vSnoop: Improving TCP Throughput in Virtualized Environments via Acknowledgement Offload

Ardalan Kangarlou, Sahan Gamage, Ramana Rao Kompella and Dongyan Xu
Department of Computer Science
Purdue University
{ardalan,sgamage,kompella,dxu}@cs.purdue.edu

***Abstract*—Virtual machine (VM) consolidation has become a common practice in clouds, Grids, and datacenters. While this practice leads to higher CPU utilization, we observe its negative impact on the TCP throughput of the consolidated VMs: As more VMs share the same core/CPU, the CPU scheduling latency for each VM increases significantly. Such increase leads to slower progress of TCP transmissions to the VMs. To address this problem, we propose an approach called vSnoop, where the driver domain of a host acknowledges TCP packets on behalf of the guest VMs – whenever it is safe to do so. Our evaluation of a Xen-based prototype indicates that vSnoop consistently achieves TCP throughput improvement for VMs (of orders of magnitude in some scenarios). We further show that the higher TCP throughput leads to improvement in application-level performance, via experiments with a two-tier online auction application and two suites of MPI benchmarks.**

I. Introduction

Virtual machine (VM) consolidation has been increasingly adopted in cloud (e.g., Amazon EC2[1], Eucalyptus [2], and Nimbus [3]), Grid, and datacenter environments. VM consolidation involves the hosting of multiple VMs on the same physical host. It allows dynamic multiplexing of computation and communication resources and leads to higher resource utilization and scalability of the physical infrastructure.

Scalable VM consolidation necessitates the sharing of the same CPU by multiple VMs. Even for a multi-core processor, the mapping from cores to VMs is *not* always one-to-one in order to achieve flexibility, scalability, and economy of VM hosting. However, we observe that VM consolidation negatively impacts TCP transport to VMs. More specifically, as more VMs are scheduled to access the same core/CPU, the CPU access latency for each VM (i.e. the interval during which a VM waits for the CPU) increases. Such increase raises the round-trip time (RTT) of a TCP connection to the VM, on top of the latency added by network device virtualization. As a result, the sub-millisecond propagation delay between hosts in a local area network (LAN) is overwhelmed by tens/hundreds of milliseconds of latency due to VM scheduling, which slows down the progress of the TCP transport considerably.

To mitigate the impact of VM consolidation identified above, we propose an approach called vSnoop that aims to improve the throughput of TCP connections to consolidated VMs. The key idea behind vSnoop is to allow the driver domain of a host (e.g., dom0 in Xen [4]) to acknowledge TCP packets on behalf of the less privileged production VMs (e.g., domUs in Xen) – whenever it is *safe* to do so. By *offloading acknowledgement* to the driver domain, vSnoop masks the portion of a TCP packet's RTT that corresponds to VM scheduling. The reduction in RTT prompts the sender to transmit to the VM at a higher rate, effectively saturating the link between the sender and the receiving VM. vSnoop requires no modification to the guest operating system or applications running in the VM. While we implement vSnoop on Xen, the methodology of vSnoop is generically applicable to other virtualization platforms (e.g., VMware, KVM, QEMU, VirtualBox) where the actual network drivers reside in a driver domain or inside the Virtual Machine Monitor (VMM).

