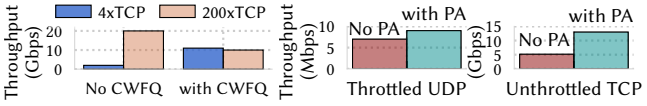
6.1.1 Predictable vNIC. To show that PicNIC ensures a bandwidth envelope, low delay and loss rate for VMs (Table 2), we use the setup shown in Fig. 7a.

Bandwidth. The key result we show is that even while bottlenecks shift, the BPS for every VM is maintained within the envelope. To construct shifting bottlenecks, we stagger the start of VM-VM flows f_{AC}, f_{BC}, f_{BD} and f_{AD}. Each host NIC's capacity is 20 Gbps, and each VM's BPS envelope is set to [4 Gbps (MIN_BPS), 12 Gbps (MAX_BPS)]. Fig. 7b shows the overall throughput for each flow. Initially, f_{AC} achieves 12 Gbps as PicNIC enforces MAX_BPS egress rate for A. When f_{BC} starts, PCCB detects the contention for C's ingress bandwidth (12 Gbps) and rate limits each flow to 6 Gbps. When f_{BD} starts, PCCB detects that D is active, and computes the share of h_3's NIC bandwidth (20 Gbps) as 10 Gbps each for C and D. This causes the rate limits for f_{AC} and f_{BC} to decrease to 5 Gbps each. This, in turn, leaves spare egress bandwidth for B, which f_{BD} grabs to achieve 12 - 5 = 7 Gbps. Finally, when f_{AD} arrives, PCCB allocates D's ingress capacity (10 Gbps) equally among f_{AD} and f_{BD}.

Low delay. We show that (i) PicNIC, with PCCP, ensures low latency and drops even in extreme cases while (ii) just BPS guarantees are insufficient to achieve these. We extend the setup in Fig. 7a with two more VMs: E on h_2 and F on h_3. f_{AC} and f_{BD} are high-PPS UDP flows with 256B packets. We measure RTT for f_{EF} which is an open-loop latency prober generating packets following a Poisson process (rate λ = 1 kpps) and discards any drops from measurements.

Fig. 7c shows f_{EF} RTTs measured with PicNIC and compares it to the cases with (i) just BPS envelope (i.e., with CWFQ+PCCB but no PCCP), which can be thought of as similar to EyeQ [35], and (ii) perfect isolation, i.e., when the latency prober is run by itself without any other traffic. As the flows are not BPS-intensive, there is no contention for bandwidth. However, the high-PPS UDP traffic causes isolation breakage at the hosts and impacts the latency for f_{EF}. PicNIC detects this isolation issue and throttles UDP to the appropriate rate so that f_{EF} achieves latency close to the case with perfect isolation.

Low loss rate. As $h3$'s ingress engine is overloaded with high-PPS UDP traffic, packets are dropped at the host NIC. Even with CWFQ+PCCB, we find such tail-drop rate to be 1.54 Mpps as shown in Fig. 7d. With PicNIC, such drops decrease to a mere 108 *pps* as PicNIC admits only enough packets as can be processed without breaking isolation.



(a) CWFQs at ingress **(b)** Packet accounting at egress

Figure 8: Ingress and egress isolation with PicNIC constructs.

Setup	Throughput (Mbps)		Timing wheel occupancy	Drops (kpps)
	Unthrottled	Throttled		
No NAPI + EC	8,993	9	4000 (100%)	300
NAPI + OOO	10,145	9	50 (1.25%)	0

Table 4: OOO completions and NAPI-TX ensure egress isolation.

Improved efficiency. Somewhat counterintuitively, even though PicNIC throttles the UDP flows, the total goodput for UDP increases from 0.96 Mpps to 1.57 Mpps as shown in Fig. 7e. By applying back-pressure to excess traffic, PicNIC avoids wasted work of pulling packets from the NIC and classifying them into per-VM queues before it can decide to drop packets exceeding their engine CPU share. Thus, performing admission control as opposed to dropping packets leads to increased engine efficiency and 1.6× higher goodput.

6.1.2 Components. Now, we evaluate the constructs in more detail and tease out their role in achieving predictable performance.

CPU-fair Weighted Fair Queues. Consider the setup from Fig. 7a with two VM-VM flows: f_{AC} with 4 parallel TCP streams and f_{BD} with 200 parallel TCP streams. Without CWFQs, f_{BD} gets an unfairly high share of CPU cycles at h_3, while f_{AC} suffers as shown in Fig. 8a. On enabling CWFQs, the ingress engine's capacity is shared equally among the two flows, and hence both are able to achieve equal (fair) throughput.

Packet Accounting (PA). We reproduce egress isolation breakage using the setup from Fig. 7a by moving VM B to h_1 so that A and B are colocated. f_{AC} is unthrottled TCP, while f_{BD} is a UDP flow throttled to 10 Mbps. Without accounting, B keeps sending packets for f_{BD}, most of which are dropped at the traffic shaper. This exhausts the descriptor pool in the engine and impacts the unthrottled flow, f_{AC}. Packet accounting limits the number of outstanding packets from B in the engine and thus provides isolation for A, as shown in Fig. 8b.

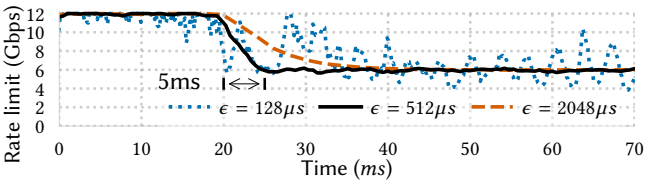


Figure 9: PCCB converges within $5ms$.

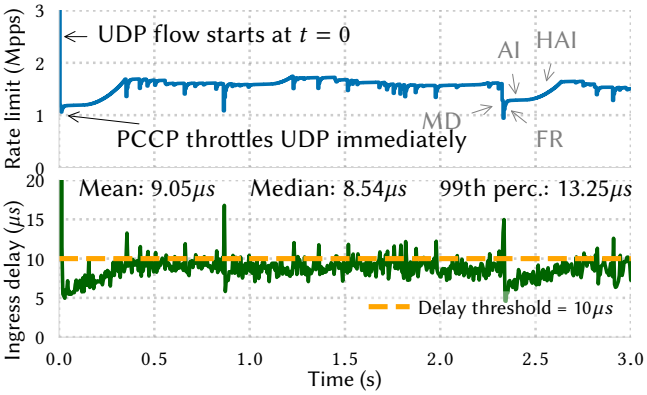


Figure 10: Predictable low latency with PicNIC.

