Prioritizing Soft Real-Time Network Traffic in Virtualized Hosts Based on Xen

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

***Abstract*—As virtualization technology becomes ever more capable, large-scale distributed applications are increasingly deployed in virtualized environments such as data centers and computational clouds. Many large-scale applications have soft real-time requirements and benefit from low and predictable latency, even in the presence of diverse traffic patterns between virtualized hosts. In this paper, we examine the policies and mechanisms affecting communication latency between virtual machines based on the Xen platform, and identify limitations that could result in long or unpredictable network traffic latencies. To address these limitations, we propose *VATC*, a *Virtualization-Aware Traffic Control* framework for prioritizing network traffic in virtualized hosts. Results of our experiments show how and why VATC can improve predictability and reduce delay for latency sensitive applications, while introducing limited overhead.**

I. Introduction

As computer hardware has increased in power, so has the use of virtualization technology in data centers and clouds. This two-prong progress has fostered the development of large-scale distributed real-time systems. In a virtualized environment, applications/services are deployed in Virtual Machines (VMs), so that the network I/O performance of the virtualized hosts becomes a critical component of meeting the communication requirements of distributed real-time applications. Examples of such applications include shipboard computing [1], where distributed mission-critical and safety-critical tasks are deployed in multiple servers, and are subject to end-to-end deadlines. Similarly, end-to-end latency constraints are also present in enterprise data centers and industrial automation systems that are increasingly deployed in virtualized environments. The main challenge in such settings is that VMs running latency-sensitive (soft real-time) applications are likely to be deployed in the same host as VMs that run bandwidth-intensive (bulk) applications. VMs in the same physical host share CPU and network I/O resources. While CPU sharing mechanisms are reasonably well understood, network I/O resources involve a complex range of interactions that are harder to predict and control. This makes meeting latency requirements for distributed real-time applications in the presence of competing non-real-time applications challenging.

In non-virtualized hosts running standard Linux, a queueing discipline (QDisc) layer implements traffic control functionality, including traffic classification, prioritization and rate shaping. Several different queueing disciplines are provided in the Linux kernel. In particular, disciplines such as Prio [2] and FQ_CoDel [3] prioritize network traffic and can, therefore, effectively protect latency-sensitive applications from contention with non real-time traffic. These queueing disciplines are also used in virtualized hosts based on Xen [4], a widely-used open-source virtualization platform. From now on, following Xen's terminology, we use the term *domain* in place of VM. Xen employs a manager domain called domain 0 (dom0) to manage the other domains (guest domains). Dom0 is also responsible for processing I/O operations (including network traffic) on behalf of the guest domains. By default dom0 runs a Linux kernel. Virtualization, however, may introduce priority inversions in some network components and between the transmission (TX) and reception (RX) routines. Those limitations are unaccounted for by the standard Linux queueing disciplines, and therefore have the potential to make queueing delays unpredictable for latency-sensitive traffic in Xen.

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 evaluates latency performance in Xen in the presence of diverse traffic patterns, and identifies the impact of different components in the network path of Xen; (2) it evaluates standard Linux traffic control policies in Xen, and highlights limitations that arise from Xen's network architecture; and (3) it introduces *VATC*, a *Virtualization-Aware Traffic Control* scheme in which the network streams in Xen are prioritized with a one-thread-per-priority communication architecture, thereby offering greater latency control and predictability for latency-sensitive applications.

II. 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.

In this section, we first review standard Linux packet transmission and reception routines, which have been stable since version 2.6. In contrast, the architecture of the virtualization-related components of Xen has seen much change across versions of Linux. Hence, we select two representative versions,

978-1-4799-8603-3/15/$31.00 ©2015 IEEE

Linux 3.10 and Linux 3.18 (the current version), identify limitations in both versions, and explore their implications for latency guarantees. As was mentioned before, addressing those issues is the main motivation behind the design of VATC.

A. Network Stack in Standard Linux

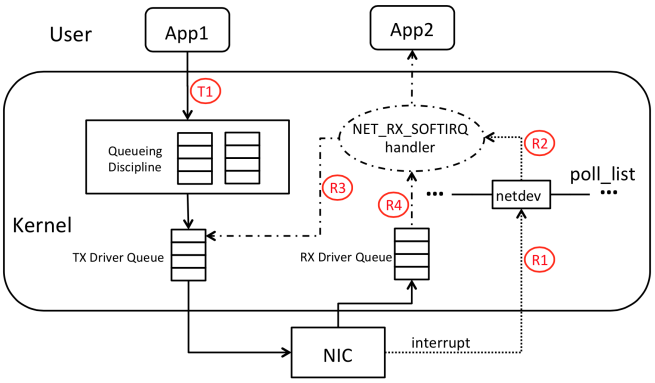


Fig. 1: Transmission/Reception in Standard Linux

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 Standard Linux: Figure 1 shows network transmission and reception routines in standard Linux. In standard Linux, packets from applications are processed by the network stack in the Linux kernel. Because the virtualization-extensions of Linux only change the link layer, we omit session, transport and network layers in the figure. Packets are enqueued into the appropriate queueing discipline (QDisc) queue(s) in the link layer, which is where Linux implements traffic control. The TX driver queue, also known as the ring buffer, is a FIFO queue that works closely with the NIC.

Queueing Discipline: The QDisc layer implements traffic classification, prioritization and rate shaping. QDisc settings can be configured through the TC command. By default, Linux uses pfifo_fast as the queueing discipline for traffic control.

Depending on the QDisc configuration, Linux can prioritize packets and reduce queueing delay for latency-sensitive applications. Prio [2] is a queueing discipline that has one queue per priority. It 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 queueing delays. FQ_Codel [3] [5] is another queueing discipline that works to reduce queueing 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 queueing delays to latency-sensitive applications with low throughput.

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 arises when too many large packets are forwarded to the NIC and the hardware is not capable of processing them fast enough.

There is typically a limit to the size of TX driver queue, which controls the number of pending packets. However, this control is insufficient to prevent congestion when the bulk of the NIC traffic consists of large packets. This limitation has been addressed in recent Linux kernels (after 3.3), by the introduction of a Byte Queue Limit (BQL) [6] policy, which limits the number of *bytes* in the TX driver queue of the NIC. In cooperation with the QDisc layer, BQL can greatly reduce the queueing delay in the TX driver queue, even in the presence of large packets. With BQL, the size of the TX driver queue is limited dynamically, based on the traffic mode and throughput. Once the queue size hits the limit, the QDisc layer holds or drops subsequent packets.

In most NIC drivers, when packets are successfully sent by the NIC, a TX completion interrupt is triggered. The interrupt handler puts a netdev (a software data structure representing the NIC driver) device into the poll_list, which is a per_CPU data structure in Linux. At the end of the interrupt handler, a software interrupt, called NET_RX_SOFTIRQ, is raised, whose handler 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. When the netdev device is fetched, the NET_RX_SOFTIRQ handler invokes the NAPI poll() method of the NIC driver. Depending on the NIC driver, the NAPI poll() method may perform different actions. In the NIC driver used in our experiments, the NAPI poll() method cleans up the TX driver queue and receives packets from RX driver queue. Other NICs have separate TX and RX interrupts.

The TX completion interrupt handler cleans up the TX driver queue, while the RX interrupt handler raises NET_RX_SOFTIRQ. The NAPI poll() method of these NIC drivers only does packet reception. 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. 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, too frequent interrupts can generate heavy CPU workloads and adversely impact progress of the 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 queueing delays there. Several NIC driver vendors offer dynamic interrupt throttle rates, which adjust the interval value on the fly based on whether the traffic is low-latency or bulk. Our evaluation in Section IV examines the effect of different configurations of these settings.

