Virtualization-Aware Traffic Control for Soft Real-Time Network Traffic on Xen

Chong Li, Sisu Xi, Chenyang Lu, Roch Guérin, Christopher D. Gill
Washington University in St. Louis
E-mail: {chong.li, xis, lu, guerin, cdgill}@wustl.edu

Abstract—As the role of virtualization technology becomes more prevalent, the range of applications deployed in virtualized systems is steadily growing. This increasingly includes applications with soft real-time requirements that benefit from low and predictable latency, even when co-located with other virtualized hosts with arbitrary traffic patterns. In this paper, we examine the policies and mechanisms affecting communication latency between virtual machines based on the Xen platform, and identify limitations that can result in long or unpredictable network stack latency for virtual machines deployed on this platform. To address these limitations, we propose and implement VATC, a Virtualization-Aware Traffic Control framework that supports differentiation (via rate-limited prioritization) of outbound and inbound network traffic from co-located virtualized hosts. Results of our experiments show how and why VATC can offer predictable (soft) latency guarantees to applications running on virtualized hosts with minimum overhead.

I. INTRODUCTION

Virtualization technology has gained wide adoption in data centers and clouds as an enabling technology for Infrastructure as a Service (IaaS). Virtualized systems are consequently increasingly used to support a broad range of applications, including distributed real-time applications with end-to-end communication latency constraints, e.g., industrial automation systems operating on edge computing platforms. In such systems, network I/O plays a major role in communication performance. In particular, the main challenge faced by virtual machines (VMs) running latency-sensitive (soft real-time) applications is that they are likely to be deployed in the same host as VMs that run bandwidth-intensive (bulk) applications. They therefore share with those VMs resources used in network I/O operations, such as CPUs and network interface cards (NIC). While NIC sharing mechanisms are reasonably well understood, CPU consumption for network I/O processing involves a complex range of interactions that are harder to predict and control. This makes meeting the latency constraints of real-time applications in virtualized environments challenging.

In non-virtualized hosts running Linux, a queueing discipline (QDisc) layer implements traffic control functionalities in the kernel, including traffic classification, prioritization and rate limiting. QDisc supports different queueing disciplines such as Prio [2] and FQ_CoDel [3], which can prioritize network traffic. When combined with Hierarchical Token Bucket (HTB), these queueing disciplines can achieve performance differentiation and isolation among latency-sensitive and non-real-time network flows sharing a NIC. These queueing disciplines are also used in virtualized hosts based on Xen [4], a widely-used open-source virtualization platform. Xen employs a manager domain1 called domain 0 (dom0) to manage the other domains (guest domains). By default dom0 runs a Linux kernel and is responsible for processing network traffic on behalf of the guest domains. Virtualization can, however, introduce priority inversions within dom0 transmission (TX) and reception (RX) routines. This arises from CPU sharing between network flows during those network I/O operations, and controlling this sharing is outside the scope of QDisc’s capabilities. Those priority inversions can then result in latency-sensitive traffic experiencing significant delay variations when traversing dom0.

This paper proposes virtualization-aware traffic control for network traffic in virtualized hosts, and implements the proposed approach in Xen. Specifically, the paper makes the following contributions: (1) it identifies the impact of QDisc traffic control mechanisms and the virtualization-related components in the network path of Xen; (2) it identifies limitations of Xen’s network architecture that can give rise to uncontrolled CPU sharing between network flows in dom0; and (3) it introduces VATC, a Virtualization-Aware Traffic Control scheme that achieves performance differentiation and isolation on multi-core CPUs, thereby offering greater latency control and predictability for latency-sensitive applications.

II. BACKGROUND AND MOTIVATIONS

As mentioned earlier, Xen uses a Linux-based manager domain (dom0) to handle network traffic from and to guest domains. Xen’s network stack is thus similar to that of the standard Linux distribution, but with additional virtualization-related components. Understanding to what extent virtualized platforms can offer latency guarantees, therefore, calls for exploring how Linux policies and mechanisms, including queueing disciplines, the sharing of transmission and reception queues, and the frequency with which interrupts (notifications) are generated and serviced, interact in a virtualized environment. A first contribution of this paper is, therefore, to offer a careful study of such interactions and how they affect latency under different traffic configurations.

---

1Following Xen’s terminology, we use the term domain in place of VM.
In this section, we first review Linux packet transmission and reception routines, which have been stable since version 2.6. Next, we study the architecture of dom0 with a focus on virtualization-related components. We select Linux 3.18 as our platform. We identify a number of limitations in its design, and explore their implications for latency guarantees. The relevant aspects of Linux kernel network stack have remained mostly stable since version 3.18. Conclusions drawn using this platform remain relevant to date and has lasting implications for virtualized network I/O architecture.

A. Network Stack in Linux

![Diagram of Transmission/Reception in Standard Linux]

**Transmission:**
- T1: packets are transferred from app to TX driver queue

**Reception:**
- R1: interrupt handler inserts netdev into the poll_list and
- R2: raises NET_RX_SOFTIRQ
- R3: NET_RX_SOFTIRQ handler cleans up TX driver queue and
- R4: delivers packets from RX driver queue to app

1) Transmission Routine in Linux: As shown in Figure 1, in Linux, packets from applications are processed by the network stack in the kernel. Because the virtualization-extensions of Linux only change the link layer, we omit transport and network layers in the figure. Packets coming from an application are enqueued into the appropriate queueing discipline (QDisc) queue(s) in the link layer, which is where Linux implements traffic control that determines when those packets are inserted in the TX driver queue, also known as the ring buffer. The TX Driver Queue is essentially a FIFO queue that works closely with the NIC.

**Queueing Discipline:** The QDisc layer implements traffic classification, prioritization and rate limiting. These can achieve flow differentiation as well as isolation in a shared NIC. In Linux, QDisc settings are configured through the TC command. By default, Linux uses pfifo_fast (no differentiation) as its queueing discipline.

QDisc can however be configured to prioritize packet transfers to the TX Driver Queue and therefore reduce transmission queuing delay for latency-sensitive applications. In particular, QDisc supports Prio [2], a queueing discipline with multiple priority queues. Prio works in cooperation with packet filters, which distribute packets from different flows (applications) into different queues. When the dequeue function of Prio is called, the order in which packets are dequeued from queues goes from high-priority to low-priority. Hence, assigning latency-sensitive applications to the highest priority queue can ensure shorter queuing delays. FQ_Codel [3] [5] is another queueing discipline that works to reduce queuing delay. FQ_Codel has one queue per flow, with a quantum for each queue. Once the quantum is reached, the corresponding queue is classified as a negative deficit queue, which has low-priority. This policy thus offers short queuing delays to latency-sensitive applications with low throughput.

