zD: A Scalable Zero-Drop Network Stack at End Hosts

Yimeng Zhao
Georgia Institute of Technology

Ahmed Saeed
Massachusetts Institute of Technology

Ellen Zegura
Georgia Institute of Technology

Mostafa Ammar
Georgia Institute of Technology

ABSTRACT

Modern end-host network stacks have to handle traffic from tens of thousands of flows and hundreds of virtual machines per single host, to keep up with the scale of modern clouds. This can cause congestion for traffic egressing from the end host. The effects of this congestion have received little attention. Currently, an overflowing queue, like a kernel queuing discipline, will drop incoming packets. Packet drops lead to worse network and CPU performance by inflating the time to transmit the packet as well as spending extra effort on retansmissions. In this paper, we show that current end-host mechanisms can lead to high CPU utilization, high tail latency, and low throughput in cases of congestion of egress traffic within the end host. We present zD, a framework for applying backpressure from a congested queue to traffic sources at end hosts that can scale to thousands of flows. We implement zD to apply backpressure in two settings: i) between TCP sources and kernel queuing discipline, and ii) between VMs as traffic sources and kernel queuing discipline in the hypervisor. zD improves throughput by up to 60%, and improves tail RTT by at least 10x at high loads, compared to standard kernel implementation.

CCS CONCEPTS

• **Networks** → **Network architectures**; • **Software and its engineering** → *Communications management*.

KEYWORDS

Network Architecture, Backpressure, Queuing Architecture, Congestion Control

ACM Reference Format:
Yimeng Zhao, Ahmed Saeed, Ellen Zegura, and Mostafa Ammar. 2019. zD: A Scalable Zero-Drop Network Stack at End Hosts. In *The 15th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '19), December 9–12, 2019, Orlando, FL, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3359989.3365425

1 INTRODUCTION

For years, improved chips added more cores rather than more capacity per core [34]. Rather than relying on improved performance through increased per-core performance, parallel execution became the only way of making use of the new chips [4, 15]. From the perspective of the networking stack, this meant that rather than having to serve a few connections per machine, new networking stacks have to cope with requirements in the tens of thousands of connections per machine (e.g., reports mention servers handling up to 50k flows per end host [28]). This is further enabled by advancements in virtualization and containerization that allows applications belonging to different users to coexist and share network resources on the same end host (e.g., reports mention 120 VMs per end host [23]). This scale sparked interest in improved scheduling and prioritization between these applications through the introduction of efficient packet processing and scheduling mechanisms [1, 3, 23, 28, 29]. Processing of egress traffic in such stacks relies on holding packets in a cascade of queues pending their processing and eventual scheduling to be transmitted on the wire.

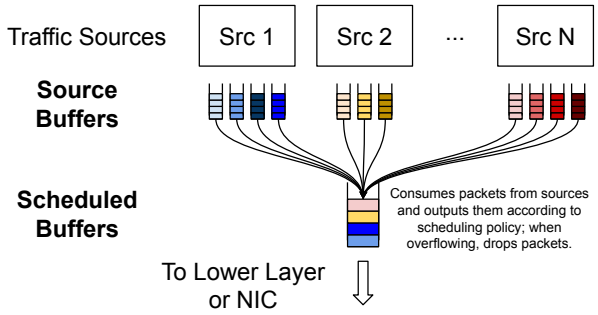


Figure 1: Schematic of queue architecture at end hosts.

Packet queues at an end host serve as buffers between producers and consumers with different speeds. There are two types of buffers we are interested in: *source buffers* and *scheduled buffers*. *Source buffers* hold packets prepared by traffic sources while awaiting consumption by the underlying layer in the stack. *Scheduled buffers* consume packets from multiple traffic sources and then determine the order of their transmission according to their configured scheduling policy. While most components of the networking stack have evolved to cope with the growing scale of applications, handling of overflowing scheduled buffers which can lead to packet drops has received little attention.

Figure 1 shows how packets flow in a typical system. Packets are first sent from source buffers (e.g., TCP socket buffers) to a scheduled buffer (e.g., Qdisc [7]). Packets from different sources accumulate in the scheduled buffer. If a scheduled buffer runs out of space, packets are dropped. Examples of such end-host congestion exists in large scale public clouds where a single end host is shared between multiple applications [14].

CoNEXT '19, December 9–12, 2019, Orlando, FL, USA

ACM ISBN 978-1-4503-6998-5/19/12…$15.00
https://doi.org/10.1145/3359989.3365425

In general, packet drops are inefficient. Resources used on processing the dropped packet (e.g., CPU) have to be used again to send the retransmission. Moreover, drops increase latency by adding more processing time to attempt retransmissions. Finally, drops can also induce severe reaction from congestion control which cuts its window in reaction to packet loss, leading to lowered throughput. When packet drops occur inside the network, then this type of inefficiency is unavoidable because of the need for end-to-end signaling. However, if packets are dropped inside the source host in the manner described in the scenario of Figure 1, then they can be handled through signaling within the host. For this latter type of loss we find that they are responsible for a up to 14% increase in CPU utilization and an order of magnitude increase in tail latency (§2). Our goal is to consider how signaling within the host can recover from these packet drops faster (in nanoseconds to microseconds as opposed to microseconds to milliseconds) while avoiding the CPU overhead.

A few proposals attempt to avoid packet drops of egress traffic at end hosts. However, their approaches have poor performance when handling a large number of senders. For example, the simplest approach is to increase the queue size which is a known cause of bufferbloat [8]. TCP Small Queue (TSQ) is one attempt to partially address this problem by limiting the number of packets per TCP socket to two packets [12]. This approach requires having $O(N)$ queue size at the end host, where N is the number of senders. TSQ works well for cases where N is between hundreds to a couple of thousands of flows. However, as N grows, TSQ can still suffer from bufferbloat issues as the number of packets in the queue grows (§2.3). Delayed-completion [28] was proposed as an approach to leverage the benefits of TSQ outside the scope of the kernel stack (e.g., in a userspace stack [1]). However, this approach inherits TSQ's scalability problem (i.e., requiring $O(N)$ queue size).

In this paper, we introduce the design, implementation, and evaluation of zD, a new architecture for handling congestion of scheduled buffers. zD has three components (§3): 1) a source buffer regulator that allows a congested scheduled buffer to pause and resume a traffic source, ii) a CPU efficient backpressure interface to define the interaction between the congested scheduled buffer and the traffic sources, and iii) a scheduler for paused flows to make sure that zD does not interfere with the scheduling policy implemented in the scheduled buffer. zD allows network operators to set a fixed queue size that is independent of the number of flows, eliminating bufferbloat issues at scale. zD maintains CPU efficiency by defining a backpressure interface that triggers packet dispatch from senders only when the scheduled buffer has room for new packets[1]. The task performed by zD can be viewed as controlling access to the scheduled buffer rather than leaving it the CPU scheduler. Thus, zD also reduces contention in accessing the scheduled buffer, further saving CPU resources. zD avoids interfering with the scheduling policy implemented in the packet queue (e.g., Qdisc policy) by scheduling flows in a way that is consistent with the underlying packet scheduling policy. To achieve CPU efficient scheduling, zD leverages recent developments in software schedulers introduced by the Eiffel system [29].

[1]Note that drops due to packet corruption can still happen.

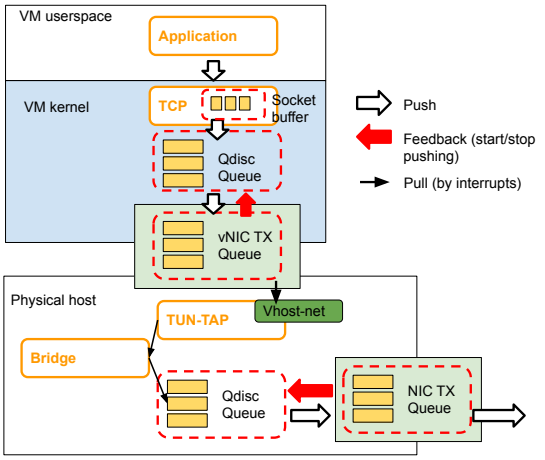


Figure 2: Architecture of queues in end hosts.

We implement zD[2] (§5) in the Linux kernel to handle backpressure for two cases: 1) when the queues and traffic sources are within the kernel stack (i.e., in the same virtual or physical machine), and 2) when the traffic sources are in the virtual machine and the queues are in the hypervisor. We find that zD can significantly improve network performance at high loads (§6). In particular, zD improves throughput by up to 60%, reduces retansmission by up to 1000x, and improves tail RTT by at least 10x at high loads. Furthermore, zD improves CPU utilization spent on the networking stack by up to 2x at the end host by reducing the effort spent on resending packets that have been dropped. We also find that zD is lightweight as it does not incur extra overhead when the system is operating at low utilization. The only downside to zD is that in some scenarios it can increase the CPU overhead inside the hypervisor.