2) Reception Routine in Standard Linux: Figure 1 also shows network reception in standard Linux, in which packet arrivals trigger hardware interrupts, and the interrupt handler then puts the netdev (the same network device mentioned above) into the poll_list and raises NET_RX_SOFTIRQ. When the NET_RX_SOFTIRQ handler fetches the netdev and invokes the corresponding NAPI poll() method, packets are

delivered from the RX driver queue to the upper layer. 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.

B. Network Stack Modifications in Xen

Recall that Xen relies on a manager domain, dom0 (domain 0), to handle inter-domain and network traffic. A Linux system is installed in dom0. Because the virtualization-extensions of Xen have been merged into the mainline of the Linux kernel (since version 3.0), the network stack in dom0 is similar to that of standard Linux. The core network stack in dom0 has been stable, but the virtualization-related components have significantly evolved across versions of Linux. Figures 2 and 3 show the Xen transmission and reception routines in two representative versions, *dom0-3.10* (dom0 built on Linux 3.10) and *dom0-3.18* (dom0 built on Linux 3.18), respectively.

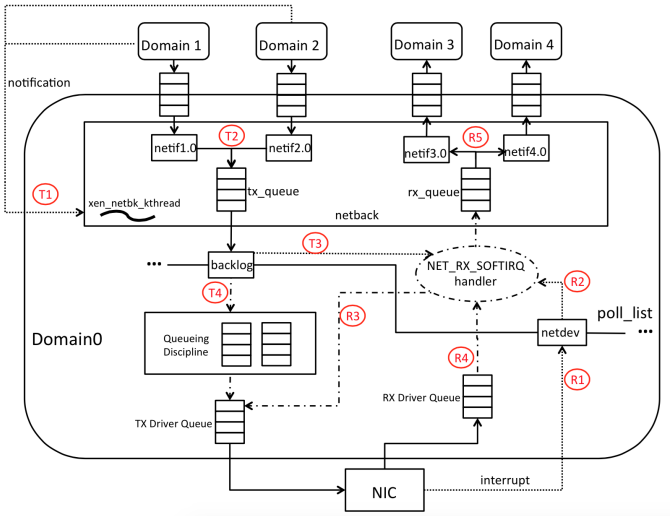


Fig. 2: Transmission/Reception in Dom0-3.10

Transmission:
T1: notification handler triggers xen_netbk_kthread
T2: xen_netbk_kthread delivers packets from netif(s) to backlog device
T3: xen_netbk_kthread raises NET_RX_SOFTIRQ
T4: NET_RX_SOFTIRQ handler delivers packets from backlog 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 rx_queue and triggers xen_netbk_kthread
R5: xen_netbk_kthread delivers packets from rx_queue to guest domain(s)

1) Network Stack in Dom0-3.10: In dom0-3.10, there are two virtualization-related components: the netif and netback devices. Each guest domain has a corresponding netif device in dom0. Traffic transmission/reception between the guest domain and its netif device is realized by a memory copy or remap based on the buffer between them. All netif devices are then included in the netback device, which works as a gateway. A kernel thread, named xen_netbk_kthread, services all tasks related to the netback device.

Transmission Routine in Dom0-3.10: When a guest domain puts packets into the buffer connected to its corresponding netif device, it also sends an event notification to dom0. In dom0, the notification handler triggers the xen_netbk_kthread to receive packets. Once the xen_netbk_kthread is scheduled, packets pending in the buffer between netif devices and guest domains are first enqueued into the tx_queue of the netback device in a *round-robin* order. When the enqueue task finishes, the xen_netbk_kthread dequeues the packets from the netback tx_queue in FIFO order. The dequeued packets are forwarded to the backlog device. When the packets arrive at the backlog device, the xen_netbk_kthread inserts the backlog device into the poll_list, and raises a NET_RX_SOFTIRQ. When the NET_RX_SOFTIRQ handler is scheduled and the backlog device is fetched, packets go through the same routine as in standard Linux. A setting specific to dom0-3.10 is that each time the xen_netbk_kthread raises a NET_RX_SOFTIRQ, it then immediately executes the softirq processing function. This ensures that in most cases only a few packets are ever pending in the backlog device. For compatibility, no modifications are made to the traffic control policies (queueing disciplines).

Reception Routine in Dom0-3.10: When a packet arrives, a hardware interrupt is triggered, whose handler inserts the netdev device into the poll_list and raises a NET_RX_SOFTIRQ. Next, in the context of its handler[1], packets pending in the RX driver queue of the NIC are forwarded to the rx_queue of the netback device. When packets are enqueued, the handler also triggers the xen_netbk_kthread. When the xen_netbk_kthread is scheduled, packets in the rx_queue are distributed to destination domains through the netif devices.

A special feature of the netback device is that the enqueue/dequeue operations of the tx_queue and the dequeue operation of the rx_queue of the netback device are done in the context of the xen_netbk_kthread. The length limit of both the tx_queue and the rx_queue is 256 by default. The xen_netbk_kthread runs a loop that first deals with the rx_queue. Only when all the packets pending in the rx_queue are delivered, does the xen_netbk_kthread perform the enqueue/dequeue operations of the tx_queue.

2) Network Stack in dom0-3.18: In dom0-3.18, the netif and netback devices are replaced by a new network backend device, named *vif* (virtual interface). Each vif device shares a buffer with its corresponding guest domain. The main difference between the netif and vif devices, is that the vif device does not need to coordinate with the netback device for transmission. Furthermore, it also has a separate rx_queue and a dedicated kernel thread (rx_kthread) for reception.

Transmission Routine in Dom0-3.18: 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 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.

[1]when dom0 is busy with transmission, NET_RX_SOFTIRQ handler function is executed by xen_netbk_kthread.

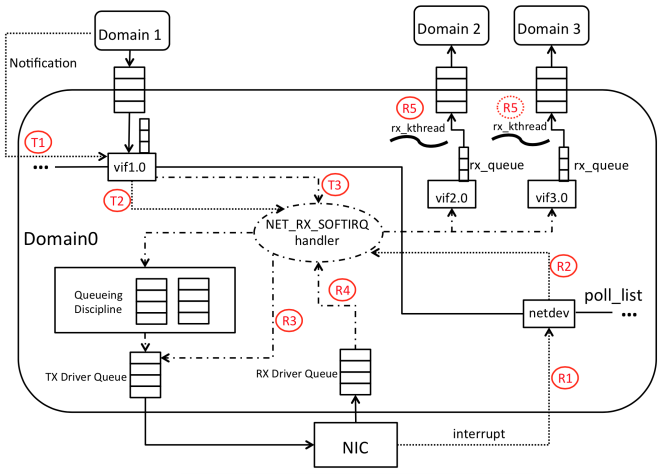


Fig. 3: Transmission/Reception in Dom0-3.18

Transmission:
T1: notification handler inserts vif into the poll_list and
T2: raises NET_RX_SOFTIRQ
T3: NET_RX_SOFTIRQ handler delivers packets from vif(s) 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 vif(s), and triggers rx_kthread(s)
R5: rx_kthread(s) delivers packets from rx_queue(s) to guest domain(s)

Reception Routine in Dom0-3.18: 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. This is in contrast to delivering them to the rx_queue shared by all netif devices in dom0-3.10. 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 the softirq processing finishes, which can cause delays, as was discussed earlier.

C. Traffic Control Limitations in Xen

We now summarize limitations of Xen traffic control mechanisms, some of them common to both dom0-3.10 and dom0-3.18 and others specific to one version. Those limitations, when combined with contention for resources such as network bandwidth or CPU, can produce unexpected delays. In Section IV, we investigate a number of scenarios under which resource contention is present, and where those limitations may affect latency-sensitive applications.