OOO Completions and NAPI-TX. Next, we see how HoL blocking can arise between flows. Consider two flows: i) f_{AC}—UDP throttled at 10 Mbps, and ii) f_{AD}—unthrottled TCP. With in-order early completions (EC), descriptors are freed as soon as they leave the guest. Thus, both flows are able to transmit packets into the egress stack at the same rate. This overwhelms the TW as it has to drop packets beyond the horizon for throttled flows. PicNIC implements out-of-order completions (OOO) plus NAPI-TX with `virtio` for the guest. As shown in Table 4, this improves the throughput for the unthrottled flow while keeping the TW occupancy low and eliminates egress drops in the engine.

Responsiveness of PCC. For each receiver VM, PCCB computes rate limits at epochs of fixed duration (ϵ). We quantify the responsiveness of PCCB with varying ϵ using the setup from Fig. 7a and a BPS envelope of 4 to 12 Gbps per VM. At $t = 0$, only f_{AC} is active and gets the entire 12 Gbps. At $t = 20ms$, f_{BC} starts and now, PCCB needs to limit each flow to 6 Gbps to meet `MAX_BPS` for C. Fig. 9 shows that using a small $\epsilon = 128\mu s$ leads to oscillations for f_{AC}'s limit, while a large value of $2ms$ leads to slow convergence. An epoch of a few RTTs leads to fast convergence—e.g., with $\epsilon = 512\mu s$, the rate converges within $5ms$. We find similar convergence for PCCP but leave a formal analysis to future work.

Predictable latency. We evaluate the efficacy of PCCP in ensuring low vNIC delay using the same setup. f_{BD} is a low-BPS latency-sensitive flow. At $t = 0$, f_{AC} starts a high-PPS UDP traffic with 100-byte packets to create extreme overload at h_3. We use a conservative value of $\epsilon = 5ms$, and set the vNIC ingress delay threshold to $10\mu s$. Even in this extreme case, PicNIC is able to deliver predictable latency for f_{BD} as shown in Fig. 10. First, even with a large ϵ, as soon as PicNIC detects an isolation issue, it turns on PCCP with a rate estimate based on goodput. This brings the vNIC delay close to the threshold within a single epoch while the rate converges. Second, PCCP remains stable at a point that ensures that the vNIC ingress delay is close to the target threshold, with a mean delay of $9.05\mu s$, and 99^{th} perc. delay of $13.25\mu s$.

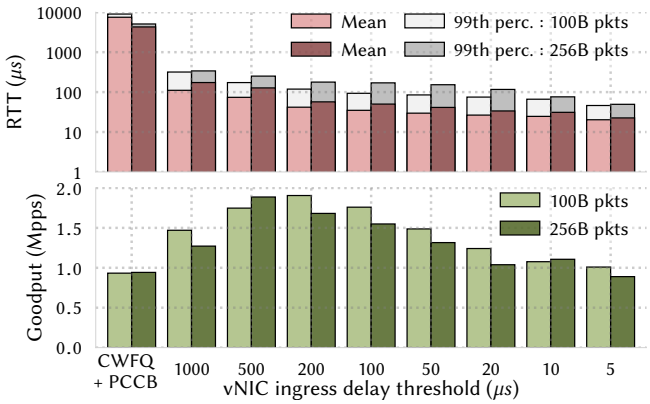


Figure 11: Goodput and RTT vs target engine delay.

High throughput and low delay. Next, we show that PicNIC delivers consistent low delay and high goodput for resource-intensive traffic while navigating strict isolation and efficiency. We extend the setup from Fig. 7a with two more VMs as before (E on h_2 and F on h_3). f_{AC} and f_{BD} are high-PPS UDP flows, while f_{EF} probes latency (Poisson process with rate $\lambda = 1$ kpps).

With just BPS envelope and 100-byte UDP packets, the UDP flows achieve a total goodput of 0.93 Mpps, while the mean f_{EF} RTT is $7.5ms$ as shown in Fig. 11. PCCP is able to reduce latency by over two orders of magnitude while delivering higher goodput for UDP flows. Fig. 11 (top) shows that as we lower the delay threshold, the measured RTT correspondingly reduces as expected. The bottom plot shows that the goodput improves by as much as 100% as the threshold is lowered to a point, perhaps counterintuitively. This is because PicNIC avoids drops at ingress, thus achieving higher efficiency which manifests as increased goodput. As the threshold becomes stricter, the rate limit for UDP flows is further reduced and ultimately leads to a decrease in goodput. However, even with the strictest threshold of $5\mu s$, PicNIC ensures a goodput (1 Mpps) comparable to the case with just BPS envelope, but with a remarkably lower mean f_{EF} RTT of $20.3\mu s$. We see similar results with 256-byte UDP packets.

6.2 Overheads, Response Time and Scalability

PicNIC's major overhead arises from implementing PCC in the datapath. To quantify the tradeoff between responsiveness and overheads, we measure the throughput of a TCP flow between two VMs placed on separate hosts while changing the PCC feedback interval. At the egress, we enforce a dummy high rate limit so that the flow is never throttled, but it will be affected as PicNIC uses CPU cycles. As shown in Fig. 12, if the feedback is received too frequently, e.g., every $10\mu s$, the goodput degrades by ~22%. However, as we see good convergence with $\epsilon \approx 500\mu s$ in §6.1.2, we expect to run PCC at a similar granularity. As we increase the interval, the overhead decreases and becomes negligible at 1 ms. We find that these overheads are good enough for rapid response to isolation breakages while remaining work-conserving in the common case. We note that while PCC is reactive, CWFQs and sender-side admission control are proactive and provide isolation at even faster timescales.

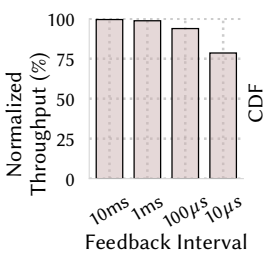


Figure 12: Overheads.

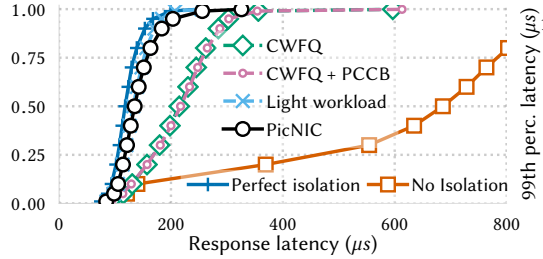


Figure 13: Memcached: Latency under low load (200 $kRPS$).

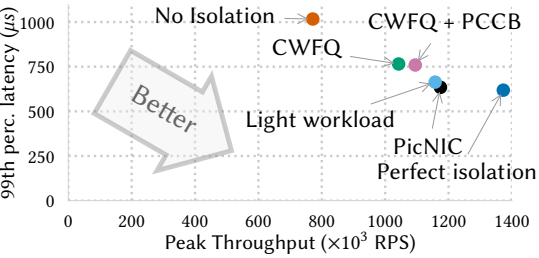


Figure 14: Memcached: Peak throughput and tail latency.