2 BACKGROUND AND MOTIVATION

2.1 Packet Queuing at End Hosts

We start by giving an overview of the packet queuing architecture at end hosts. We focus on a common architecture used in modern data centers. In particular, we focus on the case where the end host is running a Linux virtualized environment, where the IO driver interface between the guest and host is handled by `virtio` [27] and `vhost` [33]. `virtio` is an I/O para-virtualized (PV) standard used for connecting the guest and host. To avoid context switching in the host, `vhost` allows the dataplane of the guest to be mapped directly into the kernel space of the host. The queuing architecture is shown in Figure 2. We focus on queues in the packet path and differentiate between queues where it is possible to have packet drops and those that already have a form of backpressure.

The user space application in the VM generates a packet and copies it into the kernel space socket buffer. The return value of the socket system call indicates whether the socket buffer is full. This operation is lossless (i.e., zero drop). Packets from the socket buffer are then queued into a Queuing Discipline (Qdisc). Packet drops can happen if the Qdisc is full. This happens when sockets push packets faster than the Qdisc transmission speed. Next, the Qdisc

[2]zD Code and a tutorial for using it are available at https://zd-linux.github.io/

sends the packet to the vNIC TX queue. The vNIC TX queue does not drop packets. In particular, when there is no available space in the vNIC TX queue, the Qdisc will be paused until the queue length drops below the threshold, making the Qdisc the primary location for drops in the VM.

The hypervisor processes packets generated by the VM through `vhost` which starts a kernel thread that performs busy polling on the queue between the hypervisor and the VM. There can be multiple such queues, called `vrings` with a different `vhost` thread assigned for each `vring`. The `vhost` process polls the packet and sends it through the TAP device then to a Bridge device. Packets received in the virtual bridge will be forwarded to the Qdisc in the physical machine and then transmitted to the NIC TX queue. Note that in this setting, the TAP and Bridge devices do not hold or drop packets, delivering all packets they process in order to the Qdisc in the hypervisor. The Qdisc, or its counter part in a more complicated architecture (e.g., OpenVSwitch [24]), is the main place where packets can be dropped due to congestion in the hypvervisor. We mark the existing backpressure mechanism (i.e., zero drop) with solid red arrows in Figure 2.

We choose this setting because it is a stripped-down, yet general-purpose, virtualized network stack. This architecture shares the same queuing components with more complicated architectures. For instance, consider Andromeda [14], Google's virtual network stack. Andromeda relies on a similar basic architecture and augments it with an efficient fast path. Note that packet drops can only happen at Andromeda itself which corresponds to the Qdisc in the above architecture. Furthermore, the architecture we consider here, unlike DPDK-based stacks, does not require a spinning core dedicated for network processing. This allows us to perform fine grain measurements of CPU efficiency (e.g., experiments where the VM runs on a single core). This architecture also captures the major characteristics of other stacks in terms of potential for packet drops at the end host. For instance, `vhost` used in our architecture has an analogous `vhost-user` used in DPDK-based stacks where packet queues will be in the userspace network processing system. In cases where OpenVSwitch [24] is used, the TAP and Bridge devices are replaced by OpenVSwitch. Hence, we find that the conceptual building blocks we develop in this paper for solving the congestion problem apply to other settings.

Ingress traffic: Most packet drops of egress traffic can be handled by coordination within the sender. However, drops of ingress traffic can require end-to-end coordination [9, 17] or careful allocation of CPU resources [22]. We focus on packet drops that occur due to congestion that can be handled through signaling within the end host, which are mostly egress traffic packet drops.

2.2 Types of Packet Drops

In-network packet drops are easily defined as packets being discarded by a network element (e.g., switch). This singular definition typically has some well defined reaction from the source associated with it (e.g., retransmission of the lost packet and congestion control reacting by adjusting its window). However, at end hosts we find that there are two types of packet drops. Both types of drops are expensive because a packet is processed for transmission, destroyed, and a replacement packet has to be generated which leads to higher CPU cost as well as higher latency. However, the two types differ in the reaction of the traffic source.

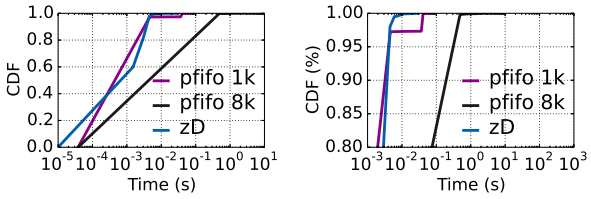
Virtual Packet Drops: In such cases the traffic source is aware that the packet was dropped at the end host. This type of drop is only feasible when transmission through the stack is performed through a series of nested function calls. The return value of these functions indicates whether the packet was successfully transmitted or dropped by one of these functions. If a packet is virtually dropped, the caller becomes aware of the location of the drop, allowing it to react appropriately. For instance, the reaction of TCP to a detected virtual packet drop is to simply attempt to resend the dropped packet without triggering its retransmission mechanisms and the congestion control algorithms.

Physical Packet Drops: In such cases the traffic source is unaware that the drop happened at the end host and consequently reacts as if the packet was dropped in the network. For example, the reaction of TCP to a physical packet drop will include triggering retransmission and congestion control algorithms. This type of drop is more expensive as it can lead to reduced network utilization, due to congestion control reaction (i.e., forcing flows to operate at a low rate), in addition to the higher CPU cost and latency.

In the stack described in Figure 2, virtual packet drops happen inside the VM where the TCP stack is aware of Qdisc packet drops. In current implementations, TCP reacts to virtual packet drops by immediately attempting to resend the dropped packet without consideration to contention at the Qdisc. This is a CPU intensive approach as we discuss in the next section. Physical packet drops occur in the hypervisor Qdisc which does not explicitly report drops to the guest kernel. Another important distinction between the two types of drops is that physical packet drops can be completely avoided. However, virtual packet drops are necessary in some cases. For example, a new flow cannot know whether the queue is full or not until it probes the queue with a packet that can be virtually dropped. Hence, the goal of a backpressure mechanism is to minimize virtual packet drops and eliminate physical packet drops.

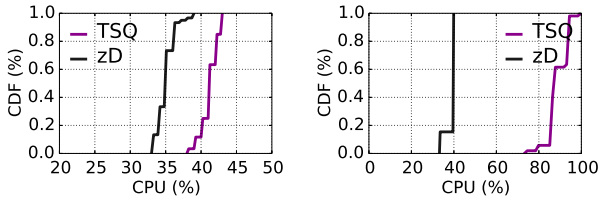
2.3 Cost of Long Queues

A naive approach to avoid loss in queues is to increase the queue size. Increasing the queue size exhibits fundamental limitation in accommodating the increasing number of concurrent connections, despite TCP Small Queue which attempts to combat bufferbloat [12]. To highlight these limitations, we conduct a simple experiment within a VM, running a large number of TCP connections using different lengths for the queue used in the VM Qdisc. In particular, we use `neper` [2] to generate 4000 TCP flows. The flows run in a VM. Queue accumulation only happens in the guest by setting a large rate for the VM in a queue not contended by any other VMs. We use the `pfifo` Qdisc [19] in the guest kernel with different queue lengths, aiming at examining behavior in two cases: 1) TSQ operation point where no packets are dropped (i.e., 2 packets are enqueued per flow), leading to a queue length of 8k slots, and 2) a queue length of 1k slots representing queue sizes that avoid bufferbloat. We also compare using the two cases to zD to highlight potential improvements. More details about our experimental setup is presented in Section 6.1.



(a) CDF of RTT (b) Zoomed in CDF of RTT

Figure 3: Bufferbloat, when pfifo queue size is 8k slots, leads to two orders of magnitude degradation in RTT. High contention and virtual packet drop rates, when pfifio queue size is 1k slots, leads to an order of magnitude degradation in tail latency compared to zD.



(a) CPU usage in VM (b) CPU usage of vhost

Figure 4: zD reduces CPU usage in both VM and the physical machine compared to standard kernel implementation for TSQ (pfifo).

We find that longer queue length leads to longer RTT, implying that relying on TSQ leads to performance degradation as the number of flows grows. Figure 3a compares the RTT of TCP flows with a pfifo queue with two queue length values. The result shows that excess buffering in a long queue increases latency as well as causing packet delay variation (long tail in Figure 3b). We also repeat the experiment with Fair Queue (FQ) Qdisc [13] and observes that FQ has similar RTT as pfifo for a queue size of 8k slots. This behavior occurs despite FQ attempting to reduce the variance in RTT using round robin scheduling of active flows.

2.4 Cost of Packet Drops

We characterize the cost of packet drops in terms of both CPU utilization as well as tail latency. We find that both metrics are interconnected with a negative feedback loop where high CPU cost leads to high tail latency, which in turns increases the CPU cost further. In this section, we explain in detail the causes of this peculiar behavior. We examine packet queues in the same setting as the previous section (i.e., flows started inside a VM). This allows us to examine CPU cost inside the stacks of both guest and host kernels. To illustrate these costs, we contrast the performance of the standard Linux implementation to our proposed system zD, which does not suffer from the same issues. We use zD simply to illustrate the inefficiency of the current approach used in the Linux kernel, explaining its details in subsequent sections.