Limitation 1: Priority Inversion between Transmissions

In dom0-3.10, because the netback tx_queue is a simple FIFO queue shared by all guest domains, packets from latency-sensitive domains can be delayed easily by a large number of packets from bandwidth-intensive domains. Meanwhile, the waiting time in the netif device also increases because the enqueue operation of the netback tx_queue is only invoked after delivering all the packets in it (worst case is 256).

In dom0-3.18, when both latency-sensitive domains and other interfering domains are transmitting packets, 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. As a result, priority inversion between domains still exists in dom0-3.18. Note that the default quantum for each network device in the poll_list is 64 (packets). Reducing the quantum can resolve this priority inversion. However, as we shall see next, there is another limitation that this approach cannot resolve.

Limitation 2: Priority Inversion between Transmission and Reception

In dom0-3.10, the xen_netbk_kthread invokes transmission and reception functions in a round-robin order. Hence, even if priorities are enforced in the tx_queue and rx_queue, respectively, priority inversion can still arise *between* the two. For example, while a latency-sensitive domain is receiving packets from another physical host, bandwidth-intensive domains may at the same time be transmitting packets. It is therefore possible that when packets destined for the latency-sensitive domain are put into the rx_queue by the NET_RX_SOFTIRQ handler, the xen_netbk_kthread is busy processing the tx_queue. As a result, latency-sensitive packets are dequeued only after all the packets in the tx_queue (up to 256 packets) are delivered. This can result in a long delay in the rx_queue.

In dom0-3.18, when the NET_RX_SOFTIRQ handler forwards a packet to the vif device, it wakes up the corresponding rx_kthread to do the follow-up tasks. However, the rx_kthread can only be scheduled after the softirq processing finishes. In CPU-bound situations with many (non real-time) domains sending packets at a high enough rate, NET_RX_SOFTIRQ can be raised frequently (by the notification handler), so that the handler continuously services the poll_list, which can delay the running of rx_kthreads for a long time. In CPU-bound scenarios in dom0-3.18, interferences can, therefore, still arise between transmission and reception. Simply reducing the quantum for each network device in poll_list cannot resolve this priority inversion, because the duration for which softirq processing runs doesn't depend on the quantum value.
Note that in dom0-3.10, there is no interference between the softirq processing and the xen_netbk_kthread, because, as mentioned in Section II-B1, the NET_RX_SOFTIRQ handler is also executed in the context of the xen_netbk_kthread.

Summary: When compared to dom0-3.10, dom0-3.18 eliminates the tx_queue and rx_queue shared across guest domains, and consequently removes this source of priority inversions. However, the introduction of per-domain vif devices brings a new source of priority inversions, because of the round-robin order with which vif devices are serviced in the poll_list. Additionally, in dom0-3.10, TX and RX tasks are jointly handled by a single kernel thread (xen_netbk_kthread), which can itself contribute to priority inversions. This limitation is not present in dom0-3.18, but interference between the softirq handler (for transmission) and rx_kthread can have a similar impact in CPU-bound situations (see Section IV-B). The existing queueing disciplines that Xen inherits from Linux cannot address these limitations, because the latency problems they introduce are in the virtualization-related components.

III. Design and Implementation

Our goal is to improve Xen's ability to protect latency-sensitive applications. For that purpose, we introduce a

virtualization-aware traffic control (VATC) architecture, which mitigates the limitations identified in dom0.

The simplest way to mitigate priority inversions among transmission flows is to extend priority awareness to the netback tx_queue (in dom0-3.10) or the vif devices in the poll_list (in dom0-3.18). However, priority inversions between transmission and reception are due to the coarse sharing of the xen_netbk_kthread (in dom0-3.10) and the interference between softirq processing and rx_kthread (in dom0-3.18). As a result, rather than implementing one queueing discipline for packet transmission in the virtualization-related components, 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 concept by assigning the network traffic of high-priority domains to high-priority kernel threads, and the network traffic of low-priority domains to low-priority kernel threads.

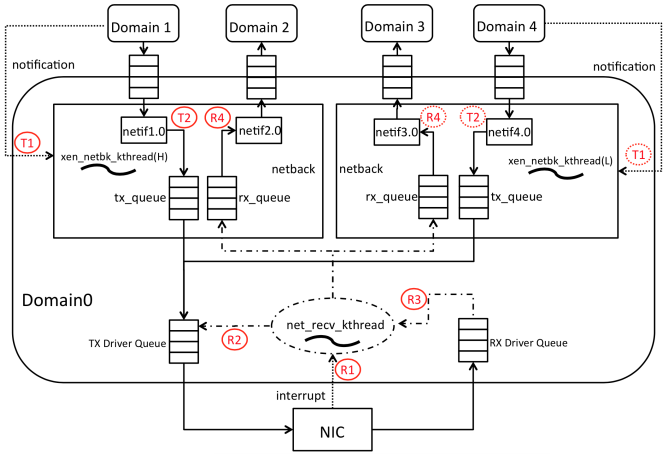


Fig. 4: VATC: Virtualization-aware Traffic Control

Transmission:
T1: notification handler(s) trigger xen_netbk_kthread(s)
T2: xen_netbk_kthread(s) deliver packets from netif(s) to TX driver queue
Reception:
R1: interrupt handler triggers net_recv_kthread
R2: net_recv_kthread cleans up TX driver queue and
R3: delivers packets from RX driver queue to rx_queue(s) and triggers xen_netbk_kthread(s)
R4: xen_netbk_kthread(s) deliver packets from rx_queue(s) to guest domain(s)

Figure 4 illustrates the overall structure of VATC[2]. In VATC, there are now multiple netback devices (instead of one shared netback device), and correspondingly multiple xen_netbk_kthreads to handle packet transmissions and receptions to/from different domains. These xen_netbk_kthreads are configured with different priorities. Guest domains with the same priority (same latency requirement) share the same netback device and xen_netbk_kthread. All the xen_netbk_kthreads are scheduled under a SCHED_FIFO policy. The number of netback devices (xen_netbk_kthreads) can be configured based on the number of priority levels needed.

[2]By replacing the network backend driver, VATC works in both dom0-3.10 and dom0-3.18.

For clarity, Figure 4 uses two priority levels. Domain 1 and domain 2 are running real-time (latency-sensitive) applications and are assigned to a high-priority thread. Domain 3 and domain 4 are low-priority domains running bandwidth-intensive (non real-time) applications. The high-priority xen_netbk_kthread(H) handles the network traffic of the high-priority domains, and the low-priority xen_netbk_kthread(L) handles traffic of the low-priority domains. The net_recv_kthread, which is triggered by TX completion and the RX interrupt handler, has the highest priority. It cleans up all the packets that have been transmitted from the TX driver queue and processes 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 VATC. Next, we review packet transmission and reception in VATC, as well as their interactions.

1) Packet Transmission in VATC: When a high-priority domain has packets to send, it notifies dom0. The notification handler in dom0 then triggers the corresponding high-priority xen_netbk_kthread, which can preempt lower-priority threads. Packets from the high-priority domain are first enqueued 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 xen_netbk_kthread to resume (if the new queue size is under the BQL limit). If multiple xen_netbk_kthreads are suspended, they will resume one by one, based on their priorities. In network-contention situations, this design protects real-time traffic in a similar way as some priority-based queueing disciplines, such as Prio. Because of this, transmission in VATC can bypass the QDisc layer. Note that each xen_netbk_kthread can process packets in the tx_queue in FIFO order because the source domains of these packets have the same priority. After packets are put into the TX driver queue, they are subsequently transmitted by the NIC. Once there are no more packets from any high-priority domain to enqueue and the tx_queue is empty, the high-priority thread stops, allowing a lower-priority thread to run.