The QDisc layer also includes mechanisms to throttle and shape network flows. The two most prevalent are Hierarchical Token Bucket (HTB) and Hierarchical Fair Service Curve (HFSC). HTB relies on a standard token bucket algorithm to shape packet transmissions and assign them to different priority queues. HFSC offers similar functionality with the addition of a queue size limit that results in dropped packets when exceeded, and can in some scenarios make for a more responsive behavior. Both HTB and HSFC allow throttling of high-priority flows to avoid instances where they saturate the NIC and starve low-priority flows.

**TX driver queue:** Packets remain pending in the TX driver queue until the next DMA transfer to the NIC. Congestion in the TX driver queue can, therefore, have a critical influence on packet transmission delays. Congestion usually arises when many large packets are forwarded to the NIC and either the hardware is not capable of processing them fast enough or the link is itself not sufficiently fast to keep up with the rate at which those packets are arriving.

TX driver queues support limiting the number of queued packets. This control is however insufficient to prevent congestion when the bulk of the NIC traffic consists of large packets. This limitation has been addressed in Linux kernels beyond 3.3 through the introduction of a Byte Queue Limit (BQL) [6] policy that limits the number of bytes in the TX driver queue of the NIC. With BQL, the size of the TX driver queue is limited dynamically, based on the inferred traffic mode, i.e., low-latency or bulk (see below), and throughput. Once the queue size hits the limit, the QDisc layer holds or drops subsequent packets. Once the queue size is under the BQL limit, the interrupt handler notifies the QDisc layer to resume releasing packets to the TX driver queue.

**Packet processing and triggers:** With the flow of packets in the transmit direction and the set of mechanisms available to control them now clear, it is important to realize that packet processing operations (as opposed to transmissions) and the triggers behind them can also play an important role in determining packet delays.

In most NIC drivers, a TX completion (hardware) interrupt is triggered when packets are successfully sent by the NIC. The interval between TX completion interrupts (the interrupt throttle rate) can be configured. In clusters and data centers, where low-latency communication is vital [7], users tend to configure a small interval. However, frequent interrupts can generate a heavy CPU workload and adversely affect
packet transmission and reception routines. Conversely, if the interrupt interval is too large, the TX driver queue may become congested because it is not refreshed often enough. In this case, packets remain pending in the QDisc layer and can experience long delays. Several NIC driver vendors, therefore, offer dynamic interrupt throttle rates or coalescing options that adjust this interval based on traffic type (interactive or bulk).

The handler of the TX completion interrupt, when scheduled, inserts a netdev device (a software data structure representing the NIC driver) into a poll_list (a per_CPU data structure in Linux), and then raises a software interrupt called NET_RX_SOFTIRQ. When the NET_RX_SOFTIRQ handler is scheduled it services the poll_list. The NET_RX_SOFTIRQ handler processes the network devices in the poll_list in a round-robin order, with a quantum of 64 packets. The NET_RX_SOFTIRQ handler function ends when either no device in the poll_list has packets pending, or it has serviced over 300 packets or has run for > 2 jiffies. This determines the resulting processing latency.

When a netdev device is fetched, the NET_RX_SOFTIRQ handler invokes the driver’s NAPI poll() method. Depending on the driver, the NAPI poll() method may perform different actions. In the driver used in our experiments, the NAPI poll() method both cleans up the TX driver queue and receives packets from the RX driver queue as needed (see below).

2) Reception Routine in Linux: Figure 1 also shows network reception in Linux, in which packet arrivals trigger hardware interrupts that consequently compete for processing resources with those in the transmit direction. The handler of those interrupts, when scheduled, again inserts a netdev device (the same structure as the one mentioned above) into the poll_list and similarly raises NET_RX_SOFTIRQ. As when invoked in the transmit direction, the NET_RX_SOFTIRQ handler fetches netdev devices from the poll_list and calls the corresponding NAPI poll() method. When the netdev device associated with the RX driver queue is handled, packets are delivered to the RX driver queue to the upper layer.

B. Network Stack Modifications in Xen

Recall that Xen relies on a Linux-based manager domain, dom0, to handle network I/O. Figure 2 shows the corresponding transmission and reception routines in dom0, together with their modifications in support of virtualization.

Each guest domain has a corresponding virtual interface (vif) (a virtualization-related component) in dom0, with a buffer dedicated to transmissions from a guest domain to its vif. Additionally, each vif also boasts a separate rx_queue for incoming packets destined to the domain with, as we shall see, a dedicated kernel thread (rx_kthread) responsible for handling those packets.

Transmission Routine in dom0: When a guest domain has a packet to transmit it first notifies dom0. The notification handler then inserts the corresponding vif device into the poll_list, and raises a NET_RX_SOFTIRQ. When the NET_RX_SOFTIRQ handler is scheduled, it processes all the devices in the poll_list in the same way as in standard Linux. After the NET_RX_SOFTIRQ handler function ends, other pending softirqs are processed. If a notification handler raises the NET_RX_SOFTIRQ before that processing finishes, the NET_RX_SOFTIRQ handler function is invoked again after other pending softirqs have been processed. In situations where the NET_RX_SOFTIRQ is frequently raised, it is therefore possible for softirq processing to run continuously for an extended period of time.

Reception Routine in dom0: When a packet arrives, the hardware interrupt handler inserts the netdev device into the poll_list and raises a NET_RX_SOFTIRQ. The handler then delivers packets from the RX driver queue to the rx_queue of the destination vif device. Each vif device has a corresponding reception kernel thread (rx_kthread). When packets are inserted into an rx_queue, the corresponding rx_kthread is also triggered. Packets are then forwarded from the rx_queue to the guest domain when that rx_kthread is scheduled. This must wait, however, until after all softirq processing finishes, which, as mentioned earlier, can cause delay.

C. Traffic Control Limitation in Xen

In both Linux and Xen’s dom0, network traffic processing requires NIC and CPU resources. Hence, when latency-
sensitive network flows face non real-time (data-intensive) competitors, the traffic control mechanisms should be able to differentiate and isolate (rate limit) both NIC and CPU sharing. However, Xen traffic control mechanisms rely only on QDisc, which is limited to controlling the sharing of NIC resources. This leaves the sharing of CPU resources unprotected. As we discuss next, contention for CPU resources can therefore result in scenarios where latency-sensitive traffic experiences non-trivial delays in dom0. We term those “limitations” and identify two such instances.