CPU Cost: The CPU cost of packet queuing in the guest kernel is caused by the contention between TCP flows competing to acquire Qdisc lock and fill its limited space. This CPU overhead is a well documented issue [26, 28]. This overhead is exacerbated in cases where virtual or physical packet drops occur. In particular, a flow competes to acquire a lock to the Qdisc only to have it dropped, forcing the flow to try to acquire the lock again for the same packet. This overhead is shown in Figure 4a. The CPU cost in the host kernel is similar to that of the guest kernel in terms of contention to acquire Qdisc lock between multiple VMs. Furthermore, the hypervisor runs a `vhost` thread per `vring` to process traffic generated by the VM. The CPU utilization of `vhost-net` threads grows as the number of packets generated by a VM grows. In our experiments, we have a single `vring` per VM. We find that avoiding packet drops and contention also reduces CPU cost of the `vhost-net` thread (Figure 4b), recorded by pinning the thread to a specific core and measuring the utilization of that core. In order to explain this behavior, we first examine the cost of packet drops on tail latency.

Latency Cost: Packet drops, in addition to time wasted on lock contention, cause delays to packet transmission. In particular, a packet has to successfully acquire the lock to the queue, and find room in the queue, in order to be transmitted. Otherwise, the packet is dropped, either physically or virtually, and forced to reattempt the process. This is clear in comparing zD, which avoids the mentioned overhead, and standard kernel implementation with 1k slots, shown in Figure 3. In particular, the impact of bufferbloat explains the behavior of the case when a queue size of 8k is used. However, the improvements in tail latency provided by zD compared to standard kernel implementation with 1k slots are explained by reducing contention as well as avoiding packet drops.

Impact of RTT tail performance on `vhost-net` CPU: This strange interaction is an artifact of years of optimization of the TCP stack yielding unexpected scenarios. These optimizations are summarized in [10]. We note that all optimizations we mention here are enabled by default in the Linux kernel stack. They start with TCP Segmentation Offload (TSO), a mechanism to achieve low CPU utilization at high networking speed by offloading TCP segmentation to hardware. However, TSO, with fixed segment size, may lead to microbursts for flows with low rate, which is not desirable in networks relying on merchant silicon switches with short buffers. Here lies a tradeoff between CPU and network performance; relying on large fixed segment size saves host CPU but results in a bursty network and using small segment sizes increases CPU cost, through processing of more packets, but yields better network performance. The current approach used in Linux attempts a compromise by automatically determining the size of TSO segments based on the transmission rate.

TSO autosizing was introduced to decide the size of data in a burst [13]. The goal of TSO autosizing is regulating the number of packets transmitted by any single TCP flow by changing the TSO size, and consequently reducing the burst size of the TCP flow. In particular, TSO autosizing aims at making TCP flows send a packet every millisecond rather than a hundred packets every 100 milliseconds. The algorithm calculating TSO size relies on an estimate of the rate of the flow calculated as $2 \times cwnd/RTT$, where $cwnd$ is the congestion window size and RTT is a moving average of the measured RTT value in the kernel. This means that a long tailed RTT distribution leads to a smaller pacing rate, which means the data will be chunked into smaller size. This leads to higher CPU cost at the `vhost-net` thread, as shown in Figure 4b. A CDF

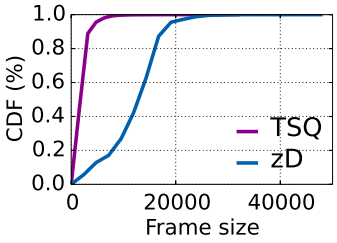


Figure 5: CDF of frame size, showing the impact of tail RTT performance on the behavior of TSO autosizing algorithm. Larger tail latency yields smaller packets, causing higher CPU cost.

of packet sizes under standard kernel implementation and zD is shown in Figure 5, where the difference between packet sizes can be explained by the difference in tail RTT shown in Figure 3. Note that when the CPU utilization of the core handling the `vhost-net` reaches 100%, the latency faced by packet can increase, further impacting packet sizes, leading to a negative feedback loop of bad performance.

2.5 Related Work

Egress path congestion: TCP Small Queue (TSQ) [12] is the most prominent backpressure mechanism in practice today, which we have already thoroughly discussed. TSQ relies on signaling within the kernel stack to maintain the per-flow limit. More recent proposals extend TSQ signalling to enforce the per-flow limit to queues beyond the kernel stack. Carousel employs delayed delivery of completion signal from the NIC to the TCP stack to apply backpressure from a userspace network processor to the kernel TCP stack [28]. While traditional completion is implemented as a signal from driver to transport stack in the same order of packets arriving at the NIC, asking the transport stack to send more packets, Carousel implements out-of-order completions and relies on TSQ to limit the number of packets per flow. PicNIC [18] extends TSQ signalling to allow backpressure from a hypervisor to traffic sources inside a VM. It also proposes a per VM budget of packets, for cases when a VM doesn't support the backpressure signal. Note that Carousel and PicNIC exhibit the inherent TSQ issues discussed earlier as queues have to accommodate $O(N)$ packets for N flows. *zD removes that relationship between the number of active flows and queue size. In particular, zD allows for setting a small queue size, handling a large number of flows without causing any packet drops.* End host queue buildup can be handed in a similar manner to in-network queue buildup through congestion control algorithms [16]. This approach does not eliminate packet drops but helps improve tail latency.

Queue overflow is not the only cause of congestion on the egress path. Another cause of congestion is exhausting CPU resources. Several systems proposed improve the CPU efficiency of queuing in the network stack, thus allowing it to handle more packets and flows. SENIC [26] improves rate limiting scalability by allowing for software queues to make use of hardware to improve rate limiting performance. Carousel [28] employs a time-based marking of packets and the timing wheel data structure to improve the performance of software-only rate limiting. Eiffel [29] presents a software only solution for general purpose packet scheduling. Several proposals explore improving efficiency of scheduling algorithms by offloading them to hardware [30, 31].

Ingress path congestion: In this paper, we focus on performing backpressure on egress traffic at the end host. Recently several proposals have looked at congestion control of the ingress path, implementing scalable networking stacks [22, 25] and enforcing isolation between receiving flows [14, 18]. Ingress path congestion at the end host occurs when one receiver (e.g., VM or socket) receives packets at a high rate so that it overwhelms the CPU at the receiver. Congestion control of ingress traffic typically requires fine grain CPU scheduling to allocate enough resources to process incoming packets for all receivers. Congestion can also happen due to incast scenarios when ingress traffic demand exceeds the NIC capacity at receiver. Resolving incast issues in datacenter networks has been an active area of congestion control research [5, 20, 21, 35].

3 zD DESIGN PRINCIPLES

Packet drops are caused by demand exceeding capacity. This means that traffic sources will get less bandwidth than their demand. The only solution to this problem is to change capacity or demand. However, congestion control aims at optimizing reaction to such scenarios. Hence, the overarching goal of zD is to change the indicator of congestion at end hosts from packet drops, and to consequently achieve less throughput than demand, lowering throughput without drops. This avoids sending an ambiguous signal, that does not differentiate between end-host drops and in-network drops. It also allows for better CPU and network performance as discussed earlier. This high level goal has to be achieved in tandem with the following objectives:

- **Prevent drops due to scheduled buffer overflows:** This is the main objective of zD. As discussed in the previous section, packet drops lead to poor network and CPU performance. zD allows overflowing queues to apply backpressure to traffic sources to prevent them from enqueuing more packets.
- **Maintain CPU efficiency:** Preventing drops can lead to cases where the traffic sources are constantly busy polling on available slots in the queue. This behavior trades CPU efficiency for network efficiency. zD avoids this type of behavior.
- **Maintain consistency with packet scheduling policies:** Backpressure is a form of controlling access to a congested scheduled buffer. zD should avoid scenarios where its coordination of access to the queue conflicts with the scheduling algorithm performed by the queue itself. An example of such conflict is an overflowing queue that has room for low priority traffic and no room for high priority traffic. zD ensures that only high priority packets get enqueued by applying backpressure to flows in a way corresponding to the scheduling algorithm of the queue which can be configured when the scheduled buffer is configured.