While the above microbenchmarks use a 20 Gbps NIC, we have evaluated PicNIC with faster NICs, such as 40 Gbps and higher, and find that PicNIC scales well with low overhead.

6.3 Application-level Performance

We quantify application-level benefits using a setup with two tenants (say, T1 and T2) and 48 VMs placed over 9 hosts. T1 runs memcached, an in-memory key-value store widely used as a low-latency caching layer for database applications [46]. T1 replicates Facebook's ETC workload [6, 54] on 24 VMs using `mutilate` load generator [42]. We ensure that memcached is not compute bottlenecked by overprovisioning with 8 memcached servers and 16 clients. T2 runs a UDP workload with 12 client-server pairs colocated with T1. Each UDP client generates 256-byte packets at 2 Mpps, i.e. at ~4 Gbps per source.

First, we set the baseline with no isolation, and then incrementally enable the different PicNIC components (§5): CWFQs, PCCB with envelope of 3 to 12 Gbps, and finally the complete PicNIC system with PCCP. We keep the egress constructs always enabled. For comparison, we consider two cases: i) *Perfect isolation*: T1 runs solely without T2 and with no `MAX_BPS` limit, and ii) *Light workload*: T2 runs a very light workload (100 kpps) to help set T1's performance target with PicNIC when resources are shared even under heavy workloads.

Latency. With no isolation, the median memcached response latency is 686μs compared to 118μs with perfect isolation, as shown in Fig. 13. CWFQs improve isolation at ingress, reducing the latency but do not prevent drops of excess traffic and wasted work. PCCB does little to improve latency because the UDP traffic is not consuming excessive bandwidth, and bandwidth is not the bottleneck for memcached either. Finally, with PCCP, PicNIC ensures performance isolation for memcached with 99^{th} perc. latency of 256μs, close to 200μs with perfect isolation (in contrast to 354μs with CWFQ+PCCB). In fact, the performance with PicNIC in the presence of high-PPS UDP is similar to the case when T1 is sharing the network with T2 running a light workload.

Throughput. Next, Fig. 14 shows the maximum achievable throughput and the corresponding tail latency for memcached. Starting with no isolation as baseline, as we incrementally enable individual PicNIC constructs, the performance approaches the perfect isolation case—i.e., when there is no contention for resources. Achieving the same level of performance is not possible in a shared setup—e.g., in a setup with `MAX_BPS` per VM of 15 Gbps and the NIC line-rate of 20 Gbps, a VM can get 15 Gbps with perfect isolation; but when another similar VM also shares the host, each VM can achieve only 10 Gbps. Even when sharing the network with a high-PPS workload tenant, PicNIC ensures performance isolation for T1 as if it was sharing the network with another light-workload tenant.

6.4 Production

To show the practicality of PicNIC, we deployed it in a large public cloud. We summarize a subset of results here; §C has more details. §2 showed how rate limiting slower flows at the egress introduces HoL blocking delay for faster flows. Deployments of out-of-order (OOO) completions and Packet Accounting *eliminated* HoL blocking between flows at the egress. Consequently, OOO completions, alone, improved the tail latency for customer traffic by ~13%.

On deploying CWFQs at the ingress, we see 96% reduction in packet drops at NIC Rx queues. Excess packets are dropped fairly at the per-VM queues. As noted in §5, CWFQs by themselves cannot eliminate NIC packet drops completely; when CWFQs are coupled with admission control, we observe further reduction in packet drops and response times in mitigating isolation breakages, even under DoS attacks. To handle congestion in the network fabric, we also extended our prototype to incorporate ECN signals.

7 Solution Space of Performance Isolation

Table 1 summarized related prior work which focus on sharing *fabric* bandwidth and not resources at *end-hosts*. PicNIC ensures isolation at end-hosts and complements prior work to build an end-to-end predictable virtualized network.

7.1 Mechanisms Used in Practice

Isolation is usually enforced using the following approaches.

Guest congestion control: While we have relied on flow-level congestion control, e.g. TCP, it is changing with the Cloud—VMs are free to use any congestion control, including none.

Static bandwidth limits: A common approach is to apply static egress BPS limits per-VM based on policy. As the limits are neither defined nor enforced for ingress, they can not ensure isolation in many cases—e.g., against incasts and DoS attacks.

Cloud product offerings: Achieving predictable performance in public cloud is challenging. A survey of major cloud providers indicates the lack of concrete performance guarantees and isolation mechanisms [3, 27, 47]. Even with the opt-in highest-performance options—e.g., SmartNIC [22] and AWS Enhanced Networking [4]—cloud providers only specify an "up to" BPS limit for each VM, similar to static bandwidth limits. None of the providers offer latency or loss-rate SLOs.

7.2 Related Work

Abstractions. A class of recent work has focused on building abstractions that aim to provide tenants an illusion of a dedicated network fabric [5, 7, 28, 41, 69, 76]. Such abstractions range from a static virtual cluster or datacenter to those encoding time-varying demands and communication patterns [7, 41, 76]. In contrast, PicNIC proposes the predictable vNIC abstraction at the VM level.

Dynamic bandwidth arbitration: For flow aggregates over WANs [30, 32], centralized systems can compute dynamic BPS allocations [38]. These are difficult to scale to enforce isolation at short timescales across thousands of VMs. Even proposed distributed approaches [35], which mostly share network bandwidth in the fabric, do not ensure isolation at end hosts. Enforcing BPS limits [5, 7, 8, 14, 28, 39, 41, 62, 63, 66, 69, 69, 76] do not automatically ensure end-to-end predictable latency and loss rate, which PicNIC aims to deliver. Further, compared to constant link capacities in the fabric, PicNIC provides predictable performance while sharing variable packet processing capacity of virtualization stacks.

Tradeoffs: FairCloud [62] and HUG [14] study fundamental tradeoffs with respect to network utilization and minimum bandwidth guarantees. HUG also explores fairness in the context of multiple resources [24, 25], which could be applicable in this case too.

8 Discussion on Hardware Design

8.1 Hardware is not a panacea

A classical approach to achieving isolation is to push the problem one level down. In this context, it means moving functionality and contention management into the hardware (HW), for instance by using per-VM queues or SR-IOV [57]. However, even physical NICs have PPS limits and have to share limited resources such as PCIe and DRAM bandwidth [26, 43, 53, 61]. Even using SR-IOV may lead to unfair sharing of resources and performance interference [13]. PPS overloads remain an Achilles heel for HW stacks too.

Using per-VM queues is challenging in practice as they are needed at all potential points of contention. Physical resource limitations imposed by the HW may mean insufficient number of queues in HW at every such point. Fundamentally, HW queues are "local" constructs lacking global visibility, and thus are insufficient to guarantee isolation which requires coordination with multiple senders. For instance, drops due to PCIe bandwidth exhaustion at the ingress can break isolation similar to §2.2. For the same reason, novel dataplane approaches [9, 59] based on HW virtualization cannot provide isolation on their own for the cases mentioned in §2. Recent work in the context of RDMA has also demonstrated isolation issues in HW [77]. Finally, as functionality moves to NICs [22, 72], it is crucial to ensure predictable sharing of HW resources.