**Limitation 1: Priority Inversion between Transmissions:**
In scenarios where latency-sensitive domains and non-latency-sensitive domains coexist and transmit packets simultaneously, their vif devices are all inserted into the poll_list and serviced in a round-robin order. A vif device holding latency-sensitive packets can, therefore, be delayed by other vif devices serviced before it in the poll_list. This can be mitigated by reducing the default service quantum of 64 packets allowed each network device in the poll_list. However, this comes with its own overhead and, as we shall see next, is ineffective in addressing another limitation that arise from interactions between packet transmissions and receptions.

**Limitation 2: Priority Inversion between Transmission and Reception:**
When the NET_RX_SOFTIRQ handler forwards an incoming packet to the vif device, it wakes up the corresponding rx_kthread to process the packet. However, the rx_kthread can only be scheduled after the softirq processing finishes. In CPU-bound situations with many domains sending packets at a high rate, NET_RX_SOFTIRQ can be raised frequently (by the notification handler), so that the poll_list is continuously serviced, which can delay the execution of rx_kthreads. This can also produce priority inversions, albeit now from competition between packet transmissions and receptions. Reducing the poll_list quantum is then ineffective, as it does not affect the total softirq processing time. Specifically, a smaller quantum simply defers pending transmissions to the next insertion of the vif in the poll_list. This prevents long uninterrupted strings of transmissions from the same domain, but as vifs get immediately reinserted in the poll_list, it does not reduce the total softirq processing time.

In summary, both limitations arise from contention for CPU resources among virtualization-related components in dom0, which as we demonstrate in Section V-A can produce significant delays for latency-sensitive applications. Addressing those limitations is the motivation behind our proposed *Virtualization-Aware Traffic Control* (VATC) scheme, which we describe next.

### III. Design and Implementation

The simplest option to mitigate priority inversions in the transmit direction is to extend priority awareness to the vif devices in the poll_list. However, as pointed out in the previous section, priority inversions can also arise because of interactions between packet transmissions and receptions. As a result, rather than implementing a priority-aware vif scheduler for packet transmissions, VATC is designed to provide fine-grained kernel-thread-based traffic control.

In Linux, both the scheduling policy and the priority of kernel threads can be configured by users. SCHED_FIFO is a preemptive fixed-priority scheduling policy, under which a high-priority thread can preempt a running low-priority thread. VATC builds on this mechanism by ensuring that traffic associated with latency-sensitive domains is handled by high-priority kernel threads, while traffic from non-latency-sensitive domains is handled by low-priority kernel threads. Additionally, VATC provides a rate limiting mechanism that throttles the CPU utilization of high-priority threads to prevent CPU starvation of low-priority traffic. Finally, VATC supports multi-core dom0 through a dynamic vif rebalancing policy. VATC has been released under an open-source license at https://github.com/Chong-Li/VATC.

In the rest of this section, we first introduce how VATC achieves latency differentiation by thread prioritization. We then introduce the rate limiting mechanism and present how VATC scales to multi-core dom0.

#### A. VATC: Latency Differentiation

Figure 3 illustrates the overall structure of VATC. It introduces multiple (software) netback devices, and correspondingly multiple kernel threads (netbk_kthreads) to handle packet transmissions and receptions to/from different domains. These netbk_kthreads are configured with different priorities. Guest domains with the same priority (same latency requirement) share the same netback device and netbk_kthread. All the

[The back-end of the driver for virtual network devices in Xen, e.g., see https://lwn.net/Articles/430434/.

---

---
netbk_kthreads are scheduled under a SCHED_FIFO policy. The number of netback devices (netbk_kthreads) can be configured based on the number of priority levels needed.

For clarity, Figure 3 uses two priority levels. Domains 1 and 2 are running real-time (latency-sensitive) applications and are assigned to a high-priority thread. Domains 3 and 4 are low-priority domains running bandwidth-intensive (non real-time) applications. The high-priority netbk_kthread(H) handles the network traffic of the latency-sensitive domains, and the low-priority netbk_kthread(L) handles traffic of the bandwidth-intensive domains. The net_recv_kthread, which is triggered by TX completion and the RX interrupt handler, is given the highest priority. It cleans up all transmitted packets from the TX driver queue and processes pending packets in the RX driver queue. Because both transmission and reception are handled by kernel threads, we remove the poll_list and software interrupt handling from V ATC. Next, we review packet transmission and reception in V ATC, as well as their interactions.

Packet Transmission in V ATC: As before, when a high-priority domain has packets to send, it notifies dom0. The notification handler in dom0 then triggers the corresponding high-priority netbk_kthread, which can preempt lower-priority threads (this eliminates Limitation 1). Packets from the high-priority domain are first enqueued\(^6\) in the tx_queue of the corresponding netback device. The thread then checks whether the BQL limit of the TX driver queue has been reached. If it has, i.e., the TX driver queue is congested, it suspends itself until the net_recv_kthread cleans up the TX driver queue and refreshes the queue size.

After cleaning up the TX driver queue, the net_recv_kthread notifies the suspended netbk_kthread to resume (if the new queue size is under the BQL limit). If multiple netbk_kthreads are suspended, they resume one by one in order of priority. Note that each netbk_kthread can process packets in the tx_queue in FIFO order because the source domains of these packets have the same priority. Packets go through the QDisc layer as usual\(^7\), before being put into the TX driver queue. Once the high-priority tx_queue is empty, the high-priority thread stops so that lower-priority threads can run.

Packet Reception in V ATC: In V ATC, when a packet arrives, the RX hardware interrupt handler wakes up the net_recv_kthread instead of the NET_RX_SOFTIRQ. Once the net_recv_kthread is scheduled, it retrieves packets from the RX driver queue and forwards them to the rx_queue of the destination netback device. An enqueue operation at the rx_queue of a netback device wakes up the corresponding netbk_kthread. It is then scheduled once the net_recv_kthread finishes its work, and delivers packets from the corresponding rx_queue to its destination domain. If multiple netbk_kthreads are woken up by the net_recv_kthread, they are scheduled based on their priorities. This eliminates Limitation 2.

Interferences between Transmission and Reception: Recall that in the original dom0, Limitation 2 was primarily caused by the rx_kthread (dealing with packet receptions) being preempted by the NET_RX_SOFTIRQ handler as it could be repeatedly triggered by sustained transmissions. V ATC avoids this by eliminating reliance on softirq’s and instead relying on separate (high and low priority) threads for transmissions and receptions. However, there remains the possibility of interference between transmissions and receptions of different priorities. This is because the net_recv_kthread has the highest priority and does not differentiate traffic types. This possibility notwithstanding, we note that the processing behind net_recv_kthread is lightweight (moving packets from driver queue to rx_queues), so that it consumes a minimal amount of CPU resources to run to completion. In addition, the NIC’s DMA threshold and the use of interrupt coalescing (recall Section II-A) both contribute to limiting the frequency with which net_recv_kthread is triggered. In none of our experiments did it result in any form of priority inversion.