We find that these objectives can be achieved through a structuring of the queuing architecture at end hosts that implements the following mechanisms (Figure 6):

1. Source Buffer Regulator (§4.1): The source buffer should keep a copy of packets still being processed by the networking stack until it is fully transmitted. The source buffer should also support an interface that allows the underlying stack to pause and resume

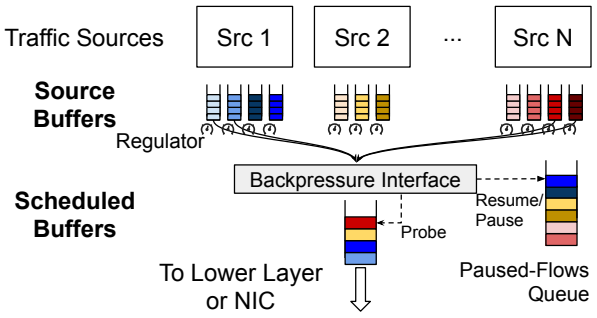


Figure 6: Schematic of zD architecture at end hosts.

transmission of packets from that buffer (e.g., TSQ). This augmentation of source buffers allows for avoiding physical packet drops by always retaining a copy of dispatched packets till they are actually consumed. It also provides an interface for the backpressure mechanism to pause and resume packet dispatch.

2. Backpressure Interface (§4.2): To eliminate physical drops, packet queues should be able to pause senders when they are full. Furthermore, senders should be able to probe for room in the queues. Such interaction between the packet queues and senders should be well defined through a backpressure interface. Furthermore, it should be CPU efficient, to avoid CPU being the bottleneck of the networking stack.

3. Paused-Flows Queue (§4.3): Backpressure should be applied in a way that does not change the intended scheduling behavior of the packet queue. Hence, zD schedules access to the packet queue by keeping paused flows in a queue that is sorted in a way consistent with that of the underlying packet scheduling policy.

The design of zD and TSQ share the regulator. Both zD and TSQ employ a mechanism that pauses and resumes source buffers. The difference between zD and TSQ lies in how the pause and resume decisions are made. In the case of TSQ, pause and resume decisions are made by the source buffer. TSQ forces source buffers to maintain a maximum of two dispatched packets per flow, leading to queue occupancy that grows as the number of active flows grow. This limits the effectiveness of TSQ backpressure in handling bufferbloat as it ignores the occupancy of the scheduled buffer. Furthermore, access to the scheduled buffer becomes dependent on CPU scheduling of sender buffers and their ability to gain the lock to the scheduled buffer. In the previous section, we show that these limitations in TSQ lead to significant performance degradation. zD mitigates these problems by extending the regulator as well as providing a Backpressure Interface and a Paused-Flows Queue.

4 zD OVERVIEW

zD applies backpressure from packet queues, which can overflow and drop packets, to source buffers from which packets are dispatched. It provides a layer between the sender buffer and the scheduled buffer. Instead of continuously pushing packets into the scheduled buffer only to drop them when the queue is full, zD adds a set of additional steps in the path of a packet. First, a copy of the packet is created to avoid physical packet drops. Then, the packet copy is used to probe the packet queue to check if it has room, and proceeds normally if the packet queue has empty slots. However, if the packet queue has no empty slots, the packet copy is dropped, causing only a virtual packet drop. This is used as a backpressure signal to the source buffer of that packet. The backpressure signal pauses the backpressued flow and registers it with zD so that it can be resumed when there is room for its packets. Figure 6 summarizes modifications to the current queue architecture. The zD logic is summarized in Algorithm 1. For the rest of this section, we elaborate on each step described in this algorithm.

Algorithm 1 zD Flow Algorithm

```
1:  procedure ProcessFlow(Flow F, Packet p, Queue q)
2:      if F.pause then return //Regulator
3:      if !q.probe() then //Backpressure Interface
4:          F.pause ← true //Regulate
5:          PausedFlowsQ.append(F)
6:      else
7:          if !F.sendTwo() or PausedFlowsQ.empty() then
8:              enqueue(F, p)
9:          else
10:             F.pause ← true
11:             PausedFlowsQ.append(F)
12: procedure ResumeFlow(Flow F)
13:     F ← GQ.popFront()
14:     F.pause ← false
15:     F.resume()
```

zD mechanisms can be applied to multiple settings where there are source buffers and scheduled buffers. In this paper, we focus on two such settings: 1) the TCP/IP kernel stack, where TCP buffers are the source buffers and Qdisc is the scheduled buffer, and 2) the hypervisor networking stack in the kernel, where the vrings of the VM are the source buffer and the Qdisc is the scheduled buffer. The details of our implementation of zD in these two settings are presented in Section 5.

Memory overhead: zD has no data-plane memory overhead except for the packets copies used to probe scheduled queue occupancy. Such packets copies are only copies of packet descriptor which are commonly used for different purposes in networking stacks. In our Linux implementation, we use one of the copies already created by the kernel's stack, incurring no exta memory overhead. Backpressure keeps data in the application buffer thus preventing the creation of new packets. The control plane overhead of zD is limited to the Paused-Flows Queue that keeps a per-flow descriptor. In our Linux implementation in a 64 bit machine, with 20k flows, the memory overhead is less than 160KB.

4.1 Source Buffer Regulator

This module has two functions: 1) define pause/resume operations, and 2) keep a copy of the dispatched packet until its transmission to the wire is confirmed. A flow can have two states "Active" and "Paused". The reaction of the stack to each state, and consequently the implementation of pause/resume functions depend on whether the stack is push-based or pull-based. In cases of a push-based stack (e.g., TCP/IP kernel stack), marking a flow as "Paused" implies that no further packets are pushed by that flow. New packets generated by the application are queued in the source buffer. Once the flow is

resumed (i.e., marked as "Active"), packets residing in the source buffer are pushed to the lower layer. On the other hand, a pull-based stack already has to sleep when it has no packet to process. We follow a similar approach by forcing the pull-based stack to sleep when it has no active flows. Note that a busy-polling stack on a dedicated core (e.g., DPDK) does not need to sleep, making the implementation of these functions a simple marking operation.

Like TSQ, zD keeps the number of packets enqueued by a single flow to a maximum of two packets. Limiting the number of packets per flow is necessary to avoid head of line blocking, where a single flow enqueues a large number of packets in the queue, slowing other flows. We found that further limiting to a single packet per flow causes performance degradation. In particular, there can be a delay between a flow becoming active and the processing of its packet. In the case of push-based model, this delay is caused by the multi-threaded nature of a push-based stack, where marking a flow as "Active" does lead to the immediate dispatch of packet by that flow. Typically, once a flow is marked as active a thread is started to kick-start packet dispatch for that flow. This approach has a processing delay associated with delaying the dispatch of packets. In the case of a pull-based stack, marking a flow as "Active" might happen during a sleep cycle. zD amortizes this delay over multiple packets by making sure that an active flow has two packets pushed to the scheduled buffer before it is paused again. Note that when a flow becomes active, it has to check the number of its packets still in the scheduled queue and make sure that it never exceeds two packets. We found that this approach, and specifically limiting the number of packets to only two, provides a good compromise between amortizing the cost of pause/resume operations and unfairness (i.e., less than two packets leads to under utilization and more than two pakcets leads to head of line blocking and unfairness).

Unlike TSQ, the sender buffer regulator can pause a flow that does not have less than two packets in the scheduled buffer. This is critical in order to decouple the queue length from the number of flows, avoiding bufferbloat scenarios in cases where there is large number of flows.

4.2 Backpressure Interface

This interface defines the interaction between source buffers and the scheduled buffer. In particular, it defines three operations: `probe`, `pause_flow`, and `resume_flow`. `probe` informs the sender buffer on whether it can push packets to the scheduled buffer. Scheduled buffers with different scheduling policy should have different implementation of `probe` function. For example, with the simplest First-in-first-out (FIFO) queue, the `probe` function returns false when the number of packets in the queue is equal to or larger than the queue capacity and returns true otherwise. For more complicated scheduling policies such as fair queue, the `probe` function needs to classify the flow first and then checks whether the flow exceeds its assigned share of the scheduled buffer.

If `probe` returns false, implying no room for that flow in the scheduled queue, `pause_flow` is invoked. `pause_flow` marks the flow as "Paused" triggering the logic of the sender buffer regulator. It also adds the flow to the Paused-flows queue. When a scheduled buffer has room (i.e., a packet is transmitted), `resume_flow` is invoked. `resume_flow` fetches the highest ranked flow in the Paused-flows queue. Then, marks it as "Active", trigger the resume logic of the sender buffer regulator. Note that this logic is deadlock-free.

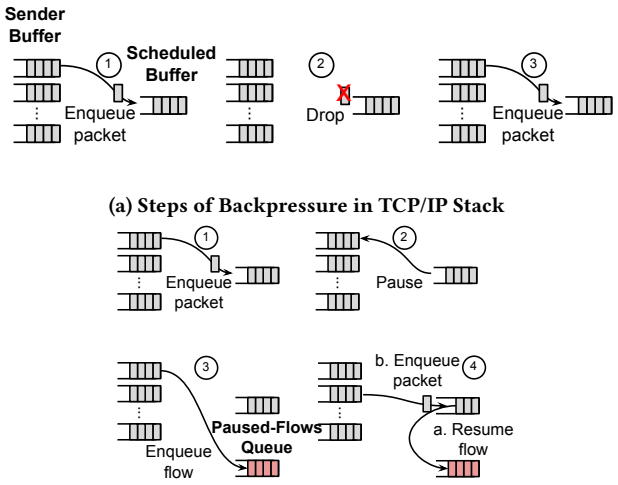


Figure 7: Illustration of different backpressure steps.