8.2 PicNIC in Hardware

We briefly discuss how PicNIC's techniques can be applied in the case of HW-based stacks. Usually, the NIC or FPGA implementing the virtualization stack is connected over PCIe to the CPU. The major potential bottlenecks along the path include PCIe bandwidth and DRAM bandwidth. If one VM's traffic monopolizes these resources, it can lead to other VMs being starved unfairly and lead to similar issues as illustrated in §2. Even if such resources are shared fairly, packet drops can occur at ingress when the ingress rate for a VM exceeds its fair-share rate. This again leads to unfairness as well as inefficiency due to resource wastage. Thus, we need backpressure to the sources to avoid contention at the receivers. At egress, just rate limiting traffic can lead to drops and buffer contention in the stack when a VM discharges packets at excessive rates. This wastes resources such as PCIe and DRAM bandwidth, HW clock cycles and SRAM buffers. So, we need to apply backpressure to the guest OS stack as well. We find that the same design principles (§3) carry over in the case of HW too. We outline the corresponding PicNIC constructs for HW. For more details, see §A.

Ingress. In order to avoid unfair drops and delay in shared NIC Rx queues, PicNIC can implement per-VM queues in HW. By monitoring resources, such as PCIe bandwidth, used by each VM, PicNIC can enforce SLO-based fair sharing by controlling how these queues are scheduled.

Congestion Control. Both PCCB and PCCP are amenable to efficient HW implementations. While PCCB monitors the ingress BPS per VM, PCCP uses statistics from per-VM queues to compute rate limits which are notified via feedback generated in the datapath. At the egress, these rate limits are stored in a table, which may be partitioned between SRAM and DRAM based on resource constraints.

Egress. Recent work on scalable shaping [64, 71] can be leveraged to enforce these rates efficiently. As the HW has accurate information about when a packet is sent out, it can hold completion events till then in order to implement OOO completions.

Overall, we believe that PicNIC's design principles and constructs are well-suited for implementing the predictable virtualized NIC abstraction on HW-based stacks too. We hope that the lessons learned from PicNIC will cause performance isolation to be considered as a primary objective for virtualization stacks and inform the design of future NICs.

9 Conclusion

Isolation is a fundamental challenge in operating systems that is *exacerbated* by VMs and cloud platforms. Today, cloud providers face a dilemma: they must provide the illusion of an isolated virtual slice of hardware to tenants, without being *too* wasteful of the underlying resources. This paper presents PicNIC, a system that uses a combination of localized SLO-based resource sharing and end-to-end admission control, to provide the illusion of a dedicated NIC to VMs, while responding to potential isolation breakages within sub-*ms* timescales.

PicNIC opens up a number of interesting avenues for future research. Can we pack more VMs onto each physical host without sacrificing predictability? Where along the spectrum of isolation and efficiency should cloud providers operate? If the time for responding to isolation breakages can be further brought down to single-digit μs, what additional efficiencies might become possible? How can future NIC designs facilitate predictable performance? Therefore, we think of PicNIC as the first word about predictable virtualized NICs, complementing prior work on sharing the network fabric, not the last.

Acknowledgments. We would like to thank the anonymous SIGCOMM reviewers, Dina Papagiannaki, Jeff Mogul and our shepherd, Manya Ghobadi, for providing valuable feedback. This work was partially supported by NSF grants CCF-1637532 and CNS-1413972 and ONR grant N00014-15-1-2177.

References

[1] Mohammad Alizadeh, Berk Atikoglu, Abdul Kabbani, Ashvin Lakshmikantha, Rong Pan, Balaji Prabhakar, and Mick Seaman. 2008. Data Center Transport Mechanisms: Congestion Control Theory and IEEE Standardization. In *46th Annual Allerton Conference on Communication, Control, and Computing*. IEEE, Urbana-Champaign, IL, USA, 1270–1277.

[2] Mohammad Alizadeh, Albert Greenberg, Dave Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data Center TCP (DCTCP). In *SIGCOMM*. ACM, New Delhi, India, 63—74.

[3] Amazon. 2018. Amazon EC2. https://aws.amazon.com/ec2/

[4] Amazon. 2018. Enhanced Networking on Linux. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/enhanced-networking.html.

[5] Sebastian Angel, Hitesh Ballani, Thomas Karagiannis, Greg O'Shea, and Eno Thereska. 2014. End-to-end Performance Isolation Through Virtual Datacenters. In *OSDI*. USENIX Association, Broomfield, CO, 233–248.

[6] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. In *SIGMETRICS*. ACM, London, England, UK, 53–64.

[7] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. 2011. Towards Predictable Datacenter Networks. In *SIGCOMM*. ACM, Toronto, Canada, 242–253.

[8] Hitesh Ballani, Keon Jang, Thomas Karagiannis, Changhoon Kim, Dinan Gunawardena, and Greg O'Shea. 2013. Chatty Tenants and the Cloud Network Sharing Problem. In *NSDI*. USENIX Association, Lombard, IL, 171–184.

[9] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2014. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *OSDI*. USENIX Association, Broomfield, CO, 49–65.

[10] Theophilus Benson, Aditya Akella, and David A Maltz. 2010. Network Traffic Characteristics of Data Centers in the Wild. In *IMC*. ACM, Melbourne, Australia, 267–280.

[11] Scott Bradner and Jim McQuaid. 1999. Benchmarking Methodology for Network Interconnect Devices. https://www.ietf.org/rfc/rfc2544.txt.

[12] Randy Brown. 1988. Calendar Queues: A Fast O(1) Priority Queue Implementation for the Simulation Event Set Problem. *CACM* 31, 10 (1988), 1220–1227.

[13] Quan Chen, Hailong Yang, Minyi Guo, Ram Srivatsa Kannan, Jason Mars, and Lingjia Tang. 2017. Prophet: Precise QoS prediction on non-preemptive accelerators to improve utilization in warehouse-scale computers. In *ASPLOS*. ACM, Xián, China, 17–32.

[14] Mosharaf Chowdhury, Zhenhua Liu, Ali Ghodsi, and Ion Stoica. 2016. HUG: Multi-Resource Fairness for Correlated and Elastic Demands. In *NSDI*. USENIX Association, Santa Clara, CA, 407–424.

[15] Jonathan Corbet. 2012. TCP Small Queues. https://lwn.net/Articles/507065/. Online, accessed: 2019-07.