B. V ATC: Rate Limiting

As just introduced, V ATC achieves latency differentiation through the use of prioritized kernel threads. However, unlimited high-priority traffic (threads) could easily starve low-priority traffic (threads). In this section, we describe how V ATC’s rate limiting functionality prevents such occurrences.

There are currently two locations at which rate limits can be enforced in dom0. A rate limiter can be configured for the vif of a guest domain to limit its outgoing traffic [8], or, as described earlier, rate limiting can be realized using a Hierarchical Token Bucket (HTB) in the QDisc layer. As discussed in Sections II-A and II-B, the QDisc layer is located near the “bottom” of the dom0 network stack, so that while HTBs in the QDisc layer remain an effective tool to control guest domains bandwidth consumption, they offer little protection when it comes to managing access to CPU resources. Hence, preventing high-priority domains from depriving low-priority ones from access to CPU resources in V ATC’s dom0 must rely on vif-level rate limiters.

The current dom0 vif-level rate limiter is based on two parameters, rate and interval. The rate is the transmission rate in bytes/sec assigned to a domain/vif, while the interval specifies the frequency at which transmission credits are replenished. A rate limiter with a rate of 1 MBytes/s and an interval of 50 ms, therefore, allows a domain to transmit 50 KBytes every 50 ms. Unfortunately, because the rate limiter is byte-based, it fails to adequately control CPU resources consumption that are expended on a per packet basis.

To validate this hypothesis, we empirically benchmarked the impact of packet rates on CPU utilization in V ATC’s dom0 using the testbed described in Section V.B. For that experiment, we dedicated one CPU core to dom0. Guest domains (running on other cores), generated streams of UDP packets with payloads of different sizes while saturating the CPU of dom0. Figure 4 shows both the maximum throughput/bandwidth (left y-axis) and packet rate (right y-axis) of
dom0, as a function of packet payload size. As packet payload size increases, the packet rate required to saturate the CPU remains essentially constant, while throughput grows. This highlights that CPU rather than bandwidth is the bottleneck resource, with packet rate rather than packet size the main contributor to CPU utilization. We note that this is consistent with earlier experiments with a Linux system [9].

As a result, VATC replaces the byte-based vif rate-limiter with a two-parameter \((r, b)\) packet-based token bucket, where \(r\) specifies the allowed rate in packets/sec and \(b\) is the maximum packet burst. Such a token bucket is typically configured only for high-priority domains, and ensures that high-priority domains do not exhaust dom0 CPU resources and in the process starve low-priority domains. We note that this vif-level packet-based mechanism can still be combined with an HTB in the QDisc layer to also limit the bandwidth consumption of high-priority (and low-priority) domains.

C. VATC: Multi-core Support

With the increasing number of CPU cores on modern servers and while cloud providers are understandably reluctant to dedicate too many resources to virtualization functions [10], multiple cores may be dedicated to dom0 to support network-intensive applications. In this section, we introduce how VATC scales to a multi-core dom0 by distributing outgoing traffic to different cores.

Historically, Xen explored two alternative approaches to distributing traffic on multiple cores. In dom0-3.10 (Linux-3.10), the xen-netback driver adopts a static approach to allocate a domain’s vif to a core when the domain is created. Specifically, dom0-3.10 creates a netback device for each core. When a domain is created, its vif is assigned to a netback device (core) using a Least-Number-First (LNF) policy. The new domain’s vif is permanently assigned to the netback device with the least number of vifs at the domain’s creation time. This policy has two limitations. First, it ignores the fact that vif devices may have different network traffic loads. Moreover, the assignment is a one-shot decision. Both factors can lead to unbalanced loads across cores.

In contrast, dom0-3.18 and subsequent versions adopt a dynamic approach to load distribution that relies on the Linux irqbalance daemon [11] to balance notifications (from guest domains to dom0) across CPU cores. In Xen, notifications are treated as hardware interrupts with affinity configurations that specify which CPU cores handle them. We note that, as is common in Linux, interrupt servicing is split into two halves\(^8\), a top half and a bottom half, with the top-half (irqbalance) responding to the interrupt and scheduling the bottom-half routines responsible for the processing itself. The irqbalance daemon dynamically re-configures the affinity of each interrupt and notification so that their subsequent handling is balanced across CPU cores. When a notification is delivered to a core, the corresponding vif is inserted into the core’s local poll list to be subsequently serviced by the softirq handler. This irqbalance-based solution therefore balances traffic more responsively than a static vif-allocation approach. This solution has, however, several drawbacks that impact its ability to balance traffic load in dom0. First, the irqbalance daemon intends to balance all interrupts regardless of their types, e.g., NIC’s TX/RX interrupts\(^9\), IPI, timers and notifications, etc. As notifications are only a subset of the interrupts, irqbalance cannot guarantee that guest domains’ traffic is evenly distributed across CPU cores. Furthermore, as irqbalance is unaware of domain priorities, it may also distribute latency-sensitive, high-priority traffic unevenly across cores.

The load balancing mechanism of VATC accounts for many of those issues by combining the responsive features of dom0-3.18 with a priority-aware vif-allocation approach reminiscent of that of dom0-3.10. Specifically, VATC still relies on irqbalance for the top-half processing of all notifications, but, as we describe next, maps high-priority vifs to specific cores to balance the load of the bottom-half packet processing.

- **Rate-aware vif-binding.** While the vifs of low-priority domains are dynamically balanced across cores as in dom0-3.18, high-priority vifs (i.e., vifs from domains whose traffic is rate-controlled) are assigned to cores based on the packet (token) rates of the corresponding guest domain. This ensures that each core is assigned a (high-priority) packet load consistent with the latency targets of the high-priority traffic. To realize this goal, VATC employs a Least-Load-First (LLF) policy: The policy maintains a per-priority sorted list of vifs in decreasing order of packet rates, and assigns vifs to cores based on a worst-fit bin-packing heuristic. Specifically, vifs are assigned to core in descending order of packet rate, and for each vif, the netback device (core) with the minimum allocated packet rate is selected.