The advantage of this interface is that a flow is only active if either the scheduled buffer has room for packet or it is first attempt of that flow to access a congested scheduled buffer. This is unlike existing attempts where flows are always active causing either physical or virtual packet drops by continuously attempting to enqueue packets to the scheduled buffer. Hence, the backpressure interface improves both CPU and network performance by avoiding drops as well as only doing work when useful. The difference between the two approaches is summarized in Figure 7.

It should be noted that the granularity of the scheduled buffer decides the granularity of Backpressure Interface. For example, in our implementation of backpressure in the hypervisor, the backpressure is performed per VM because packets lose flow-level information when it passes through from the VM to the hypervisor. The Qdisc in the physical machine treats all traffic from a VM as an aggregate flow and `probe` API provides information at the granularity of VMs.

4.3 Paused-Flows Queue

The aforementioned building blocks rely the ability of zD to track paused flows. This tracking function is performed by the Paused-flows queue. The paused-flows queue is a global queue accessible to all stack threads through a global lock. The order in which flows are sorted within this global queue determines the overall scheduling policy for traffic going through the stack. zD implements a library of Paused-Flows Queuing Disciplines that correspond to the queuing disciplines implemented in the scheduled queue. The network operator has to install a Paused-Flows Queuing Discipline that corresponds to their chosen queuing discipline in the scheduled queue. This operation can be simplified by a simple network utility application. We note that the focus of our work on zD is managing congestion due to queue overflow of packets. Hence, in this work

we implement only a small library of Paused-Flows Queuing Disciplines (i.e., FIFO and rate limiting disciplines). Complex queuing disciplines can be implemented to extend the functionality of zD. Efficient implementation of such disciplines is critical to avoid congestion due to high CPU utilization. Such efficient implementation is feasible relying on building blocks proposed in our earlier work on efficient per-flow scheduling [29].

5 IMPLEMENTATION

We implement backpressure in two places: (1) the Linux TCP/IP stack, and (2) the vHost stack in the Linux hypervisor stack. Our implementation is based on Linux kernel 4.14.67, however it is not restricted to that specific version. While the zD design described in Section 4 can be generally applied to both cases, we focus on these two settings as discussed earlier.

5.1 TCP/IP Stack Implementation

Implementing zD requires modifying the way that the TCP stack interacts with the IP stack. We start by giving an overview of the transmission path (Tx path) of the standard TCP/IP stack implementation in the kernel. In the TCP Tx path, data from the userspace application is pushed into the Socket Buffer (`skb`) and all paths of function calls end up calling `tcp_write_xmit` function regardless of whether the TCP socket is sending a packet for the first time or is retransmitting a packet. In the `tcp_transmit_skb` function, each `skb` is cloned so that TCP can always keep a copy of the original data until the packet is ACKed by the receiver. The `tcp_transmit_skb` function calls `dev_xmit_skb`, which tries to queue the packet into the corresponding Qdisc (i.e., the scheduled buffer). If the Qdisc queue is full, the `skb` will be virtually dropped. In particular, the pointer to the next packet to send, `sk_send_head`, will not be advanced.

Under the standard kernel implementation, when an `skb` is virtually dropped, TCP will attempt to resend it immediately unless the socket is throttled by TSQ. TSQ reduces the number of TCP packets in the Tx path by limiting the amount of memory allocated to the socket, forcing `sk->sk_wmem_alloc` to not grow above a given limit. By default, if a socket already has two TSO packets in flight, the socket will be throttled until at least one of the packets is freed. Note that TSQ can be viewed as sender buffer regulator. A socket paused by TSQ will be resumed by a callback function when a skb is free (i.e., when `skb_free` function is executed), with the assumption that if an skb is destroyed, an extra space in the queue is available. This approach means that when a slot in the queue is freed, its replacement is notified. This implies that the approach of reattempting to send a dropped `skb` immediately can only make congestion at the Qdisc worse. Our implementation is shown in Figure 8, where the yellow blocks show function calls we modified.

Probe: Before `dev_xmit_skb` function pushes the packet into the queue according to the queueing discipline, it checks whether the packet should be passed to the next scheduled buffer through our extended `probe` API. We implement `probe` for the three most basic scheduling algorithms: `pfifo_fast` as the default qdisc for Linux interfaces, classful multiqueue (mq) for multiqueue devices, and token bucket filter (TBF) as a traffic shaper.

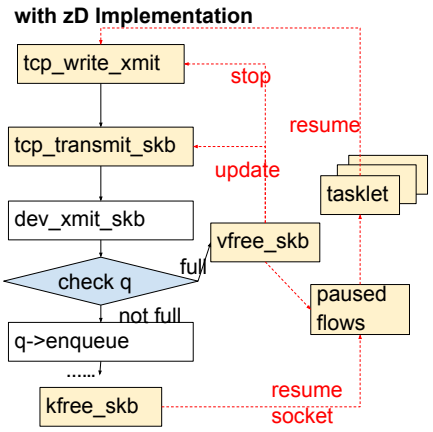


Figure 8: Flow chart describing TCP/IP stack with zD

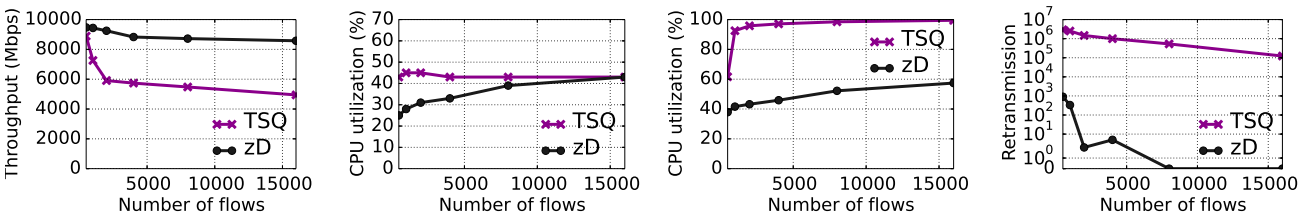
Pause: If the `probe` returns false, instead of resuming the socket, we mark the socket as stopped and place a pointer of the socket into a global queue shared by all sockets. Access to the global list is serialized through a global lock.

Resume: After an `skb` is consumed by the driver, the global list dequeues a socket and marks the socket as nonstop. To ensure the socket is resumed immediately, we use a tasklet to schedule the retransmission operation as soon as the CPU allows. We use a tasklet as a per-CPU variable for performance considerations. As indicated earlier, the existing TSQ interface for handling flow pause and resume is not very helpful for zD. In particular, TSQ relies on the `sk_wmem_alloc` field of `struct sock` to make decision on throttling the socket. However, our implementation keeps increasing the value `sk_wmem_alloc` until `has_room` returns true. Hence, TSQ cannot properly decide whether the flow should be throttled. Therefore, we disable TSQ and implement our flow activation algorithm discussed in the previous section.

5.2 Hypervisor Implementation

We implement zD in the hypervisor based on the zero-copy `virtio` Tx path. Zero-copy transmit is effective in transmitting large packets between a guest VM to an external network without affecting throughput, consuming lower CPU and introducing less latency [11]. The `vring`, where `virtio` buffers packets, is a set of single-producer, single-consumer ring structures that share scatter-gather I/O between the physical machine and the guest VM. `vring` keeps track of two indexes: `upend_idx` and `done_idx`. The indexes represent the last used index for outstanding DMA zerocopy buffers in the `vring` and the first used index for DMA done zerocopy buffers, respectively. The `vhost` thread pulls packets from the `vring` and attempts to enqueue them to the Qdisc.

When a process transmits data, the kernel must format the data into kernel space buffers. Zero-copy mode allows the physical driver to get the external buffer to directly access memory from the guest `virtio-net` driver, hence reducing the number of data copies that require CPU involvement. In the hypervisor, the `vhost` process passes the userspace buffers to the kernel stack `skb` by pinning the guest VM user space and allowing direct memory access (DMA)



(a) Throughput (b) VM CPU (c) vHost CPU (d) TCP Retransmission

Figure 9: 10Gbps network speed with a qdisc of 100 slots in the hypervisor

for the physical NIC driver. The path of the skb in the hypervisor is shown in Figure 2. The Tap socket associated with the vhost process sends out the packet through the Tap device. Packets are then received by the virtual bridge, and the packet is passed to Qdisc. Finally, the packet is consumed by the physical NIC. Note that when vhost pulls a packet from the vring, once the packet is processed, the kfree_skb callback function will inform the vring to destroy the packet, whether it was actually transmitted or dropped by Qdisc.