In dom0-3.10, the xen_netbk_kthread raises NET_RX_SOFTIRQ and executes the softirq in its own context, and therefore also has to handle all the pending softirqs. VATC removes softirqs because they can lead to priority inversion when a high-priority thread has to process softirqs raised by low-priority threads. VATC therefore handles packet TX/RX entirely in the xen_netbk_kthreads and the net_recv_kthread. To ensure the implementation is thread-safe, VATC deals with all the critical sections (e.g., updating forwarding table, or putting packets into TX driver queue) in the same way as the softirq processing in dom0-3.10 (which is also executed in thread context).

2) Packet Reception in VATC: When packets arrive at a virtualized host, the RX hardware interrupt handler wakes up the net_recv_kthread, instead of the NET_RX_SOFTIRQ, to process them. Once the net_recv_kthread is scheduled, it picks up packets from the RX driver queue and forwards them to the rx_queue of the destination netback device. For the rx_queue of the netback device, each enqueue operation wakes

up the corresponding xen_netbk_kthread, which will be scheduled after the net_recv_kthread finishes its work. If multiple xen_netbk_kthreads are woken up by the net_recv_kthread, they will be scheduled based on their priorities. When a xen_netbk_kthread is scheduled, it delivers packets from the corresponding rx_queue to the destination domains.

3) Interference between Transmission and Reception: Recall that in dom0-3.10, interference between transmission and reception arises because both are processed by the xen_netbk_kthread in a round-robin order (with a quantum of up to 256 packets). This coarse sharing results in the transmission (or reception) of real-time traffic potentially being delayed by the reception (or transmission) of non-real-time traffic. In dom0-3.18, the rx_kthread (for reception) can be preempted by the NET_RX_SOFTIRQ handler (for transmission). In VATC, the transmission/reception of real-time traffic is handled by high-priority kernel thread(s). Hence, the interference from either the transmission or reception of non-real-time traffic (handled by lower-priority kernel threads) is greatly reduced.

IV. Evaluation

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 in Xen with dom0-3.18, and under existing Xen traffic control mechanisms[3], i.e., Prio and FQ_CoDel in both dom0-3.10 and dom0-3.18.

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 Xen 4.3 installed, on which dom0 is a 64-bit CentOS built on Linux kernel 3.10.43 or 3.18.0. Host 0 acts as the host server. Five other physical machines, hosts 1 to 5, run standard 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 [8], [9] 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. Figure 5 offers a schematic overview of the testbed.

In host 0, dom0 is given a dedicated physical CPU core. This is common practice to handle communication and interrupts [10], [11], and is also recommended by the Xen community to improve I/O performance [12]. 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 VM scheduler. In our setup, domain 1 is the latency-sensitive domain and domains 2 to 5 are interfering domains.

[3]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.

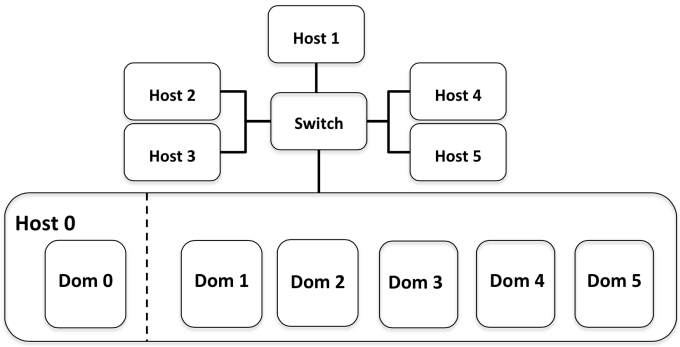


Fig. 5: Testbed Setup

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.

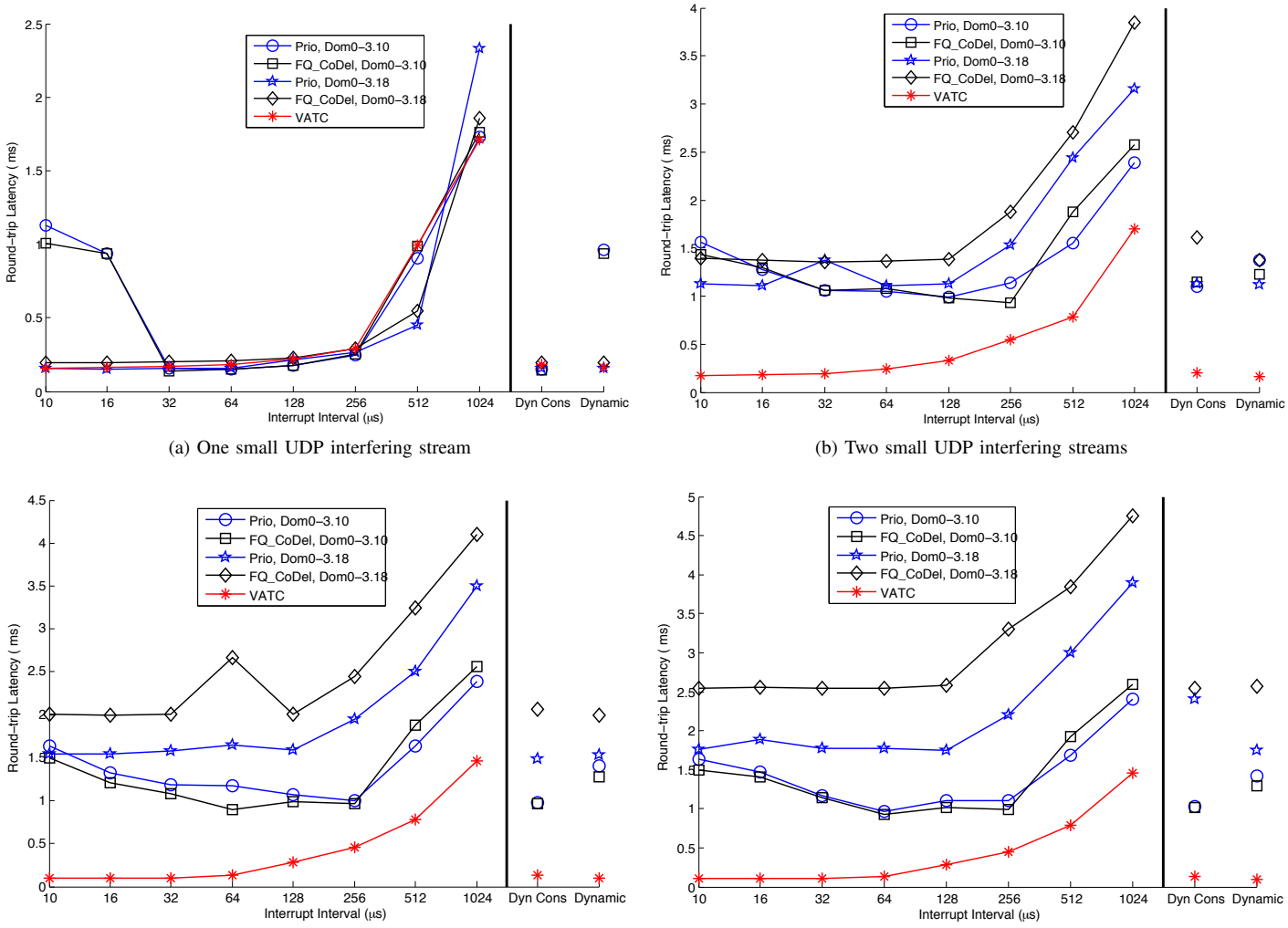
The round-trip latency between domain 1 and host 1 is measured as follows. Domain 1 pings (with ICMP packets) host 1 every 10 ms, and host 1 replies back. This traffic pattern seeks to emulate the behavior of common periodic real-time applications. Each experiment records latency values for 1,000 ICMP 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 [13] 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.