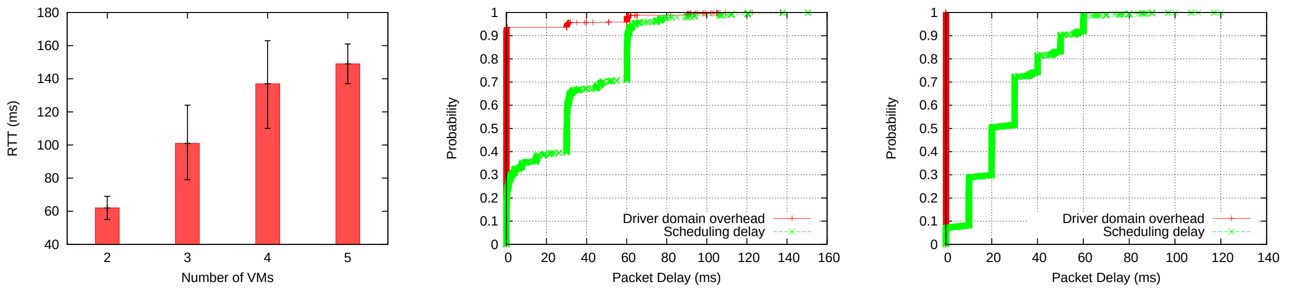
In our Xen-based prototype, vSnoop is implemented as part of the Linux bridge module [5] inside dom0. vSnoop does not lengthen the receive I/O path and only maintains a minimum state about each TCP connection. As a result, vSnoop is lightweight and incurs very low CPU overhead. We have performed extensive evaluation of vSnoop at both network transport and application levels. Our transport-level evaluation indicates that vSnoop constantly achieves higher TCP throughput than the original Xen – in some scenarios the improvement is of orders of magnitude. Our application-level evaluation shows that vSnoop consequently improves application performance, such as that of the RUBiS online auction benchmark and the High-Performance Linpack and Intel MPI benchmarks.

The main contributions of this paper are summarized as follows: (1) We identify and analyze the impact of CPU sharing on the TCP throughput of VMs (Section II). (2) We propose vSnoop as a light-weight, VM-transparent approach to mitigating such impact that can be instantiated on a range of virtualization platforms (Section III). (3) We develop a Xen-based prototype of vSnoop (Section IV) and demonstrate considerable improvements in TCP throughput and application-level performance for the VMs (Section V).

II. The Problem and Motivation

In this section we present a detailed description and investigation of the problem, namely the negative impact of VM consolidation/CPU sharing on TCP transport to VMs. On most existing virtualization platforms, the driver domain or the VMM hosts the actual device driver for a physical device. As such, the production VMs cannot directly interact with physical devices, including the network interface card (NIC).

SC10 November 2010, New Orleans, Louisiana, USA 978-1- 4244-7558-2/10/$26.00



(a) Sender-observed RTT

(b) Latency breakdown for incoming packets (RX path) in the host

(c) Latency breakdown for outgoing packets (TX path) in the host

Fig. 1. Effects of CPU sharing and network device virtualization on transport to consolidated VMs.

Regardless of whether the VM platform uses paravirtualized (e.g., as in paravirtualized Xen) or emulated (e.g., QEMU) devices, the extra hop in the network I/O path affects network performance due to the additional processing performed there such as interrupt handling, copying, and queuing. In this paper, we identify a more significant (yet less addressed) hurdle: As multiple VMs share the same core/processor, each VM may not get the CPU *in time* to process incoming TCP packets and advance the connection. To better understand how (and by how much) VMs' CPU sharing affects TCP throughput, we seek to answer the following questions:

(1) How does the CPU sharing by VMs affect the RTT of network packets?

(2) Is the RTT increase mostly due to VM scheduling or network device virtualization?

(3) Given the nature of RTT increase, how is TCP throughput affected?

Investigations. To answer the first question, we conduct a very simple experiment where a physical host sends *ping* packets to a non-idle (60% CPU load) Xen VM in the same LAN[1]. In this experiment, we vary the number of guest VMs (*i.e.*, domUs) that share the same core with the driver domain (*i.e.*, dom0) and observe the effect of VM CPU sharing on the RTT. From Figure 1(a), we observe that, as the number of non-idle VMs per core increases, the RTT of the *ping* packets increase almost in proportion to the 30ms VM scheduling slice in Xen. Similar findings [6] were recently reported for the "small" instances on Amazon's EC2 platform where two VMs share the same core.