The **Probe** and **Resume** steps are implemented in this setting in a very similar fashion to that of the TCP/IP stack. Implementation of **Pause** requires handling some corner cases. In particular, if the Qdisc is full, instead of calling the kfree_skb function to free the packet and mark the DMA as done, we mark the corresponding VM as paused, stops polling from its vring. This step also requires moving the upend_idx back to point to the position of the dropped packet. A significant difference between the hypervisor setting and the TCP/IP stack setting is the potential existence of further packets in-flight from the VM that have been pulled from the vring before the VM was marked as paused. The situation is further complicated as those packets can reach the Qdisc and find that it now has room. This behavior can lead to introduction of out-of-order packet delivery which can lead to TCP performance degradation. Hence, all in-flight packets between the vring and the Qdisc are dropped to avoid such scenarios. Note that moving the upend_idx makes sure that those packets are retransmitted later. We implement a callback function to resume polling from the vring when a packet is passed to the physical NIC driver.

6 EVALUATION

6.1 Experiments Setup

We conduct experiments between two Intel Xeon CPU E5-1620 machines, connected with a 10Gbps link. Both machines have four cores, with CPU frequency fixed to 3.6GHz. We generate traffic with neper [2], a network performance measurement tool that can generate thousands of TCP flows. The TCP flows are generated inside a virtual machine and are sent to a remote machine. We use Qemu with KVM enabled as the hypervisor. For a baseline, both VM and physical machines run Alpine Linux with kernel version 4.14.67. We run a modified version of that kernel with zD implementation. In our experiments, we ran into a known issue of vhost where the Rx path of a VM becomes bottlenecked on the Tx path, because both are handled with the same thread [32]. The issue is inherent in the current implementation of virtualization in the Linux kernel, affecting baselines and zD. The bottleneck is resolved by allocating more CPU to the receiving path or improving the receive path architecture [22, 25]. Hence, we perform our experiments in two settings, one with 6 vCPUs assigned to the virtual machine (experimenting with a bottleneck-free end host) and another with 1 vCPU assigned to the virtual machine (exposing the Rx path bottleneck to evaluate zD under a resource constrained end host). In the first setting, we tune CPU affinity to assign 5 cores for the Rx path. None of the six cores hit 100% thus eliminating the issue. The second setting can still face that issue, however, we find that zD alleviates pressure on the Tx path, making the performance of the Rx path the main bottleneck.

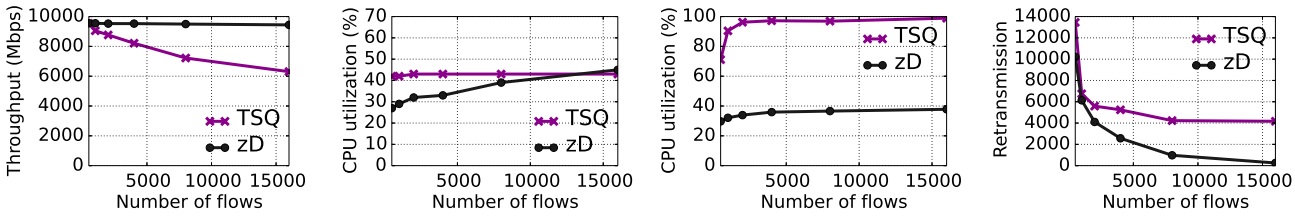
The default Tx queue length is set to 1000 in both the VM and the hypervisor[3]. Experiments are run for 60 seconds each. Our primary metrics are aggregate throughput of all flows, CPU utilization inside the VM, vhost CPU utilization for its pinned core, TCP retransmissions, and RTT. We track CPU utilization in the virtual machine using dstat and track CPU utilization of the vhost process in the physical machine using top. CPU utilization is recorded every second. We track the number of TCP retransmissions using netstat. In all experiments, machines are running only the applications mentioned here making any CPU performance measurements correspond with network overhead.

6.2 Overall Performance

We start by reporting the overall performance of zD in a setting where packet drops can occur in the VM and the hypervisor. These experiments represent the general case of modern cloud infrastructure. In particular, we consider three cases: 1) a high bandwidth VM with a short queue in the hypervisor, where we allocate the whole 10 Gbps to the VM but configure a short queue of 100 slots[4], 2) a high bandwidth VM with a long queue in the hypervisor, where we allocate the whole 10Gbps to the VM and configure a queue of 1000 slots, and 3) a low bandwidth VM with a long queue, where the hypervisor forces a 1 Gbps limit on the VM in a queue with 1k slots. In the high bandwidth VM setting, we use a pfifo Qdisc in the physical machine. In the low bandwidth VM setting we use tbf to perform rate limiting in the hypervisor. We use the default queue size 1000 for the qdisc inside the VM. The first setting represents strict performance requirements (i.e., small processing budget per packet and high probability of packet drop, as shown in recent work [18]), while the second and third represent the more general case.

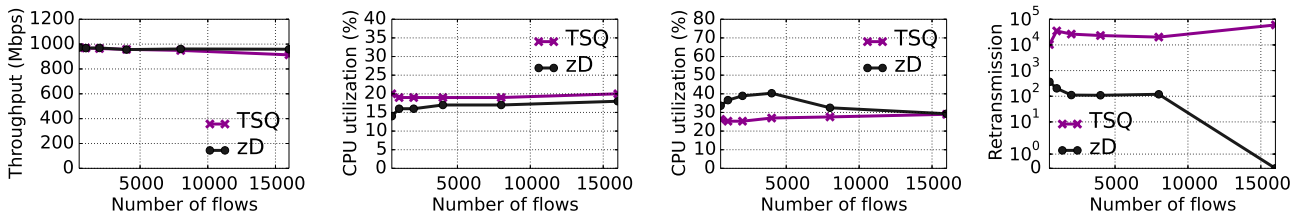
[3]Earlier work with larger scale experiments used a queue length of 4000. Note that a small queue length is also critical to avoid bufferbloat.

[4]We choose a small queue length to force congestion in the hypervisor. This emulates production scenarios where queue lengths are larger but the number of VMs per end host will also be much larger, making the effective queue length per VM small.



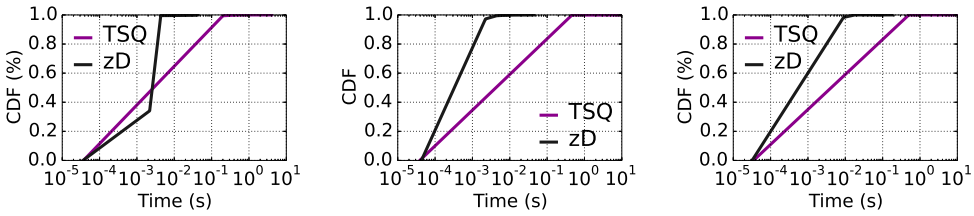
(a) Throughput (b) VM CPU (c) vHost CPU (d) TCP Retransmission

Figure 10: 10Gbps network speed with a qdisc of 1000 slots in the hypervisor



(a) Throughput (b) VM CPU (c) vHost CPU (d) TCP Retransmission

Figure 11: 1Gbps network speed with a qdisc of 1000 slots in the hypervisor



(a) 10G network with 100 qlen (b) 10G network with 1000 qlen (c) 1G network with 1000 qlen

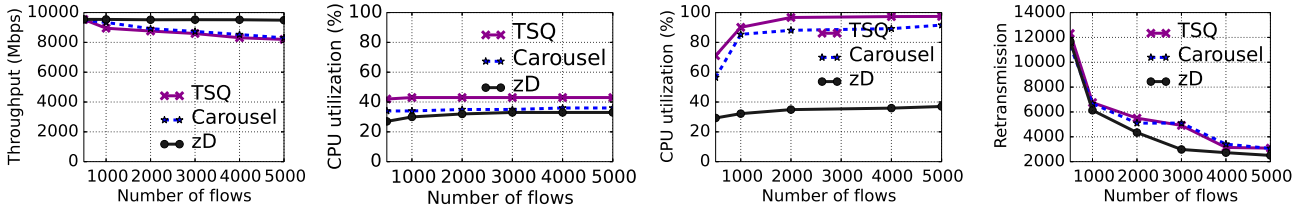
Figure 12: zD reduces the tail of RTT by 100x with both 10G network and 1G network

We vary the number of flows from 500 to 16k and measure throughput, CPU utilization inside the VM, vhost CPU utilization and TCP retransmissions. For both settings we measure the RTT at 4k flows. We focus on cases with a single VM to be able to better analyze the results. We further restrict the settings in the following sections, to explain the value of individual zD mechanisms. We allocate 6 vCPUs to the VM to avoid having the Rx path being the bottleneck. VM with less vCPUs will be evaluated in the micro-benchmark section.