A. Evaluation Scenarios

Latency of high-priority (latency-sensitive) traffic is measured for different scenarios that can give rise to resource (CPU or network) contention in dom0. CPU contention can occur when low-priority domains are sending many small packets. In those scenarios, the TX handler (xen_netbk_kthread in dom0-3.10 or NET_RX_SOFTIRQ handler in dom0-3.18) is frequently triggered. This 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 *CPU-contention* scenarios, whether using dom0-3.10 or dom0-3.18, the two *Limitations* identified in Section II, introduce long queueing delays in virtualization-related network components (netif, tx_queue, rx_queue in dom0-3.10, or vif, QDisc, rx_queue in dom0-3.18). *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, long delays happen in the hardware, and as a result, VATC and existing traffic control schemes perform similarly.

We explore CPU-contention and Network-contention scenarios in Section IV-B and Section IV-C, respectively, and throughout the experiments use the terms *small packet stream* and *large packet stream* to identify the packet size of traffic



(a) One small UDP interfering stream

(b) Two small UDP interfering streams

(c) Three small UDP interfering streams

(d) Four small UDP interfering streams

Fig. 6: CPU-contention – Increasing number of interfering streams: Median high-priority latency

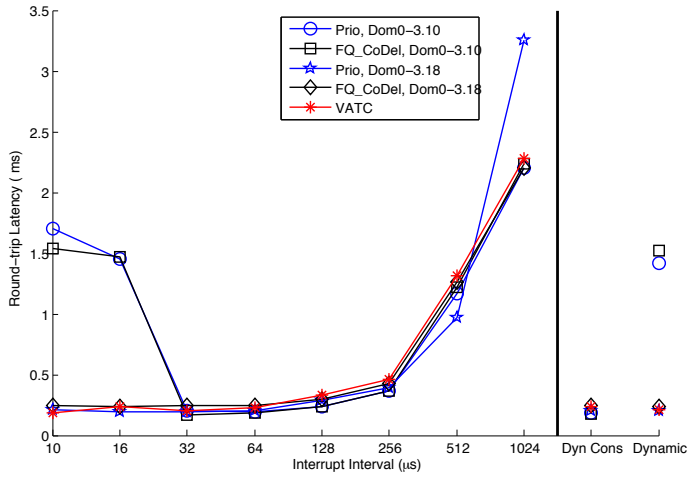
from low-priority domains, i.e., 1 byte for small packet streams and $1,500$ bytes for large packet streams. Small packet streams can lead to CPU contention because of high processing overhead, while large packet streams can cause network contention because of high bandwidth consumption.

B. CPU-Contention Scenarios

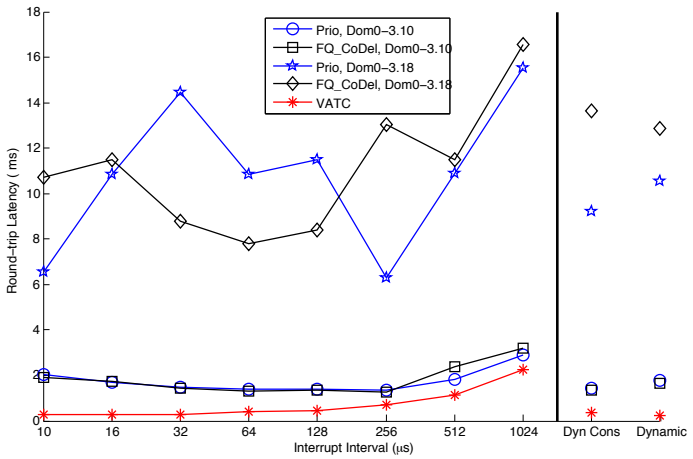
In this section, we evaluate the latency of high-priority traffic in the presence of interfering low-priority streams consisting of small UDP packets. Figure 6 and Figure 7 show the round-trip latency (median and 95th percentile) of ICMP packets from the high-priority domain for different numbers of interfering low-priority small UDP streams and different interrupt intervals (from 10μs to 1024μs and using the dynamic and dynamic conservative modes).

1) Impact of Interrupt Interval: In order to isolate the impact of different interrupt intervals, we focus on the case of one interfering stream (Figures 6a and 7a). We note that because packets are small, the dynamic conservative mode and the dynamic mode tend to default to setting the interrupt interval to the lower bound of their range, i.e., 50μs and 14μs, respectively.

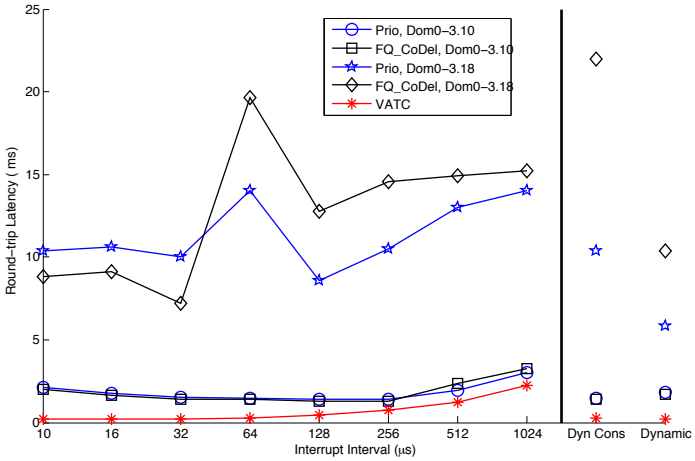
When the interrupt interval is small (10μs to 16μs, or dynamic), high-priority packets experience long delays under both Prio and FQ_CoDel in dom0-3.10. The reason is frequent preemption by the NET_RX_SOFTIRQ handler (issued by TX completion interrupts), which results in the xen_netbk_kthread not getting enough CPU resources. Note that some NIC drivers clean up TX driver queue in the context of the hardware interrupt handler, but this mechanism still preempts the xen_netbk_kthread and may result in long delays. This problem is absent in dom0-3.18, which removed the shared tx_queue of dom0-3.10 and replaced it with separate vif devices for each domain. The vif devices are put into the poll_list that is serviced by the NET_RX_SOFTIRQ handler. Hence, frequent invocations of the handler function contribute to more frequent servicing of vif devices, especially with only one competing vif device (interfering domain) and one netdev device (for TX driver queue clean-up) in the poll_list. VATC can be seen to perform similarly to dom0-3.18, and the reason is that even though the xen_netbk_kthread is frequently preempted by the net_recv_kthread, there is no shared queue (like the netback tx_queue in dom0-3.10) where low-priority packets can accumulate and delay high-priority packets.



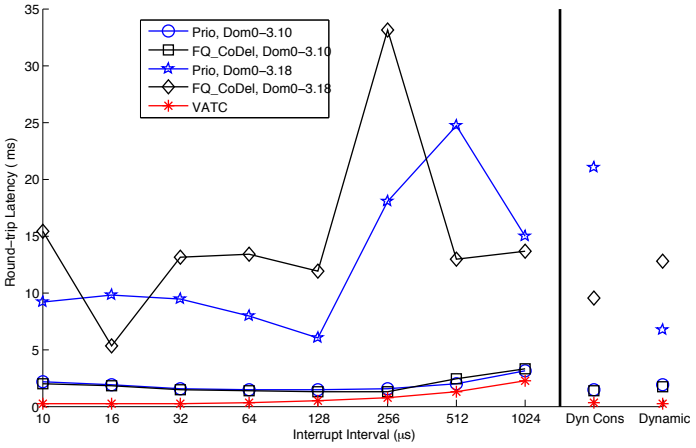
(a) One small UDP interfering streams



(b) Two small UDP interfering streams



(c) Three small UDP interfering streams



(d) Four small UDP interfering streams

Fig. 7: CPU-contention – Increasing number of interfering streams: Tail (95th percentile) high-priority latency

To validate the above conclusions, we performed a detailed analysis of how different dom0 components contribute to the overall latency of high-priority packets.