[16] Michael Dalton, David Schultz, Jacob Adriaens, Ahsan Arefin, Anshuman Gupta, Brian Fahs, Dima Rubinstein, Enrique Cauich Zermeno, Erik Rubow, James Alexander Docauer, Jesse Alpert, Jing Ai, Jon Olson, Kevin DeCabooter, Marc de Kruijf, Nan Hua, Nathan Lewis, Nikhil Kasinadhuni, Riccardo Crepaldi, Srinivas Krishnan, Subbaiah Venkata, Yossi Richter, Uday Naik, and Amin Vahdat. 2018. Andromeda: Performance, Isolation, and Velocity at Scale in Cloud Network Virtualization. In *NSDI*. USENIX Association, Renton, WA, 373–387.

[17] Peter Druschel and Gaurav Banga. 1996. Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems. In *OSDI*. USENIX Association, Seattle, WA, 261–275.

[18] Nick G Duffield, Pawan Goyal, Albert Greenberg, Partho Mishra, Kadangode K Ramakrishnan, and Jacobus E van der Merwe. 1999. A Flexible Model for Resource Management in Virtual Private Networks. In *SIGCOMM*. ACM, Cambridge, MA, 95–108.

[19] Nandita Dukkipati and Nick McKeown. 2006. Why Flow-Completion Time is the Right Metric for Congestion Control. *CCR* 36, 1 (2006), 59–62.

[20] Daniel E Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. 2016. Maglev: A Fast and Reliable Software Network Load Balancer. In *NSDI*. USENIX Association, Santa Clara, CA, 523–535.

[21] Daniel Firestone. 2017. VFP: A Virtual Switch Platform for Host SDN in the Public Cloud. In *NSDI*. USENIX Association, Boston, MA, 315–328.

[22] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *NSDI*. USENIX Association, Renton, WA, 51–66.

[23] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. 2007. HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm. In *Discrete Mathematics and Theoretical Computer Science*. Discrete Mathematics and Theoretical Computer Science, 137–156.

[24] Ali Ghodsi, Vyas Sekar, Matei Zaharia, and Ion Stoica. 2012. Multi-Resource Fair Queueing for Packet Processing. In *SIGCOMM*. ACM, Helsinki, Finland, 1–12.

[25] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. 2011. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *NSDI*. USENIX Association, Boston, MA, 323–336.

[26] Younghwan Go, Muhammad Asim Jamshed, YoungGyoun Moon, Changho Hwang, and KyoungSoo Park. 2017. APUNet: Revitalizing GPU as Packet Processing Accelerator. In *NSDI*. USENIX Association, Boston, MA, 83–96.

[27] Google. 2018. Google Compute Engine. https://cloud.google.com/compute

[28] Chuanxiong Guo, Guohan Lu, Helen J Wang, Shuang Yang, Chao Kong, Peng Sun, Wenfei Wu, and Yongguang Zhang. 2010. SecondNet: A Data Center Network Virtualization Architecture with Bandwidth Guarantees. In *CoNEXT*. ACM, Philadelphia, PA, 15:1–15:12.

[29] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. 2015. SoftNIC: A Software NIC to Augment Hardware. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-155* (2015).

[30] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. 2013. Achieving High Utilization with Software-Driven WAN. In *SIGCOMM*. ACM, Hong Kong, China, 15–26.

[31] Intel. 2018. Intel Ethernet Controller XL710 10/40 GbE. https://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/xl710-10-40-controller-spec-update.pdf.

[32] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jonathan Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. 2013. B4: Experience with a Globally Deployed Software Defined WAN. In *SIGCOMM*. ACM, Hong Kong, China, 3–14.

[33] Virajith Jalaparti, Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. 2012. Bridging the Tenant-Provider Gap in Cloud Services. In *SoCC*. ACM, Article 10, 10:1–10:14 pages.

[34] Keon Jang, Justine Sherry, Hitesh Ballani, and Toby Moncaster. 2015. Silo: Predictable Message Latency in the Cloud. In *SIGCOMM*. ACM, London, United Kingdom, 435–448.

[35] Vimalkumar Jeyakumar, Mohammad Alizadeh, David Mazières, Balaji Prabhakar, Albert Greenberg, and Changhoon Kim. 2013. EyeQ: Practical Network Performance Isolation at the Edge. In *NSDI*. USENIX Association, Lombard, IL, 297–312.

[36] M Tim Jones. 2010. Virtio: An I/O virtualization framework for Linux. *IBM White Paper* (2010).

[37] Teemu Koponen, Keith Amidon, Peter Balland, Martín Casado, Anupam Chanda, Bryan Fulton, Igor Ganichev, Jesse Gross, Paul Ingram, Ethan Jackson, Andrew Lambeth, Romain Lenglet, Shih-Hao Li, Amar Padmanabhan, Justin Pettit, Ben Pfaff, Rajiv Ramanathan, Scott Shenker, Alan Shieh, Jeremy Stribling, Pankaj Thakkar, Dan Wendlandt, Alexander Yip, and Ronghua Zhang. 2014. Network Virtualization in Multi-tenant Datacenters. In *NSDI*. USENIX Association, Seattle, WA, 203–216.

[38] Alok Kumar, Sushant Jain, Uday Naik, Anand Raghuraman, Nikhil Kasinadhuni, Enrique Cauich Zermeno, C Stephen Gunn, Jing Ai, Björn Carlin, Mihai Amarandei-Stavila, Mathieu Robin, Aspi Siganporia, Stephen Stuart, and Amin Vahdat. 2015. BwE: Flexible, Hierarchical Bandwidth Allocation for WAN Distributed Computing. In *SIGCOMM*. ACM, London, United Kingdom, 1–14.

[39] Vinh The Lam, Sivasankar Radhakrishnan, Rong Pan, Amin Vahdat, and George Varghese. 2012. Netshare and Stochastic Netshare: Predictable Bandwidth Allocation for Data Centers. *CCR* 42, 3 (June 2012), 5–11.

[40] Jason Lawley. 2014. Understanding Performance of PCI Express Systems. *WP350 (v1. 2). Xilinx* (2014).

[41] Jeongkeun Lee, Yoshio Turner, Myungjin Lee, Lucian Popa, Sujata Banerjee, Joon-Myung Kang, and Puneet Sharma. 2014. Application-driven Bandwidth Guarantees in Datacenters. In *SIGCOMM*. ACM, Chicago, IL, 467–478.

[42] Jacob Leverich. 2014. Mutilate: High-Performance Memcached Load Generator. https://github.com/leverich/mutilate.

[43] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. 2017. KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC. In *SOSP*. ACM, Shanghai, China, 137–152.

[44] Linux. 2018. NAPI (New API). https://wiki.linuxfoundation.org/networking/napi. Online, accessed: 2019-07.

[45] Rob McGuinness and George Porter. 2018. Evaluating the Performance of Software NICs for 100-Gb/s Datacenter Traffic Control. In *ANCS*. ACM, Ithaca, NY, 74–88.

[46] Memcached. 2018. Memcached key-value store. https://memcached.org/. Online, accessed: 2019-07.