- **Adaptive reconfiguration.** Unlike the static allocation of dom0-3.10, VATC avoids creating imbalances as domains come and go by rebalancing high-priority vifs whenever a guest domain is created or shutdown. The creation and shutdown of guest domains are relatively rare events, so that rebalance operations are infrequent. Furthermore, from our measurements, migrating an individual vif during a rebalancing operation takes less than 10 nano seconds on our testbed. Hence, the overhead of VATC’s

---


\(^9\) In spite of their high priority, we found the impact of those interrupts limited to no more than 25% of the CPU load in our system under a fully saturated configuration.
rebalancing is moderate. This is not overly surprising since a rebalancing operation simply amounts to updating a variable that specifies which netback a vif binds to. There may be additional overhead following such an operation, e.g., because of cache misses. However, the relatively low frequency with which such penalty may be incurred should contribute to keeping it acceptable.

- **Priority-aware bottom-half processing.** VATC adds priority-awareness to the bottom-half processing of notifications from guest domains. This is realized by creating on each core a net_recv_kthread and a per-priority netback device with the corresponding priority setting. As a result, on any core traffic within a given priority class is handled by a different (prioritized) netback device, thereby preserving latency differentiation.

IV. RELEVANCE TO RECENT PLATFORMS

The results presented in the paper relied on a Linux/Xen release and hardware platform that, as of today, are relatively old. In this section, we discuss the implications of this work for more current software and hardware platforms. Our analysis suggests that the problems VATC targeted still exist and may even have been exacerbated by differences in the relative pace of change between computation and communication. As a result, most of the findings still stand and the approach and design remain applicable and relevant to current Linux/Xen implementations and hardware settings.

A. Current Linux/Xen Implementation

The analysis and design of VATC was based on Linux 3.18 that is now several years old. However, the relevant aspects of the Linux network stack have seen little modifications in subsequent releases. Specifically, we analyzed the latest long-term Linux version, 5.10.4710 and found that the problem identified in the paper still exists. In this latest release, guest domains’ vif devices and NIC’s netdev are still inserted into a per-core shared poll_list, and consequently are, as before, handled in a round-robin order. This architecture inevitably results in the same priority inversions (the two Limitations of Section II-C) that the paper identified and addressed. As a result, the possibility of priority inversion that was present in the network stack of release 3.18 of Linux/Xen has persisted to this day, so that the type of solution that VATC offers continues to be relevant.

B. Increasing Network Speed

Network speeds have grown at a faster rate than CPUs, so that current line speeds are usually higher, relative to processing speeds, than what was used in our experiments. Arguably, faster links only amplify the problem that VATC seeks to address, as CPUs are then even more likely to become the bottleneck. The scheduling approach of VATC is designed to deal with CPU contention, namely, scenarios where the processing is the primary resource bottleneck, which becomes even more likely with faster links. In other words, we anticipate that faster link speeds (relative to CPU speeds) make the problem that VATC targets even more acute than what our experiments display.

There is, however, one aspect of the faster growth of link speeds over CPU speeds that VATC may not be able to fully address, namely, that it increases the odds of priority inversions in net_recv_kthread (packet reception from the hardware – see Figure 3) where traffic differentiation is not available. To mitigate the priority inversions that can arise in net_recv_kthread, we have minimized its processing footprint such that it is only responsible for checking packets’ header before directing them to what is then a prioritized netback rx_queue. Hence, we have kept undifferentiated packet processing to a minimum. We also note that priority inversion on the receive side is indicative of an inherent imbalance between link and CPU speeds, which is normally handled by dedicating multiple CPU cores to dom0 to match packet processing (in the kernel stack) to link speed. As VATC relies on a distinct net_recv_kthread at each CPU core of dom0, any priority inversion those threads can create are largely mitigated.

C. Increasing Number of CPU Cores

Modern servers comprise increasing numbers of CPU cores. Having access to more cores facilitates allocating multiple CPUs to dom0, which, as mentioned, can alleviate CPU contention. However, this cannot avoid priority inversion as long as real-time and non-real-time traffic share any of those CPUs. One approach would then be to dedicate specific cores to real-time traffic. This is, however, likely to be wasteful of CPU resources as it can lead to significant under-utilization of those cores. This runs counter to the goal, common in today’s clouds, of accommodating heterogeneous users’ requirements while maintaining a high level of resource utilization. VATC was designed to address such a scenario and realize an efficient trade-off by prioritizing traffic through dedicated threads and queues (instead of entire cores), thereby reducing priority inversion without over-provisioning CPU resources.

D. SmartNICs

Modern multi-queue NICs can map traffic flows to different queues and mitigate priority inversions at the hardware layer. However, this cannot avoid priority inversion in the software stack (when traffic is handled by NAPI polls). Alternatively, kernel bypass solutions exist that directly map VMs to dedicated NIC queues. This mapping is often hardware specific and lacks flexibility so that some kernel stack features then have to be re-implemented in the user space of VMs, which introduces a trade-off between performance and flexibility. This has in turn motivated the development of SmartNICs that have seen increased adoption in public clouds [10]. SmartNICs deliver the hardware performance of kernel bypass solutions, while preserving the flexibility of software-based kernel approaches. However, while they can often eliminate the type of priority inversion that VATC addresses, they come at a cost and may, therefore, not be suitable in all settings, e.g., in edge computing platforms.

V. Evaluation

This section presents an empirical evaluation of VATC’s three main features: 1) latency differentiation, 2) rate limiting, and 3) multi-core support.

A. Latency Differentiation

As outlined in Section II, various factors can delay soft real-time traffic in Xen. In this section, we explore a number of scenarios where such delays can arise, and both quantify their magnitude and analyze their causes. We evaluate latency and latency predictability for delay-sensitive traffic under our implementation of VATC, and under existing Xen dom0 traffic control mechanisms that are primarily aimed at controlling network traffic, i.e., Prio and FQ_CoDel.footnote{FIFO is the default traffic control scheme in Linux, but as expected it performs poorly when it comes to latency guarantees. As a result, we only compare the latency of VATC to that of Prio and FQ_Codel.}

The evaluation is carried out on a testbed consisting of six physical machines, hosts 0 to 5. Host 0 is an Intel i7-980 six core machine with CentOS 4.3 installed, on which dom0 is a 64-bit CentOS built on Linux kernel 3.18.0. Host 0 acts as the host server. Five other physical machines, hosts 1 to 5, run Linux. All machines are equipped with Intel 82567 Gigabit NICs and are connected by a TP-LINK TL-SG108 Gigabit switch. Because both Prio and FQ_CoDel are fine-grained packet schedulers, it is recommended [12], [13] that the TCP Segmentation Offload (TSO) and Generic Segmentation Offloading (GSO) of the NIC be disabled, which we do. This ensures that large packets with a size greater than the MTU (1,500 bytes in our system) are segmented in the kernel instead of in the NIC, and avoids long head-of-the-line blocking delays in the TX driver queue. Our NIC driver uses the NAPI poll() method, which is invoked by the NET_RX_SOFTIRQ handler, to clean up the TX driver queue.