Figure 9 shows the performance of the standard kernel implementation and zD for the first setting. zD performs better in terms of all metrics. In particular, zD achieves around 50% improvement on the aggregate throughput when there are more than 4k flows (Figure 9a). Such improvements in throughput come from the elimination of the vhost CPU utilization as the bottleneck (Figure 9c). zD saves between 40% to 50% of the thread utilization of its CPU core, which is 100% utilized in the standard implementation, making it the performance bottleneck and leading to 50% loss in network throughput. Furthermore, zD reduces tail latency by 80x from 4s to 0.05s (Figure 12) which is mostly due to reduction of TCP retransmissions by 1000x (Figure 9d). There is a slight degradation in median latency but such slight degradation is generally tolerable to significantly reduce the tail latency [6]. Note that in this scenario zD is lightweight as at low loads it consumes less CPU and achieves better network performance, compared to the standard kernel implementation.

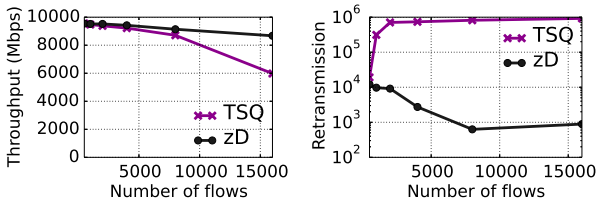
Figure 10 shows the results for the second case. Compared with the first setting, TSQ achieves higher throughput and less retransmission because of fewer drops on the hypervisor qdisc. But still, zD achieves higher throughput, lower VM CPU usage, lower vHost CPU usage, and fewer TCP retransmissions. We observe there is less than 100 packet drops in the hypervisor qdisc so the improvement mainly comes from the advantages of using zD in the VM. The zD vHost CPU usage is lower than that of the standard (TSQ) kernel when the number of flows is smaller than 16K. When there are 16K flows, zD has higher vhost CPU usage because it pushes much more traffic than the standard kernel. The tail latency is significantly reduced from 8s to 0.05s (Figure 12).

Figure 11 shows the results for the third setting. zD again improves all network metrics. In particular, zD improves throughput by up to 5% (Figure 11a) and reduces retransmissions by 1000x (Figure 11d). Most notably, zD reduces tail latency by 45x from 9s to 0.2s (Figure 12). zD also reduces VM CPU utilization by 15%. However, zD incurs higher vhost CPU cost by up to 40%. The higher vHost CPU usage results from the extra work of vhost trying to resend the packets dropped in the physical machine Qdisc instead



(a) Throughput (b) VM CPU (c) vHost CPU (d) TCP Retransmission

Figure 13: Compared with Carousel, zD achieves higher throughput, lower VM CPU usage, lower vHost CPU usage, and fewer TCP retransmissions



(a) Throughput (b) TCP retransmission

Figure 14: Compared with TSQ, zD achieves higher throughputand and fewer TCP retransmissions when 1 vCPU is assigned to the VM.

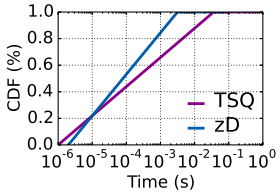


Figure 15: CDF of flow RTT with hypervisor-only zD.

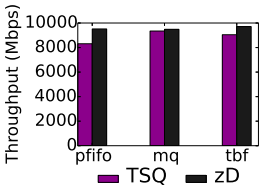


Figure 16: Throughput under different Qdiscs

	Drop in Qdisc	TCP retransmission
TSQ	1.6×10^5	1×10^5
zD	562	110

Table 1: Drops and retransmission in case of hypervisor-only zD implementation.

of relying on the TCP socket in the VM to retransmit the packets. This shows a tradeoff between network performance and VM CPU on one side and hypervisor CPU on the other side. We also envision that userspace stacks can amortize the cost in the hypervisor due to their busy polling nature [1, 3].

Comparison with Carousel: We use Carousel as a baseline to examine if the combination of efficient queuing data structure and TSQ-like backpressure can improve on the performance of standard Linux Qdiscs. We implemented Carousel in Linux Qdisc using a more efficient integer priority queue data structure [29] and compared zD with carousel in the 10G network setting with a queue of 1000 slots in the hypervisor. Figure 13 shows that zD outperforms Carousel in all metrics. While Carousel achieves higher CPU efficiency and a higher throughput compared with TSQ, it does not fundamentally solve the problem when queue runs out of space with a large number of flows. As discussed earlier, Carousel relies on TSQ-like backpressure to limit the number of packets per flow, which works reasonably well with a small number of flows. Unfortunately, with a large number flows, limiting two packets per flow can still overflow the queue, leading to performance degradation.

6.3 Microbenchmark

zD with VM-only bottleneck: In the previous section, we looked at the general case where drops happen in both the VM and the hypervisor. In this section, we look at cases where there is a single bottleneck. We focus on the case where drops happen at the VM because it is easier to test it at large scale (i.e., large number of flows) compared to the hypervisor which requires scaling to a large number of VMs. We prevent drops in the hypervisor by a long unscheduled queue (i.e., `pfifo` with 1k slots). Note that this setting is convenient and allows for a better understanding of the performance of zD because adding more VMs causes drops in the hypervisor, which results in a similar scenario as the one we studied earlier.

We start by looking at the case where the VM is allocated 6 vCPUs, thus eliminating the Rx path bottleneck. The result is similar to what we show in Figure 10. When we use a queue of 1000 slots in the hypervisor, regardless whether zD is implemented in the hypervisor or not, the performance is similar because the hypervisor queue is not easily congested due to the high-speed NIC and the low latency of the networking stack.

To highlight the value of zD, we rerun the experiment in the setting where 1 vCPU is assigned to the virtual machine. zD's value is clear in its impact on throughput as shown in Figure 14a. In particular, zD can maintain 43% higher throughput at 16k flows. This significant improvement is mostly due to reduction in retransmission rate (Figure 14b). We find that zD and the standard kernel exhibit similar CPU performance for both the VM and the `vhost` thread when the number of flow is larger than 2k. Both systems have 100% VM CPU utilization and their `vhost` CPU utilization was about 49% for kernel and 43% for zD. The reason of degradation of kernel implementation moving from allocating 6 vCPUs to a single vCPU is the higher retransmission rate in the later case. The is mostly due to the Rx path congestion which leads ACK packets to be delayed. This cases TCP spurious retransmissions, where senders timeout and retransmit packets whose ACK is delayed. zD achieves lower TCP retransmission by reducing the number of interrupts in VM Tx path reducing the Rx path congestion.

zD with Hypervisor-only bottleneck: Next we quantify the benefit of zD when it is only implemented in the physical machine.

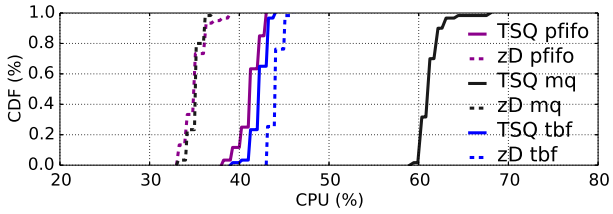


Figure 17: CPU usage in VM under different Qdiscs

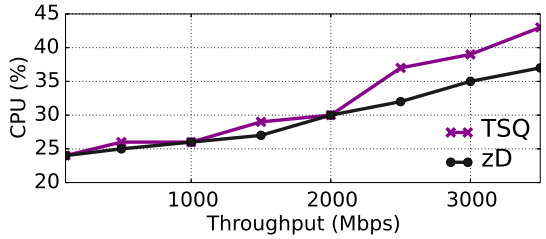


Figure 18: CPU usage in VM under light traffic load

We use `tbf` Qdisc to set a bandwidth limit of 1Gbps in the physical machine with only 100 flows. This setting forces packet drops to happen only in the hypervisor. In both settings, the flows achieve the targetted aggregate rate. However, zD improves RTT by avoiding packet drops. Figure 15 shows the CDF of flow RTT. With vanilla kernel, the 99.99^{th} percentile is large at around 0.8s. The 99.99^{th} percentile of zD is less than 0.1s. We present the number of drops in Qdisc and the number of TCP retransmission for both the standard kernel and zD in Table 1. zD reduces the TCP retransmission caused by packet drop in Qdisc by 1000x. Note that CPU utilization is low and comparable for both implementations due to the limited scale of the experiment.

Interaction with different Qdisc: We explore zD performance under different Qdiscs to show that zD can operate with different underlying policies. We implement a FIFO policy for the Paused-flows Queue, which is compatible with all queuing policies used here. However, each queuing policy requires a different implementation of the `has_room` function. We conduct our experiments for `pfifo`, `mq`, and `tbf` Qdiscs and measure the throughput and CPU usage inside the VM with 4k flows. The chosen setting is similar to the case where drops happen only at the VM.