[47] Microsoft. 2018. Azure. https://azure.microsoft.com/.

[48] Radhika Mittal, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. 2015. TIMELY: RTT-based Congestion Control for the Datacenter. In *SIGCOMM*. ACM, London, United Kingdom, 537–550.

[49] Jeffrey C. Mogul and Lucian Popa. 2012. What We Talk About when We Talk About Cloud Network Performance. *CCR* 42, 5 (Sept. 2012), 44–48.
[50] Jeffrey C. Mogul and K. K. Ramakrishnan. 1997. Eliminating Receive Livelock in an Interrupt-driven Kernel. *ACM Trans. Comput. Syst.* 15, 3 (Aug. 1997), 217–252.
[51] Al Morton. 2013. IMIX Genome: Specification of Variable Packet Sizes for Additional Testing. https://tools.ietf.org/html/rfc6985.
[52] David Mulnix. 2017. Intel® Xeon® Processor Scalable Family Technical Overview. https://software.intel.com/en-us/articles/intel-xeon-processor-scalable-family-technical-overview. Online, accessed: 2019-07.
[53] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W Moore. 2018. Understanding PCIe performance for end host networking. In *SIGCOMM*. ACM, Budapest, Hungary, 327–341.
[54] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. 2013. Scaling Memcache at Facebook. In *NSDI*. USENIX Association, Lombard, IL, 385–398.
[55] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. 2016. NetBricks: Taking the V out of NFV. In *OSDI*. USENIX Association, Savannah, GA, 203–216.
[56] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A. Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, Changhoon Kim, and Naveen Karri. 2013. Ananta: Cloud Scale Load Balancing. In *SIGCOMM*. ACM, Hong Kong, China, 207–218.
[57] PCI-SIG. 2019. Single Root I/O Virtualization (SR-IOV). https://pcisig.com/specifications/iov. Online, accessed: 2019-07.
[58] Luis Pedrosa, Rishabh Iyer, Arseniy Zaostrovnykh, Jonas Fietz, and Katerina Argyraki. 2018. Automated Synthesis of Adversarial Workloads for Network Functions. In *SIGCOMM*. ACM, Budapest, Hungary, 372–385.
[59] Simon Peter, Jialin Li, Irene Zhang, Dan RK Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. 2014. Arrakis: The Operating System is the Control Plane. In *OSDI*. USENIX Association, Broomfield, CO, 1–16.
[60] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, Keith Amidon, and Martín Casado. 2015. The Design and Implementation of Open vSwitch. In *NSDI*. USENIX Association, Oakland, CA, 117–130.
[61] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. 2018. Floem: A Programming System for NIC-Accelerated Network Applications. In *OSDI*. USENIX Association, Carlsbad, CA, 663–679.
[62] Lucian Popa, Gautam Kumar, Mosharaf Chowdhury, Arvind Krishnamurthy, Sylvia Ratnasamy, and Ion Stoica. 2012. FairCloud: Sharing the Network in Cloud Computing. In *SIGCOMM*. ACM, Helsinki, Finland, 187–198.
[63] Lucian Popa, Praveen Yalagandula, Sujata Banerjee, Jeffrey C. Mogul, Yoshio Turner, and Jose Renato Santos. 2013. ElasticSwitch: Practical Work-Conserving Bandwidth Guarantees for Cloud Computing. In *SIGCOMM*. ACM, Hong Kong, China, 351–362.
[64] Sivasankar Radhakrishnan, Yilong Geng, Vimalkumar Jeyakumar, Abdul Kabbani, George Porter, and Amin Vahdat. 2014. SENIC: Scalable NIC for End-Host Rate Limiting. In *NSDI*. USENIX Association, Seattle, WA, 475–488.
[65] Scott Rixner, William J Dally, Ujval J Kapasi, Peter Mattson, and John D Owens. 2000. Memory access scheduling. In *ACM SIGARCH Computer Architecture News*, Vol. 28. ACM, 128–138.
[66] Henrique Rodrigues, Jose Renato Santos, Yoshio Turner, Paolo Soares, and Dorgival O Guedes. 2011. Gatekeeper: Supporting Bandwidth Guarantees for Multi-tenant Datacenter Networks. In *WIOV*. USENIX Association, Portland, OR, 784–789.
[67] Ahmed Saeed, Nandita Dukkipati, Valas Valancius, Terry Lam, Carlo Contavalli, and Amin Vahdat. 2017. Carousel: Scalable Traffic Shaping at End-Hosts. In *SIGCOMM*. ACM, Los Angeles, CA, 404–417.
[68] SDNCentral. 2014. Brocade Vyatta 5600 vRouter: Performance Validation. https://networkbuilders.intel.com/docs/Vyatta_5600_Performance_Test_Full_Report.pdf Online, accessed: 2019-07.
[69] Alan Shieh, Srikanth Kandula, Albert G Greenberg, Changhoon Kim, and Bikas Saha. 2011. Sharing the Data Center Network. In *NSDI*. USENIX Association, Boston, MA, 309–322.
[70] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. 2016. Programmable Packet Scheduling at Line Rate. In *SIGCOMM*. ACM, Florianópolis, Brazil, 44–57.
[71] Brent Stephens, Aditya Akella, and Michael Swift. 2019. Loom: Flexible and Efficient NIC Packet Scheduling. In *NSDI*. USENIX Association, Boston, MA, 33–46.
[72] Brent Stephens, Aditya Akella, and Michael M Swift. 2018. Your Programmable NIC Should be a Programmable Switch. In *HotNets*. ACM, Redmond, WA, 36–42.
[73] Tolly. 2016. Mellanox Spectrum vs. Broadcom StrataXGS Tomahawk: 25GbE and 100GbE Performance Evaluation – Evaluating Consistency & Predictability. https://www.mellanox.com/related-docs/products/tolly-report-performance-evaluation-2016-march.pdf. Online, accessed: 2019-07.
[74] George Varghese and Tony Lauck. 1987. Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility. In *SOSP*. ACM, Austin, Texas, USA, 25–38.
[75] Wikipedia. 2019. Internet_Mix. https://en.wikipedia.org/wiki/Internet_Mix. Online, accessed: 2019-07.
[76] Di Xie, Ning Ding, Y Charlie Hu, and Ramana Kompella. 2012. The Only Constant is Change: Incorporating Time-varying Network Reservations in Data Centers. In *SIGCOMM*. ACM, Helsinki, Finland, 199–210.
[77] Yiwen Zhang, Juncheng Gu, Youngmoon Lee, Mosharaf Chowdhury, and Kang G Shin. 2017. Performance Isolation Anomalies in RDMA. In *Workshop on Kernel-Bypass Networks (KBNets)*. ACM, 43–48.
[78] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. 2015. Congestion Control for Large-scale RDMA Deployments. In *SIGCOMM*. ACM, London, United Kingdom, 523–536.