In both dom0-3.10 and VATC, queueing may arise in:

- netif: packets wait in netif while the xen_netbk_kthread dequeues packets from the tx_queue or rx_queue;
- tx_queue: this (FIFO) queue is shared by packets from all domains;
- QDisc: when there is congestion in the TX driver queue, packets are kept waiting in the QDisc layer;
- rx_queue: when the xen_netbk_kthread is servicing the tx_queue, the response packets of the high-priority ICMP stream wait in the rx_queue.

In dom0-3.18, queueing 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 8 reports the delay contributions of individual components for high-priority packets in *Prio–dom0-3.10*, *Prio–dom0-3.18* (the results with FQ_CoDel is similar to those with Prio), and VATC for an interrupt interval value of 10μs.

The results confirm our earlier hypothesis, namely, that because of dom0-3.10's reliance on a shared netback tx_queue, a large number of (low-priority) packets can accumulate there, and in the process introduce large delays in the netif and netback tx_queue (*Limitation* 1), as well as the netback rx_queue (*Limitation* 2). Because neither dom0-3.18 nor VATC rely on such a shared queue (the tx_queue of the high-priority traffic is separate with its own high-priority thread in VATC), those delay contributions are eliminated. We note, though, the relatively long tail of rx_queue latency for dom0-3.18. It is because the softirq processing (triggered by the TX notifications from the interfering domains) can delay the rx_kthread. As we shall see in the next set of experiments, this effect becomes more pronounced as the number of interfering domains increases. Increasing the interrupt interval (beyond

32μs) allows the xen_netbk_kthread of dom0-3.10 to get enough CPU resources and eliminate most of the queueing delays in those components, so that it then performs as well as dom0-3.18 and VATC. As the interrupt interval value increases further, the pending time in the QDisc layer is larger because the TX driver queue is cleaned up less frequently and becomes a bottleneck for all schemes. This explains the steady and comparable increases in latency all configurations experience in Figure 6a and Figure 7a.

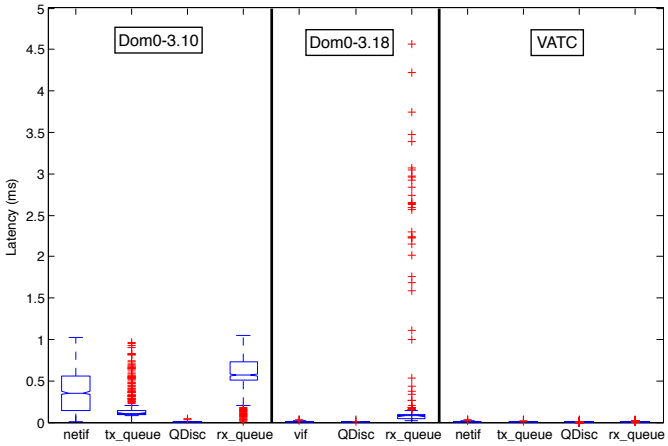


Fig. 8: CPU-contention – One interfering stream: *dom0* high-priority latency (10μs interrupt interval)

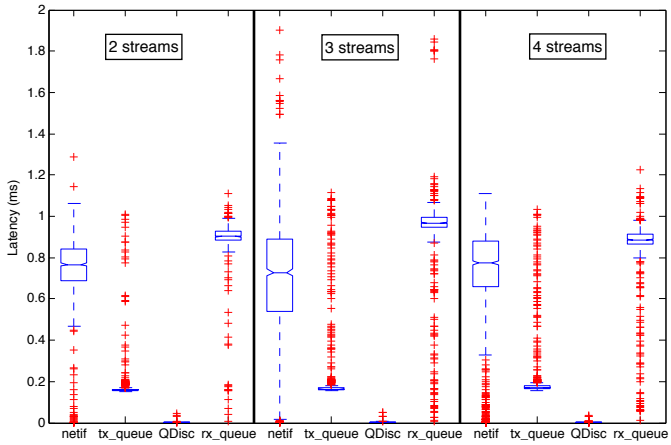


Fig. 9: CPU-contention – Increasing interfering traffic and # streams: *dom0-3.10* high-priority latency (10μs interrupt interval)

2) Impact of the Number of Streams: The growth in the number of cores in a single host means that multiple guest domains can coexist within one host. Supporting latency differentiation with more than one domain is, therefore, an important scalability concern. In this section, we evaluate the latency of the high-priority (ICMP) stream as the number of interfering domains increases.

Figures 6b, 6c, and 6d show the median latency of high-priority packets with 2, 3, or 4 interfering low-priority small UDP streams, while Figures 7b, 7c, and 7d show tail latency for the same configurations. Additionally, Figures 9 and 10 detail the delay contributions of individual components in dom0. We note from the figures that VATC's latency performance

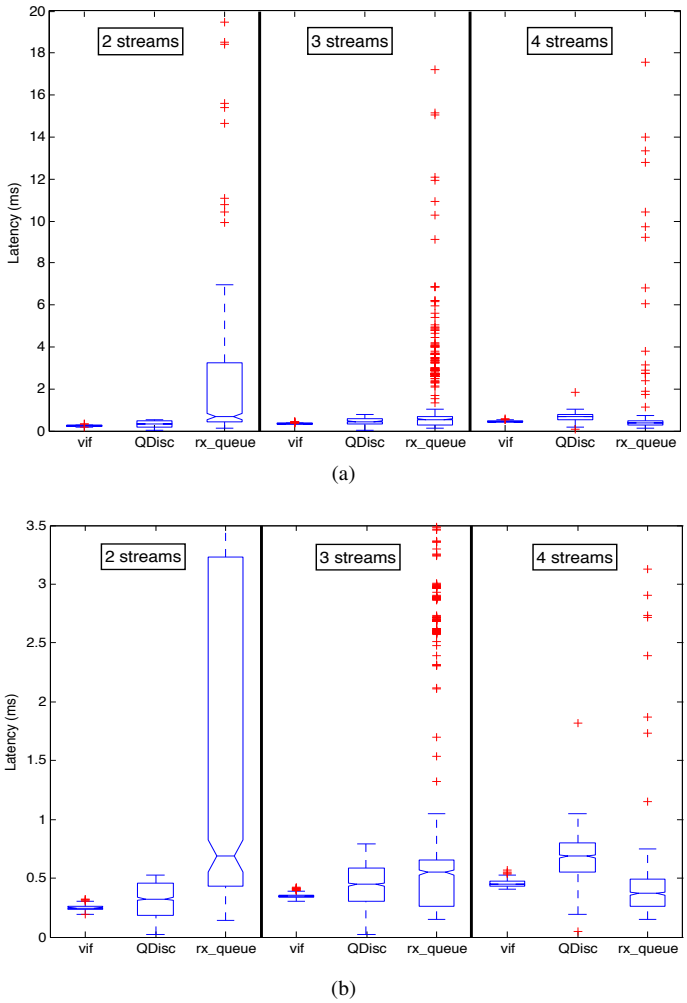


Fig. 10: CPU-contention – Increasing interfering traffic and # streams – *dom0-3.18* high-priority latency (10μs interrupt interval, (b) presents a subset (values between 0 ms and 3.5 ms) of the results of (a))

is unaffected by the number of interfering streams, as the high-priority xen_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.