In host 0, dom0 is initially assigned one dedicated physical CPU core. This is common practice to handle communication and interrupts [14], [15], and is also recommended by the Xen community to improve I/O performance [16]. We boot up five guest domains, domain 1 to domain 5 on host 0. Each of them is pinned to a separate physical CPU core to avoid influences from the domain/VM scheduler. In our setup, domain 1 is the latency-sensitive domain and domains 2 to 5 are interfering domains. Hence under VATC, traffic from/to domain 1 is handled by a high-priority kernel thread in dom0, while traffic belonging to domains 2 to 5 is handled by a low-priority one.

The round-trip latency between domain 1 and host 1 is measured as follows. Domain 1 pings (with ICMP packets) host 1 every 10 ms, and host 1 replies back. This traffic pattern seeks to emulate the behavior of common periodic real-time applications. Each experiment records latency values for 1,000 ICMP request/response pairs. We report both median and tail latency (95th percentile). Tail latency is important to many soft real-time applications because it reflects latency predictability. In domain 2 to domain 5, we run the stream test of Netperf [17] to simulate non-real-time applications.

The Intel NIC in our hosts supports interrupt intervals from 10µs to 10ms. The Intel NIC driver also provides two adaptive modes, dynamic conservative (50µs to 250µs) and dynamic (14µs to 250µs). Both modes dynamically adjust the interrupt interval based on the type of network traffic, bulk or interactive. The dynamic conservative mode is the default mode of the Intel NIC driver. We evaluate both modes and a range of static values.

Latency of high-priority (latency-sensitive) traffic is measured for scenarios where resource (CPU or network) contention arises in dom0. CPU contention can occur when low-priority domains are sending many small packets, as NET_RX_SOFTIRQ handler is then frequently triggered. Its impact can be compounded by setting the interrupt handler interval to a small value, as the bottom-half processing of the interrupt handler can then overload dom0. As we shall see, in these scenarios, the two Limitations identified in Section II, can introduce long queueing delays in virtualization-related packet handling components. Network contention arises when guest domains generate a high volume of large packets that saturate the NIC. In those cases and as we shall again see, delays can also happen. However, they are not specific to virtualization components as they arise directly in the hardware. As a result, VATC performs similarly as a base configuration that only relies on existing traffic control schemes.

We explore CPU-contention and Network-contention scenarios in Section V-A1 and Section V-A2, respectively, and throughout the experiments use the terms small packet stream and large packet stream to identify the (UDP) packet size of traffic from low-priority domains, i.e., 1 byte for small packet streams and 1,500 bytes for large packet streams. For reference purpose, we note that the average latency through a lightly loaded dom0 is 0.1ms.

1) CPU-Contention Scenarios: We evaluate the latency of high-priority traffic in the presence of interfering low-priority small packet streams. Figures 5 and 6 show the round-trip latency (median and 95th percentile) of ICMP packets from the high-priority domain for different numbers of interfering low-priority UDP streams and different interrupt intervals (from 10µs to 1024µs), as well as when using the dynamic and dynamic conservative settings (right-hand-side of the figure with corresponding labels on the x-axis).

To isolate the impact of interrupt interval size, we first focus on a scenario involving a single interfering stream (Figures 5a and 6a). We note that because packets are small, which is more typical of interactive traffic, the dynamic conservative mode and the dynamic mode default to setting the interrupt interval to the lower bound of their range, i.e., 50µs and 14µs, respectively. Both figures illustrate that, irrespective of the mechanism used, latency grows with the size of the interrupt interval. This is because TX/RX operations are triggered less frequently for longer interrupt intervals.

As multicore systems allow the presence of multiple guest domains in a system, with each domain possibly generating small packet streams, we consider next scenarios that involve multiple such streams from different low-priority domain. The results are reported in Figures 5b, 5c, and 5d, which show the median latency of high-priority packets with 2, 3, or 4
interfering low-priority UDP streams, while Figures 6b, 6c, and 6d show tail latency for the same configurations.

We note from the figures that VATC’s latency performance is unaffected by the number of interfering streams, as the high-priority netbk_kthread handling the real-time traffic cannot be preempted by the lower priority ones. In other words, VATC successfully mitigates Limitation 1 and Limitation 2.

In contrast, under Prio and FQ_CoDel, the high-priority stream latency (median and tail) increases with the number of interfering streams. To better understand the sources of these increases, we performed a detailed analysis of the contributions of individual dom0 components. Under Prio and FQ_CoDel, queueing delays may arise in:

- **vif**: when the NET_RX_SOFTIRQ handler is servicing other vif devices in the poll_list, high-priority packets are pending in the corresponding vif device;
- **QDisc**: when there is congestion in the TX driver queue, packets are kept waiting in the QDisc layer;
- **rx_queue**: After ICMP response packets are forwarded to the rx_queue, the corresponding rx_kthread must wait for the softirq processing to finish before it can be scheduled.

Figure 7 displays delays in these three components under Prio (results are similar under FQ_CoDel). As the number of interfering domains increases, so does the number of vif devices inserted in the poll_list. This contributes a small but steady increase in delay. The delay in the QDisc layer also increases, as there are now more pending packets in the TX driver queue, and the bottom-half processing of the TX completion interrupt is delayed since in our NIC driver the TX completion interrupt handler simply inserts the netdev device into the poll_list and leaves the bottom-half processing (clean up of the TX driver queue) to be executed by the NET_RX_SOFTIRQ handler. As the number of vif devices in the poll_list grows, the netdev device is serviced less frequently. This increases congestion in the TX driver queue, which induces longer queueing delays in the QDisc layer. Other NIC drivers clean up the TX driver queue in the hardware interrupt handler, which avoids having vif devices in the poll_list delay the clean-up of the TX driver queue; in turn reducing the queueing delay in the QDisc layer.

Both contributions to higher latency can be attributed to Limitation 1, but Limitation 2 can be seen to have an even more pronounced effect. Figure 7b shows that even only two interfering streams can significantly affect the delay in rx_queue. This is caused by the NET_RX_SOFTIRQ handler repeatedly servicing the poll_list when the softirq is raised frequently by notification and interrupt handlers. This can then result in the rx_kthread being delayed for an unpredictably long time as illustrated in Figure 6b which captures the tail of the delay distribution. For comparison purposes, the latency 95th percentile is 40 times higher in Prio than in VATC.