Appendices

Appendices are supporting material that has not been peer reviewed.

A PicNIC for Hardware-based Stacks

While we have described PicNIC's design and implementation for a software-based network virtualization stack, we find that (i) isolation can break, resulting in unpredictable performance, even with virtualization stacks implemented in hardware [22], and (ii) PicNIC's design principles (§3) can ensure predictable performance in such stacks as well.

A.1 Causes of Unpredictable Performance

Isolation breakage in hardware also arise from contention for shared resources. Consider a hardware NIC or FPGA-based virtualization stack connected to the CPU over PCIe. Fig. 15 shows a simplified view on a host with a modern system-on-chip (SoC) architecture such as Intel Skylake [52]. The NIC hardware can be connected over PCIe either directly to the I/O controller in the System Agent on the SoC, or through a Platform Controller Hub (PCH). The primary resource under contention is the bandwidth between the NIC and the main memory (DRAM) where packets are delivered to the guest OS networking stack. There are multiple potential bottlenecks along the path: PCIe bandwidth (① or ②), DMI bandwidth (③) and the DRAM bandwidth (④).

PCIe bandwidth. Consider the NIC at either `dev0` or `dev1` in Fig. 15 connected through PCIe Gen3 x8, commonly used for 40Gb/s NICs today [31]. While the theoretical raw bit-rate offered by PCIe Gen3 x8 is 8 GT/s × 8 bits = 64 Gb/s, the entire bandwidth is not achievable for data transfer because of, for example, encoding, packetization, and protocol overheads of PCIe. In fact, the theoretical throughput when accessing main memory in 64B granularity from NIC turns out to be 44.8 Gb/s [43], and credit-based PCIe protocols for reliable read and write further reduce the effective bandwidth [40]. In addition to packet DMA, the NIC also shares this bandwidth for operations such as reading/writing descriptors, updating queue pointers and signalling interrupts. Performing this naively results in effective bandwidth for packet DMA to be less than 40 Gb/s [53] and thus PCIe bandwidth becomes a bottleneck, especially with small packets.

DMI bandwidth. If the NIC is connected to the PCH on the chipset, the PCH to System Agent connection can introduce another bottleneck. In the Intel Skylake architecture, this connection is through

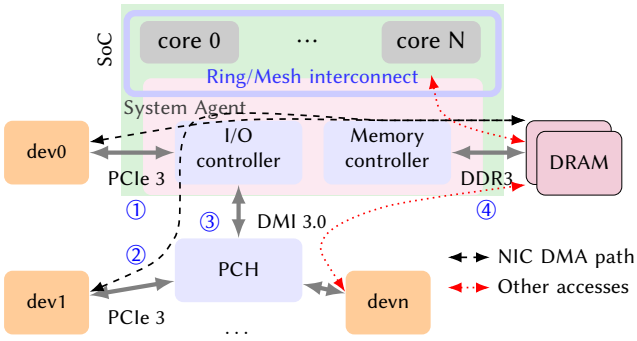


Figure 15: Potential bottlenecks within host hardware.

Direct Media Interface (DMI) 3.0, which offers a maximum bandwidth equivalent to PCIe Gen3 x4 with raw bit-rate of 8 GT/s × 4 bits = 32 Gb/s.

DRAM bandwidth. Finally, packets need to be written to or read from guest OS stack memory on the host DRAM. Each DRAM channel has a bandwidth that depends on the DRAM frequency and bus-width. For instance, a single-channel DDR3-1600 which runs at 1600 MT/s with a 64-bit bus-width offers a peak bandwidth of 102.4 Gb/s. The sustainable bandwidth for DRAM is much lower (typically 70-80%) than the peak owing to factors such as access patterns [65]. Even though DRAM bandwidth in modern systems is much greater than networking bandwidth, the DRAM bandwidth is also shared by others such as I/O and compute. This leads to memory isolation issues in shared public clouds, and also affects networking performance.

Isolation breakage. In practice, PCIe bandwidth can often be a primary bottleneck [43, 53, 61]. Sharing these resources based on SLOs is key to ensuring predictable performance (design principle $\mathcal{P}1$). Failing to do so, e.g., if one VM monopolizes PCIe bandwidth, can lead to other VMs being starved unfairly with unpredictable network performance. Fair-sharing at ingress may lead to packets being dropped at ingress for VMs receiving packets at rates exceeding their fair-share. Drops at ingress lead to wasted resources at egress, in the fabric, and at the ingress NIC. So, we need to implement admission control by creating fair backpressure all the way to the sources and applying appropriate rate limits at the egress NIC (design principle $\mathcal{P}2$). Egress rate limiting, by itself, isn't sufficient to ensure isolation at the egress as we showed in §2.1. If VMs keep sending excessive packets for throttled flows that are going to be dropped by the traffic shaper in the NIC, it leads to unnecessarily wastage of DRAM and PCIe bandwidth, hardware clock cycles, and limited SRAM that buffers packets in the NIC for shaping. Thus, we need to augment in-hardware rate limiting with appropriate backpressure to the guest networking stack.

A.2 Hardware Design of PicNIC

Based on the design principles of PicNIC (§3), and the causes of unpredictable performance in hardware-based virtualization stacks, PicNIC's constructs can be adopted in hardware as outlined next.

Ingress. At the ingress, PicNIC implements per-VM queues in hardware so that packets can be moved from shared NIC Rx queues to per-VM queues at NIC line rate and avoid unfair NIC drops. Unlike the software-based implementation where we need to explicitly provision enough CPU cycles for this function, in case of HW, this module can hash packet headers and enqueue to the per-VM queue independent of other functions. To ensure that the NIC to DRAM bandwidth is shared as per-SLO, PicNIC keeps track of the PCIe bandwidth (which can be different from the network bandwidth because PCIe bandwidth overheads are per-TLP) used per VM and schedules per-VM queues based on SLOs and PCIe bandwidth usage. Thus, packets belonging to offending flows (sending more than SLOs and breaking isolation) will experience queueing delays and may also be dropped fairly in the per-VM queues, while packets belonging to well-behaved flows continue to experience low delays. Packets may also be dropped in per-VM queues if the receiver VM is slow (e.g. compute bottlenecked) and is not able to consume packets destined to it.

To ensure VMs receive traffic at a predictable bandwidth, PicNIC implements timing wheel (TW) [74] in NIC hardware at the ingress. Based on per-VM ingress rate, the TW sets a timestamp for each packet which decides when the packet is released from the stack and delivered to the VM. Thus, VMs receive regularly paced packets as per their SLO instead of bursts. Packets that exceed the timing wheel quota are dropped. However, we expect PCCB to implement proper admission control and avoid drops at ingress.