The main difference among the results of Figures 6a to 6d is the significant degradation in latency experienced in dom0-3.18 as the number of interfering domains/streams increases. This degradation is such that dom0-3.10 eventually outperforms dom0-3.18. This, however, does not mean that dom0-3.10 is immune to the number of interfering streams. As Figures 6b and 7b (two interfering streams) illustrate, VATC now outperforms dom0-3.10 for all interrupt interval values, including the range 32μs to 1024μs where dom0-3.10 does not suffer from frequent preemptions by the softirq handler. The main reason is that with two interfering low-priority streams, the packet arrival rate (at dom0) is high enough to overload the packet processing capacity of the xen_netbk_kthread. As a result and as shown in Figure 9, longer queueing delays now arise in the netif (*Limitation* 1) and in the rx_queue (*Limitation* 2). The impact of this effect saturates quickly. Note that two interfering streams are enough to overload the

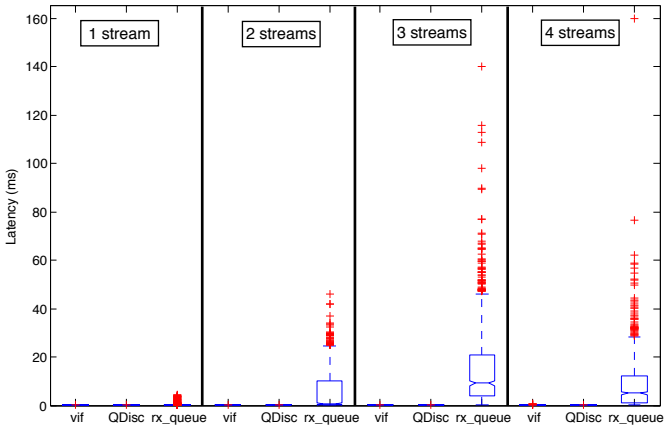


Fig. 11: CPU-contention – Constant interfering traffic, increasing # streams: *dom0-3.18* high-priority latency (10μs interrupt interval)

xen_netbk_kthread, and furthermore the size of the shared netback rx_queue is limited to 256. As a result, dom0-3.10 only experiences limited changes as the number of interfering streams increases further.

The situation is different for dom0-3.18, which experiences increases in latency (median and tail) with the number of interfering streams. Figure 10 sheds some light on this behavior. As the number of interfering domains increases, so does the number of vif devices inserted in the poll_list, which 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 competing vif devices (in the poll_list) increases, the netdev device is serviced less frequently. This results in higher congestion in the TX driver queue and, therefore, longer queueing delays in the QDisc layer. In some other NIC drivers that clean up TX driver queue in hardware interrupt handler, the vif devices in poll_list won't delay the clean-up of TX driver queue, thus the queueing delay in QDisc layer may be reduced.

Both of those contributions to higher latency can be attributed to *Limitation 1*, but *Limitation 2* can be seen to have an even more pronounced effect. The addition of a second interfering stream (see Figure 10b) significantly affects 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[4]. This can then result in the rx_kthread being delayed for an unpredictably long time as illustrated in Figure 7b which captures the tail of the delay distribution in dom0-3.18. For purposes of illustration, the 95th latency percentiles are 8 and 40 times higher in *Prio–Dom0-3.10* and *Prio–Dom0-3.18*, respectively, than in VATC. Interestingly though, this trend somewhat reverses as the number of interfering streams increases further beyond 2, because the NET_RX_SOFTIRQ is raised less frequently as more low-priority streams are added. In our experiments, a large fraction of the softirqs are raised by notification handlers from guest domains. Because dom0's CPU is overloaded, not all low-priority packets can be serviced in time, and a backlog of packets builds-up in the buffers between vif devices and the corresponding (low-priority) guest domains. This backlog prevents the corresponding guest domains from putting more packets into the buffer, and thus no new notifications are issued to dom0 until the buffer is refreshed.

The next experiment explores further the impact of notification frequency on latency performance in dom0-3.18. The results are shown in Figure 11, which parallels Figure 10 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 10 and 11 (the worst delay observed in the experiment of Figure 11 was 160 ms!). Of note in Figure 11 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.

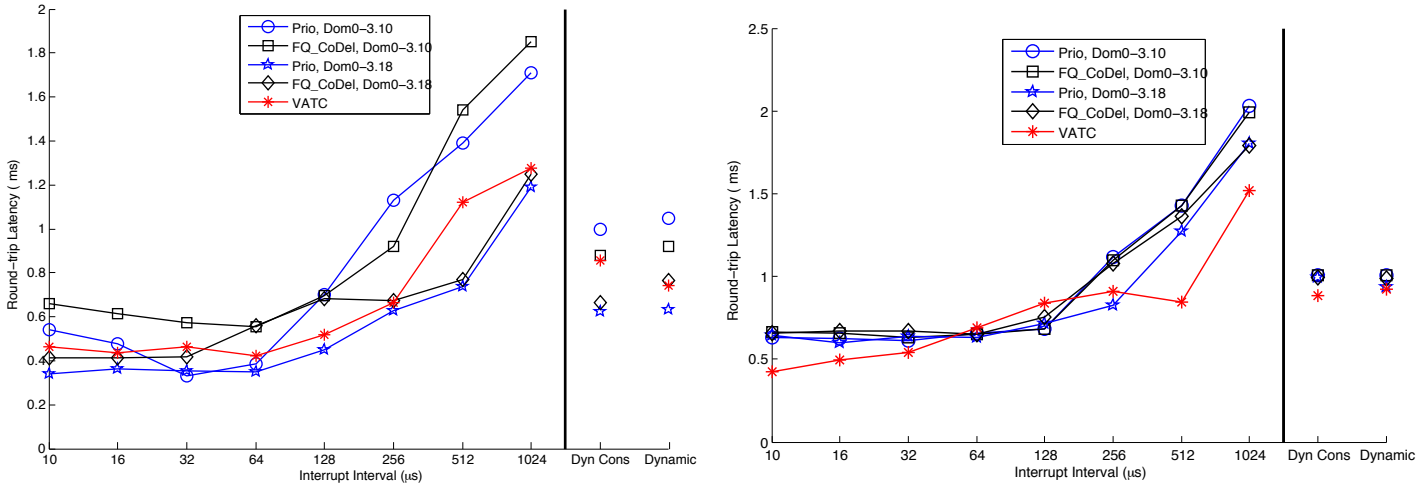
Summary:

- In dom0-3.10, small interrupt intervals can overload the xen_netbk_kthread and cause high latency. Dom0-3.18 addresses this problem by eliminating shared packet queues.
- Neither dom0-3.10 nor dom0-3.18 can protect latency-sensitive traffic when dom0 is overloaded by small packet streams. As the interference becomes larger, the queueing delay in dom0-3.10 is bounded, while the delay in dom0-3.18 worsens because vif devices share the poll_list and the softirq handler delays reception.
- *Limitation* 1 and *Limitation* 2 are both present in dom0-3.10 and dom0-3.18, so scenarios exist where traditional Linux traffic control mechanisms such as Prio and FQ_CoDel cannot protect latency-sensitive applications. VATC overcomes these limitations by dedicating a netback device and a prioritized kernel thread to each priority level, so that real-time streams are protected from interference from other traffic.

C. 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 interfering

[4]NIC drivers that clean up the TX driver queue in the hardware interrupt handler without, therefore, raising NET_RX_SOFTIRQ might 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.



(a) UDP

(b) TCP

Fig. 12: Network-contention – One interfering stream (UDP or TCP): Median high-priority Latency (ms)

streams with large packet sizes, i.e., $1,500$ bytes (MTU) for each UDP or TCP packet in the network link. In our experiments a Netperf_stream application runs in domain 2 and sends a stream of large packets (UDP or TCP) to host 2 which is 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 to the upper bound of their range, i.e., 250μs. As a result, their performance is, therefore, now aligned with that of a static interrupt interval set to approximately that value.