Interestingly, this trend somewhat reverses as the number of interfering streams increases beyond 2. This is because the NET_RX_SOFTIRQ is raised less frequently as more low-priority streams are added. Adding streams eventually overloads dom0. This results in many low priority packets not serviced in time, so that a backlog builds-up in the buffers between vif devices and the corresponding (low-priority) guest

---

12 NIC drivers that clean up the TX driver queue in the hardware interrupt handler without, therefore, raising NET_RX_SOFTIRQ might again reduce this workload. However, because the dominant contribution to the softirqs is the notification handler, we do not expect this would be of much benefit.
domains. This in turn prevents those domains from putting more packets into the buffer, and consequently from issuing new notifications until the buffer is refreshed. As notification handlers are the dominant contributors to the number of softirqs raised, this reduces their number.

The next experiment explores further the impact of notification frequency on latency performance. The results are shown in Figure 8, which parallels Figure 7 but keeps the total throughput of the interfering traffic constant and evenly distributed across streams (as opposed to each stream contributing their own independent traffic volume). This largely eliminates the possibility of congestion in the buffer between each vif and guest domain. Consequently, the notification frequency from guest domains is much higher than in the previous experiments. For example, with 4 interfering streams, we measure a notification frequency that is 100 times larger than with the same number of streams each contributing their own traffic. This difference is largely responsible for the significant increase in latency seen between Figures 7 and 8 (the worst delay observed in the experiment of Figure 8 was 160 ms!). Of note in Figure 8 is the fact that while latency initially experiences significant increases as more streams are added, adding a fourth stream appears to contribute to a slight decrease. We were not able to pinpoint the exact sources of the decrease, but conjecture that it may be partially due to some streams now not always having new packets, which would in turn lower the notification rate.

Summary: Limitation 1 and Limitation 2 are both present under Prio and FQ_CoDel, which exposes latency-sensitive traffic to significant interference. VATC eliminates the sources
of those interferences in dom0 virtualization components by dedicating a netback device and a prioritized kernel thread to each priority level. This guarantees the availability of dom0 CPU resources to latency-sensitive streams.

2) Network-Contention Scenarios: The next set of experiments explores the ability to protect latency-sensitive stream in scenarios where network bandwidth is the scarce resource. This is realized by relying on large packet streams (either UDP or TCP) as the source of interference. A large packet stream is created by having a Netperf stream application run in domain 2 and send a stream of large packets to host 2, with an intensity sufficient to saturate the network link. Note that since the majority of the traffic is now large packets, both the dynamic conservative mode and the dynamic mode set their interrupt interval consistently to the upper bound of their range, i.e., 250μs. Their performance is, therefore, now aligned with that of a static interrupt interval at that value.

Large packets can congest the TX driver queue, so that high-priority (ping) packets may end-up waiting in the QDisc layer for a long time. This delay depends on the duration of the (TX completion) interrupt interval, as packets pending in the QDisc layer can only be sent after the handler cleans up the TX driver queue. Meanwhile, ICMP response packets can also experience long delays in the NIC, because the RX interrupt frequency is also limited by the interrupt interval configuration.

Figures 9a and 9b show the median latency (tail latency is close to the median value) of high-priority packets with one interfering UDP and TCP large packet stream, respectively. We see that the round-trip latency is penalized as the interrupt interval increases. Because the bulk of the delays either happen in the hardware itself (on the receive side) or because of congestion induced by the hardware (on the transmit side), VATC whose focus is on the virtualization component offers little or no improvement in latency over Prio and FQ_CoDel.

B. Rate Limiting

To better stress VATC’s ability to not only preserve latency guarantees for real-time (high-priority) traffic, but also protect the packet throughput of non-real-time guest domains from potential abuses by real-time ones, we conducted the remainder of our experiments on a different testbed that supported higher traffic volumes. This testbed comprises 7 hosts connected by 40 Gbps Ethernet links. Each host has two 8-core Intel Xeon E5-2630 processors and 8 GB of memory.

We then proceed to evaluate the efficacy of packet-based token buckets in regulating the CPU resource consumed by high-priority traffic in dom0. In the following experiments, we create two high-priority and one low-priority guest domains. In each of the high-priority domains, we run iperf3 [18] to generate UDP network traffic with fixed throughput at 50 Mbps. We vary the packet size across runs so that each high-priority domain generates traffic at different packet rates but the same throughput. In the low-priority domain, we run iperf3 to measure its UDP/TCP throughput. To evaluate the impact of the packet-based token buckets, we compare the low-priority domain’s throughput when the high-priority domains are with and without packet-based token buckets. When rate-limited, each high-priority domain has a (20 packets) packet-based token bucket to regulate its consumption of CPU resources.

Figure 10 plots the throughput of the low-priority domain as the total packet rate of the high-priority domains increases. When the high-priority domains are not rate-limited, the (UDP/TCP) throughput of the low-priority domain drops sharply when the high-priority packet rate exceeds 100 kpackets/s due to increasing CPU contention. In contrast, when the high-priority domains are regulated by their packet-based token bucket, the low-priority domain maintains consistent throughput, irrespective of the (target) packet rates of the high-priority domains. This demonstrates the efficacy of the packet-based token buckets in preventing high-priority domains from saturating the dom0 CPU, thereby protecting low-priority traffic from excess high-priority traffic.

C. Load Distribution

This section considers a scenario where dom-0 is assigned multiple cores, and compares VATC’s load balancing mechanism (LLF) to two baseline solutions: 1) LNF, the static vif assignment policy adopted in dom0-3.18; and 2) IRQB, the dynamic load-balancing solution adopted in dom0-3.18 and subsequent versions.

We assume that two cores, core-0 and core-1, are dedicated to dom0, while 8 high-priority13 guest domains (dom-1 to

---

13We omit including low priority domains, as the priority structure of VATC makes high-priority performance insensitive to any incorrect initial allocation that LNF may make, and its reliance on the same dynamic load-balancing irqdb mechanism as IRQB for low-priority traffic means that both will offer similar performance for low-priority traffic.
share the remaining cores. Those guest domains generate packets at rates of 10, 20, 30, 40, 50, 60, 70 and 80 kpackets/s, respectively, for a total packet rate of 360 kpackets/s. As LNF assigns a domain’s vif to a core when the domain is created and the assignment is based on the current number of vifs per core, the resulting load distribution depends on the sequence of domain creation. In the best case, LNF evenly splits the traffic load and assigns 180 kpackets/s to each core, when the guest domains are created in the order of dom-1, dom-3, dom-2, dom-4, dom-7, dom-5, dom-8, dom-6. In the worst case, LNF assigns 260 kpackets/s to core-0 and the remaining 100 kpackets/s to core-1 when the domains are created in the order of dom-5, dom-1, dom-6, dom-2, dom-7, dom-3, dom-8, dom-4.