Figure 16 shows that zD achieves around 12% throughput improvement with `pfifo` and around 8% throughput improvement with `tbf`. With `mq`, both zD and the standard kernel have similar throughput because the instances of `vhost` process scale as the number of queues increases. Figure 17 shows the CDF of CPU usage in the VM under different Qdiscs. zD reduces the CPU usage by 7% under `pfifo` and by 25% under `mq` but has slightly higher CPU usage under `tbf`. The higher CPU usage with `tbf` results from the extra work performed by zD to stop and resume the vring. Although dropping packets directly can save hypervisor CPU, the dropped packets need to be recovered by the TCP retransmission mechanism thus wasting CPU in the VM.

zD at light loads: zD achieves its goals by adding more coordination between traffic sources and packet queue, which might cause significant overhead at light loads. However, we find that this is not the case. To conduct experiments with low loads, we use 10 instances of iperf as traffic generator, each generating 100 TCP flows. We control the load by setting a rate limit in the application layer, reducing demand of individual flows. Figure 18 shows effect of varying the TCP loads on CPU usage in VM. zD has similar CPU usage as the standard kernel because there is little packet drop in Qdisc when the traffic load is light. Hence, no coordination between traffic sources and the packet queue is needed. As the traffic load increases, getting closer to an aggregate rate of 3 Gbps, the number of drops in Qdisc also increases and zD starts to outperform the standard kernel.

7 DISCUSSION

Limitations: We show that zD can improve network and CPU performance by applying backpressure from the scheduled buffer to the source buffers. Our work on zD has some modest limitations. In particular, the overhead of backpressure in the hypervisor can, in some cases, cause the `vhost` thread to consume more CPU than a standard implementation. We believe that with further engineering this overhead can be eliminated completely. A minor limitation of our evaluation approach is our focus on simple scheduling policies in the Paused-flows queue. However, we find that recent work on efficient packet schedulers in software has thoroughly handled the issue [29], allowing us to focus more on handling cases of congestion.

zD for UDP, ingress traffic, and userspace Stacks: We focus in this paper on the TCP stack in the kernel, mostly due to the ubiquity of such a setting. As QUIC gains a larger share of Internet traffic, handling backpressure on UDP flows becomes more important. Such backpressure is particularly important because UDP packet drops are physical drops, as UDP does not provide reliability. This puts more stress on the QUIC stack to recover these losses. We do not envision any significant engineering or research challenges extending zD to the UDP stack. The situation is similar for ingress traffic where drops are caused by NIC buffers overwhelming a kernel buffer. We envision that CPU overhead can be saved if zD is applied in such scenarios with minimal engineering efforts. The situation is different for userspace Network Stacks (e.g., DPDK). While we believe the building blocks of zD can be mapped to such stacks, we envision that porting it will require engineering effort.

8 CONCLUSION

Packet queuing and scheduling is a standard operation at end hosts. Congestion of scheduled queues at end hosts typically incurs packet drops which lead to high CPU cost as well as degradation in network performance. In this paper, we show that by augmenting existing architectures with three simple mechanisms, CPU and network performance can be significantly improved under high loads, improving tail latency by 100x. Our work on zD should extend the scalability of current end-host stacks and motivate revisiting the queuing architecture in other network elements.

9 ACKNOWLEDGEMENT

This work was funded in part by the National Science Foundation grant NETS 1816331.

REFERENCES

[1] 2014. Intel DPDK: Data plane development kit. https://www.dpdk.org/.
[2] 2016. neper: a Linux networking performance tool. https://github.com/google/neper.
[3] 2017. BESS: Berkeley Extensible Software Switch. https://github.com/NetSys/bess/wiki.
[4] Vikas Agarwal, MS Hrishikesh, Stephen W Keckler, and Doug Burger. 2000. Clock rate versus IPC: The end of the road for conventional microarchitectures. In *ACM SIGARCH Computer Architecture News*, Vol. 28.
[5] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2011. Data center tcp (dctcp). In *Prof. of ACM SIGCOMM '11*.
[6] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. 2012. Less is more: trading a little bandwidth for ultra-low latency in the data center. In *Prof. of USENIX NSDI '12*.
[7] W. Almesberger, J. H. Salim, and A. Kuznetsov. 1999. Differentiated services on Linux. In *Proc. of IEEE GLOBECOM '99*.
[8] Vint Cerf, Van Jacobson, Nick Weaver, and Jim Gettys. 2011. BufferBloat: What's Wrong with the Internet? *ACM Queue* 9 (2011).
[9] Yanpei Chen, Rean Griffith, Junda Liu, Randy H Katz, and Anthony D Joseph. 2009. Understanding TCP incast throughput collapse in datacenter networks. In *Proc. of the ACM workshop on Research on enterprise networking (WREN '09)*.
[10] Yuchung Cheng and Neal Cardwell. 2016. Making Linux TCP Fast. In *Netdev 1.2 Conference*.
[11] Hsiao-keng Jerry Chu. 1996. Zero-copy TCP in Solaris. In *Proc. of USENIX ATC '96*.
[12] Eric Dumazet and Jonathan Corbet. 2012. TCP small queues. https://lwn.net/Articles/507065/.
[13] Eric Dumazet and Jonathan Corbet. 2013. TSO sizing and the FQ scheduler. https://lwn.net/Articles/564978/.
[14] Michael Dalton et al. 2018. Andromeda: Performance, Isolation, and Velocity at Scale in Cloud Network Virtualization. In *Proc. of USENIX NSDI '18*.
[15] David Geer. 2005. Chip makers turn to multicore processors. *IEEE Computer* 38 (2005).
[16] Keqiang He, Weite Qin, Qiwei Zhang, Wenfei Wu, Junjie Yang, Tian Pan, Chengchen Hu, Jiao Zhang, Brent Stephens, Aditya Akella, and Ying Zhang. 2017. Low Latency Software Rate Limiters for Cloud Networks. In *Proc. of ACM APNet'17*.
[17] Vimalkumar Jeyakumar, Mohammad Alizadeh, David Mazières, Balaji Prabhakar, Albert Greenberg, and Changhoon Kim. 2013. EyeQ: Practical Network Performance Isolation at the Edge. In *Proc. of USENIX NSDI '13*.
[18] 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 *Proc. of ACM SIGCOMM '19*.
[19] Alexey N. Kuznetsov. 2002. pfifo-tc: PFIFO Qdisc. https://linux.die.net/man/8/tc-pfifo/.
[20] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. 2019. HPCC: High Precision Congestion Control. In *Proc. of ACM SIGCOMM '19*.
[21] Radhika Mittal, Vinh The Lam, 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 *Proc. of ACM SIGCOMM '15*.
[22] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *Proc. of USENIX NSDI '19*.
[23] Ben Pfaff, Justin Pettit, Keith Amidon, Martin Casado, Teemu Koponen, and Scott Shenker. 2009. Extending networking into the virtualization layer. In *Proc. of ACM HotNets-VIII*.
[24] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, Keith Amidon, and Martin Casado. 2015. The Design and Implementation of Open vSwitch. In *Proc. of USENIX NSDI '15*.
[25] George Prekas, Marios Kogias, and Edouard Bugnion. 2017. ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. In *Proc. of ACM SOSP '17*.
[26] Sivasankar Radhakrishnan, Yilong Geng, Vimalkumar Jeyakumar, Abdul Kabbani, George Porter, and Amin Vahdat. 2014. SENIC: Scalable NIC for End-Host Rate Limiting. In *Proc. of USENIX NSDI '14*.
[27] Rusty Russell. 2008. virtio: towards a de-facto standard for virtual I/O devices. *ACM SIGOPS Operating Systems Review* 42 (2008).
[28] Ahmed Saeed, Nandita Dukkipati, Valas Valancius, Terry Lam, Carlo Contavalli, and Amin Vahdat. 2017. Carousel: Scalable Traffic Shaping at End-Hosts. In *Proc. of ACM SIGCOMM '17*.
[29] Ahmed Saeed, Yimeng Zhao, Nandita Dukkipati, Ellen Zegura, Mostafa Ammar, Khaled Harras, and Amin Vahdat. 2019. Eiffel: Efficient and Flexible Software Packet Scheduling. In *Proc. of USENIX NSDI '19*.
[30] Vishal Shrivastav. 2019. Fast, Scalable, and Programmable Packet Scheduler in Hardware. In *Proc. of ACM SIGCOMM '19*.
[31] Brent Stephens, Aditya Akella, and Michael Swift. 2019. Loom: Flexible and Efficient NIC Packet Scheduling. In *Prof. of USENIX NSDI '19*.
[32] Jianfeng Tan, Cunming Liang, Huawei Xie, Qian Xu, Jiayu Hu, Heqing Zhu, and Yuanhan Liu. 2017. VIRTIO-USER: A New Versatile Channel for Kernel-Bypass Networks. In *Proc. of the ACM Workshop on Kernel-Bypass Networks (KBNets '17)*.
[33] M Tsirkin. 2010. vhost-net and virtio-net: Need for Speed. In *Proc. KVM Forum*.
[34] M Mitchell Waldrop. 2016. The chips are down for Moore's law. *Nature News* 530 (2016).
[35] 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 *Proc. of ACM SIGCOMM '15*.