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

Figure 12a and Figure 12b respectively show the median latency (tail latency is close to the median value) of high-priority packets under the impact of one large-packet UDP stream or one large-packet TCP stream. We see that the round-trip latency is penalized as the interrupt interval increases. Because the majority of the latency happens in hardware or is caused by the congestion there, VATC offers little or no improvement in latency.

In a few cases (sending a large-packet TCP stream with interrupt intervals between 10 μs to 32 μs), the overhead of VATC contributes to a higher CPU utilization. However, the increase is limited and was not found to affect overall system throughput across the range of experiments that were conducted.

V. 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 [14] can reserve I/O resources (e.g. network bandwidth) for business-critical traffic based on user-defined network resource pools [15]. In Windows Server 2012 R2 [16], Hyper-V QoS [17], [18] 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 [19] is another virtualization platform based on Linux. It creates multiple vhost threads to handle traffic from different guest VMs. However, different vhost threads are 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, the vhost thread servicing real-time traffic can be preempted by threads for non-real-time traffic or softirq handlers.

Xu et al. [9] 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. [9] also optimize the VCPU scheduler and the network switch to further reduce host-to-host latency in a data center setting. That work, however, does not consider queueing delays in the virtualization layer of dom0, i.e., the netif, netback, or vif devices, which can play a significant role.

RT-Xen [20], [21], [22] provides a real-time VCPU scheduling framework recently included in Xen 4.5. Xi et al. [23] develop RTCA, which implements a prioritization-aware packet scheduling in the netback device of dom0. RTCA is able to offer real-time guarantees to *local* inter-domain communications. VATC seeks to extend those guarantees to communications with *remote* hosts.

Another related topic is how to improve guest domains communication performance by allocating additional cores to each domain. Xu et al. [11] 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 [24] 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 the average network performance in virtualized hosts (Xen or KVM).

Finally, other work has focused on NICs supporting SR-IOV [25], a pass-through mechanism 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 [26], [27]. In contrast, VATC does not require special hardware support. Radhakrishnan et al. [28] present SENIC, which implements rate limiters and transmit schedulers in hardware. While SENIC is designed to improve the scalability and performance of the low-level network stack, VATC focuses on mitigating priority inversion in the virtualization layer above the native network stack. SENIC and VATC are therefore complementary to each other.

VI. 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 traffic control schemes. Our investigation reveals that mechanisms introduced to support virtualization can interfere with the ability of those traditional traffic control schemes to enforce prioritization and, therefore, delay guarantees. This is because of priority inversions in both transmissions and through interferences between transmission and reception in Xen's current network architecture. This motivates the development of VATC, a virtualization-aware traffic control architecture, which addresses these limitations. VATC dedicates netback devices and prioritized kernel threads to different priority levels so that real-time traffic can be protected. An implementation of VATC based on the kernel of dom0-3.18 is then evaluated over a testbed that supports different configurations of traffic patterns. The results of those experiments show that under VATC real-time traffic experiences lower and more predictable latency than under traditional Xen traffic control schemes.

VII. Acknowlegments

The authors would like to thank the reviewers and the shepherd for comments that greatly improved the paper. This research has been supported in part by ONR grant N000141310800 and by NSF grant CNS-1329861.

References

[1] United States Navy, "Total Ship Computing Environment(TSCE)," peoships.crane.navy.mil/ddx/tsce.htm.

[2] "Linux Advanced Routing and Traffic Control," http://www.lartc.org/.

[3] T. Høiland-Jørgensen, "Technical Description of FQ Codel," https://www.bufferbloat.net/projects/codel/wiki/Technical_description_of_FQ_CoDel.

[4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the Art of Virtualization," in *SOSP*, 2003.

[5] Nichols, Kathleen and Jacobson, Van, "Controlling Queue Delay," http://queue.acm.org/detail.cfm?id=2209336.

[6] Herbert, Tom, "bql: Byte Queue Limits," http://lwn.net/Articles/454378/.

[7] Intel, "Intel Network Driver Documentation," https://www.kernel.org/doc/Documentation/networking/e1000e.txt.

[8] www.bufferbloat.net, "Best Practices for Benchmarking CoDel and FQ CoDel," https://www.bufferbloat.net/projects/codel/wiki/Best_practices_for_benchmarking_Codel_and_FQ_Codel.

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

[10] B. Brandenburg and J. Anderson, "On the Implementation of Global Real-time Schedulers," in *RTSS*, 2009.

[11] C. Xu, S. Gamage, H. Lu, R. Kompella, and D. Xu, "vTurbo: Accelerating Virtual Machine I/O Processing Using Designated Turbo-Sliced Core," in *USENIX ATC*, 2013.

[12] Xen Wiki, "Xen Common Problems," http://wiki.xen.org/wiki/Xen_Common_Problems.

[13] Rick Jones, "Netperf Manual," http://www.netperf.org/.

[14] "VMware vSphere," http://www.vmware.com/products/vsphere.

[15] Vyenkatesh Deshpande, "vSphere 5 New Networking Features - Enhanced NIOC," http://blogs.vmware.com/vsphere/2011/08/vsphere-5-new-networking-features-enhanced-nioc.html.

[16] "Windows Server 2012 R2," http://www.microsoft.com/en-us/server-cloud/products/windows-server-2012-r2/.

[17] "Microsoft Cloud Platform," http://www.microsoft.com/en-us/server-cloud/solutions/virtualization.aspx.

[18] "Quality of Service (QoS) Overview," https://technet.microsoft.com/en-us/library/hh831679.aspx.

[19] "KVM: Kernel Based Virtual Machine," http://www.linux-kvm.org/page/Main_Page.

[20] S. Xi, J. Wilson, C. Lu, and C. Gill, "RT-Xen: Towards Real-time Hypervisor Scheduling in Xen," in *EMSOFT*, 2011.

[21] J. Lee, S. Xi, S. Chen, L. T. Phan, C. Gill, I. Lee, C. Lu, and O. Sokolsky, "Realizing Compositional Scheduling through Virtualization," in *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2012.

[22] S. Xi, M. Xu, C. Lu, L. T. X. Phan, C. Gill, O. Sokolsky, and I. Lee, "Real-Time Multi-Core Virtual Machine Scheduling in Xen," in *EMSOFT*, 2014.

[23] S. Xi, C. Li, C. Lu, and C. Gill, "Prioritizing Local Inter-Domain Communication in Xen," in *IWQoS*, 2013.

[24] N. Har'El, A. Gordon, A. Landau, M. Ben-Yehuda, A. Traeger, and R. Ladelsky, "Efficient and Scalable Paravirtual I/O System," in *USENIX ATC*, 2013.

[25] "PCI-SIG SR-IOV Primer: An Introduction to SR-IOV Technology," http://www.intel.com/content/www/us/en/pci-express/pci-sig-sr-iov-primer-sr-iov-technology-paper.html.

[26] "Deploying Extremely Latency-Sensitive Applications in VMware vSphere 5.5," http://www.vmware.com/files/pdf/techpaper/latency-sensitive-perf-vsphere55.pdf.

[27] Schnackenburg, Paul, "Hyper-V Deep Dive: Networking Enhancements," http://virtualizationreview.com/articles/2013/03/06/hyper-v-dive-3-network.aspx.

[28] S. Radhakrishnan, Y. Geng, V. Jeyakumar, A. Kabbani, G. Porter, and A. Vahdat, "SENIC: Scalable NIC for End-Host Rate Limiting," in *NSDI*, 2014.