Table I, reports tail latency results (mean and standard deviation of the 99th percentile computed over 10 runs) on each core for LNF, IRQB and VATC under 5 different orders of domain creation for which LNF produces packet loads on core-0 ranging from 180 kpackets/s to 260 kpackets/s.

The results show that when the initial load allocation (column 180) is balanced, the three mechanisms perform similarly when it comes to mean performance. The situation, however, rapidly changes as LNF’s rate-unaware assignment becomes more unbalanced. In those scenarios (columns 200 to 250), LNF’s mean latency becomes increasingly worse (a difference of close to 50% for core-0), while the adaptive nature of both IRQB and VATC make them insensitive to the order in which domains are created. The table also illustrates that while IRQB and VATC perform similarly when it comes to mean latency, IRQB exhibits a much more variable tail latency (3 to 4 times larger than VATC). This is likely caused by the looser control it enforces on how notifications are assigned to the two cores.

VI. RELATED WORK

As soft real-time applications are widely deployed in virtualized platforms, protecting latency-sensitive traffic has become an important topic.

The network I/O control in VMware vSphere [19] can reserve I/O resources (e.g., network bandwidth) for business-critical traffic based on user-defined network resource pools [20]. In Windows Server 2012 R2 [21], Hyper-V QoS [22], [23] also provides bandwidth management to network traffic. In environments with network-contention, these can effectively enhance the performance of latency-sensitive VMs. However, because they focus on managing bandwidth, they may not effectively handle the priority inversions caused by CPU contentions as we observed in Xen, which is the focus of VATC. Therefore existing approaches to bandwidth management and VATC are complementary solutions for network- and CPU-contention scenarios, respectively.

KVM [24] is another Linux-based virtualization platform. It creates vhost threads to handle traffic from guest VMs. vhost threads are, however, not assigned priorities corresponding to the priorities of the VMs. In addition, because vhost threads service traffic as in standard Linux, KVM may experience similar priority inversion problems. For example, a vhost thread servicing real-time traffic can be preempted by threads servicing non-real-time traffic or by softirq handlers.

Xu et al. [13] investigate optimizing the network stack of Xen’s dom0 by fragmenting large packets into small ones so that BQL and FQ-CoDel can work more efficiently to reduce queueing delay. In addition to those network stack modifications, Xu et al. [13] also optimize the VCPU scheduler and the network switch to further reduce host-to-host latency in a data center setting. Jia et al. [25] improve I/O latency by dynamically migrating I/O tasks to active vCPUs. These works, however, do not consider queueing delays in the virtualization layer of dom0, i.e., the netback or vif devices, which can play a significant role.

RT-Xen [26]–[28] provides a real-time VCPU scheduling framework recently included in Xen 4.5. Xi et al. [29] develop RTCA, which implements a prioritization-aware packet
scheduling in the netback device of dom0. RTCA is able to offer real-time guarantees to communications between local hosts. VATC seeks to extend those guarantees to communications with remote hosts.

In spite of its cost [10], another related topic is improvement in guest domains communication performance by allocating additional cores to each domain. Xu et al. [15] improve the I/O performance of a multi-VCPU guest domain by delegating all its I/O processing to a dedicated VCPU. Because of the availability of a dedicated VCPU, the guest domain can process interrupts more efficiently with limited CPU overhead. Similarly, Har’El [30] proposes an efficient and scalable paravirtual I/O system by implementing a fine-grained I/O scheduling and exitless request/reply notification model in KVM. Neither of these two systems seeks to prioritize network traffic with different real-time requirements. Their goal is to improve average network performance in virtualized hosts (Xen or KVM).

As we have discussed, rate limiters are commonly used to achieve latency guarantee in shared network environments. Grosvenor et al. [31] leverage QDisc rate limiters and IEEE 802.1Q standard to bound queueing delay of different priorities at shared datacenter switches. Zhu et al. [32] use per-instance token buckets and network calculus to guarantee tail latency SLOs and optimize workload distribution across servers in data centers. Radhakrishnan et al. [33] present SENIC, which implements rate limiters and transmit schedulers in hardware. With the exception of [32] that is concerned with latency and workload distribution, but at the scale of a data center rather than a physical server shared by multiple VMs, these works focus on regulating bandwidth utilization. Instead, VATC focuses on controlling CPU access in a shared physical server by offering packet-based rate limiting and traffic prioritization above the native network stack. In other words, VATC is complementary to works that focus on “network layer” rate limiters.

Finally, as alluded to earlier, other works have focused on kernel passthrough technologies, such as SR-IOV [34] and DPDK [35], to bypass the network virtualization layer and dom0 to reduce network latency and have specialized hardware support for network communication. These technologies have been supported by commercial virtualization platforms [36], [37] and have been the focus of much academic research [38], [39]. However, as discussed in Section IV-D, the use of kernel bypass introduces a trade-off between performance and flexibility. Customized hardware such as SmartNICs can avoid this trade-off, but at an added cost. In contrast, VATC is able to offer soft real-time guarantees without additional hardware and while preserving the ability to leverage kernel space traffic control mechanisms.

VII. CONCLUSION

With the development of ever more powerful and flexible virtualization platforms, distributed soft real-time applications are increasingly deployed in virtualized environments. Those deployments introduce new challenges when it comes to guaranteeing low and predictable latency. This paper evaluates network latency in Xen in the presence of diverse traffic patterns and system configurations, including the use of several existing Linux network control mechanisms. Our investigation reveals that some virtualization-related components of Xen can introduce priority inversions in network transmission and reception. Because these limitations involves CPU sharing between network streams, Linux QDisc traffic control mechanisms that target control of network resources are ineffective in addressing those issues. This is the motivation behind VATC, a virtualization-aware traffic control framework that tackle those problems using a novel network I/O architecture based on prioritized kernel threads. VATC includes a packet-based rate-limiting mechanism to limit the CPU resources consumed by high-priority traffic, and a dynamic vif assignment policy to effectively balance load in a multi-core dom0. Our evaluation shows that VATC provides an effective software framework to support latency-sensitive network traffic on virtualized hosts.

VATC is designed to support prioritization among VMs. It may be possible to leverage multi-queue vifs to support prioritization at a finer granularity, e.g., prioritizing traffic within a vif when applications in the same VM have different latency requirements. Such an intra-VM prioritization capability would be complementary to the inter-VM prioritization provided by VATC. The integration of intra- and inter-VM prioritization is a direction of future research. Finally, while VATC is designed and evaluated on Xen, the approaches on which it relies could be applied to other virtualization and container platforms.

REFERENCES