Congestion Control. PicNIC needs to avoid ingress drops as it leads to wasted resources and decreases efficiency. Again, the delay experienced by packets in per-VM queues acts as a good signal for how much the ingress rate exceeds the SLO-based fair share, and it is also an early indicator for drops. Using this delay as congestion signal, PicNIC implements PCCP to apply backpressure to the sources under PCIe bandwidth contention. Both PCCB and PCCP are implemented in hardware using a similar approach as the software-based implementation. The feedback packets can be generated and consumed in the NIC without involving the host CPU. The state needed for congestion control at ingress (~5kB) consumes a very small fraction of on-NIC SRAM. At the egress, the VM-VM rate-limit table can be maintained either completely in the NIC or in the DRAM (with entries cached in NIC) depending on the trade-off between on-NIC SRAM capacity and the PCIe and memory bandwidth needed for read/write to in-memory table. The choice also depends on how the virtualization stack implements other lookup tables.

Egress Shaping and backpressure. At the egress, PicNIC implements flexible shaping framework in HW that can enforce a hierarchy of rates—e.g., total egress rate limit per-VM and VM-to-VM rate limits—while ensuring isolation. PicNIC uses a combination of i) an on-NIC scheduler and ii) per-flow (⟨VM-src, VM-dst, vnet-id⟩) FIFO queues in main memory to achieve scalable shaping similar to SENIC [64] and Loom [71]. When a packet is ready to be sent by a VM, the guest vNIC driver enqueues the packet in the corresponding per-flow queue but doesn't mark the packet as transmission complete immediately. The driver sends the packet descriptor along with any metadata, e.g. VM-src, VM-dst and vnet-id, needed to make scheduling decisions to the NIC as a doorbell write using memory-mapped I/O over PCIe. The on-NIC scheduler uses the metadata to compute the egress timestamp for each packet based on the applicable rate limits and enqueues the descriptor to a calendar queue [12, 67, 70]. When the on-NIC scheduler dequeues a descriptor from this queue, it issues a DMA read request to fetch the corresponding packet

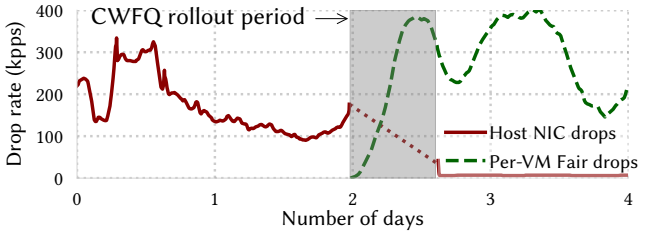


Figure 17: PicNIC's CWFQs reduce unfair ingress drops at host NIC by 96% and drop packets fairly on a per-VM basis.

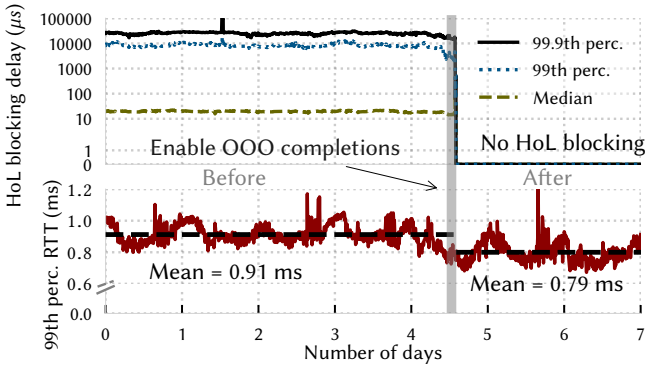


Figure 16: OOO completions with PicNIC eliminates HoL blocking delay and decreases tail RTT by 13% in production.

from DRAM, performs any on-NIC virtualization functions (e.g., encapsulation) and sends it on the wire. This avoids the need for buffering packets in the NIC. When reading a packet from DRAM, the NIC also generates the corresponding completion event by writing to the in-DRAM completion queue. The driver forwards these completion events as generated by the NIC. As completions are ordered by the scheduler based on rate limits, they can be out of order w.r.t. descriptor ordering. This avoids HoL blocking as discussed in §5.4.3. Since the on-NIC scheduler needs to store a reference to each packet until it is sent out, and on-NIC SRAM is limited, we need to ensure that the number of pending descriptors is bounded. PicNIC achieves this using packet accounting (§5.4.2) by limiting each per-VM on-DRAM queue capacity, and hence prevents rate limited flows from exhausting on-NIC SRAM capacity for packet descriptors. As further optimization, e.g. to minimize the number of PCIe writes, doorbell writes can be batched together [43, 71].

B Deferred Completions

Deferred Completions in networking stacks return the completion of a packet to VM only once processing has been completed and the packet has left the NIC.

To enable deferred completions without support for out-of-order completions, a "reordering buffer" is required. Basically, before a completion is returned to the VM, it is first put in a buffer. The order of completions in this buffer is then checked, and if there is a batch of reordered completions that can now be returned, all are returned. If not, this completion is held until all missing completions become available. We call this behavior deferred completion.

C Production Results

To demonstrate the practicality of PicNIC, we deployed it in production of a large-scale public cloud provider. We present a subset of results based on the deployment.

Egress. Of PicNIC's egress features (§5.4), we show the results with Packet Accounting + out-of-order (OOO) completions. Of the Guest OS features, TSQ is enabled by default in Linux based kernels, while NAPI-TX is a feature we are working to turn on in production. Packet Accounting improves isolation for buffered traffic across VMs as shown in §6.1. This feature is enabled throughout the period of interest; in the interest of space, we do not tease out the before/after impact of Packet Accounting.

§2 demonstrated how rate limiting flows at the egress can introduce HoL blocking delay for non-rate-limited flows. Fig. 16 (top) shows the extent of HoL blocking delay in production before and after we deployed OOO completions. OOO completions and Packet Accounting eliminate HoL blocking between rate-limited and non-rate-limited flows at the egress. Consequently, OOO completions, by itself, also improves the tail latency for customer traffic by ~13% as shown in Fig. 16 (bottom).

Ingress. The key construct at the ingress is CWFQ. Fig. 17 shows that as a result of CWFQs, we see 96% decrease in packet drops at NIC Rx queues. Excess packets are dropped at the per-VM queues. We note that CWFQs by themselves cannot eliminate NIC packet drops when the incoming PPS load is greater than the engine's capacity to pull packets from NIC Rx queues, such as DoS attacks and incast type workloads. When CWFQs are coupled with congestion control, we observe a substantial reduction in packet drops and a sub-*ms* response time in mitigating isolation breakages even under DoS attacks.

Fabric Congestion. We expect PicNIC to complement prior work on performance isolation in the network fabric in order to guarantee predictable network performance to tenants. We built a prototype of such a complete system that can handle congestion in the fabric. For this, we extended PicNIC to incorporate ECN signals from the fabric and use a DCTCP-like algorithm [2] to compute the appropriate rates in PCC.