The answer to the second question is quite insightful. We find that the main culprit of the RTT increase is VM scheduling, *not* network device virtualization. To "zoom in" on the dynamics of VM scheduling, we trace packets (1) within the driver domain and (2) between the driver domain and the VM – on both receive (RX) and transmit (TX) paths, in a scenario where three non-idle VMs are hosted on the same core as dom0. Figure 1(b) illustrates the cumulative density functions (CDFs) for (1) the amount of time for dom0 to process a packet and (2) the amount of time for the receiver VM to get scheduled and consume the packet on the RX path. The figure shows that, for 93% of the packets, driver domain processing adds at most 0.45ms to the RTT[2]. However, the majority of the RTT increase takes place *after* the driver domain processing. During this period, the packets stay in a shared buffer between the driver domain and the receiver VM, until the VM gets scheduled to consume the packets. The "jumps" in Figure 1(b) at 30ms intervals correlate to the 30ms VM scheduling slice used by Xen's credit scheduler [7].

Figure 1(c) shows the dynamics on the TX path. The major difference between Figures 1(b) and 1(c) is the *shorter time* the packets spend on the TX path from the VM to dom0. Particularly, the "jumps" at 10ms intervals suggest that the driver domain gets scheduled quite frequently. However, both figures indicate that the sub-millisecond driver domain overhead is completely dominated by the tens/hundreds of milliseconds of latency from VM/driver domain scheduling. This observation also suggests that such considerable RTT increase *cannot* be eliminated by new devices (*e.g.*, NetChannel2 [8]) that support direct VM access to hardware (which only alleviate the latency caused by network device virtualization).

Now that we have identified CPU sharing and VM scheduling as the major source of RTT increase, we need to understand how it affects TCP throughput. As seen in Figures 1(b) and 1(c), Xen's credit scheduler can add varying amount of latency to a packet's RTT. Such latency ranges from a negligible amount to a few tens/hundreds of milliseconds – depending on when the VM is scheduled to run as well as the precise timing of various events. In general, the credit scheduler schedules the driver domain more frequently than the guest VMs. To illustrate this point further and study its impact on TCP throughput, we compare packet traces of a 1MB file transferred to the driver domain with traces of the same file transferred to a guest VM. This experiment involves the same 3-VM setup as in the previous experiment. While the traces vary between experiment runs, we pick two traces where dom0 and the VM get scheduled in almost uniform intervals. Figure 2 shows these traces. The main observation from this figure is that scheduling preference towards the driver domain results in a much faster transfer to the driver domain than to

[1] Details of the experiment setup are described in section V.

[2] This latency is comparable to the 0.1ms RTT in the LAN.

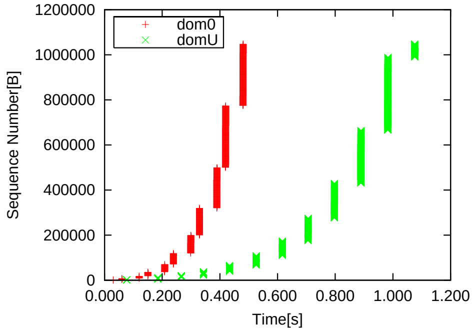


Fig. 2. Sequence/Time graph for a 1MB transfer to the driver domain and to a guest VM

the guest VM.

A more detailed explanation of the result above is as follows: As the driver domain gets scheduled more frequently, TCP slow start progresses a lot faster as packets are acknowledged at a higher rate than in the guest VM's case. Since the receive window at the receiver grows with every acknowledgement, the advertised window of the connection advances a lot more quickly too. Larger advertised receive window in turn prompts the sender to increase its congestion window and send more data in a shorter span of time. As a result, the connection to the driver domain progresses much more rapidly than the connection to the guest VM. We point out that VM scheduling heavily affects small flows (*i.e.*, the "mice" flows that typically spend their entire lifetime in TCP slow start). Since a vast majority of flows in a cloud/datacenter environment tend to be short transfers [9], [10], [11], such impact can be quite significant in those environments.

Implications. Findings from our investigations suggest the following idea: Since much of the RTT increase is due to VM scheduling on the RX path, if we somehow eliminate or mask this latency, we can greatly improve TCP throughput to the VMs. A natural way to hide the portion of RTT that corresponds to VM scheduling is to *offload* the TCP acknowledgment to the driver domain. This solution leverages the fact that the driver domain gets scheduled more frequently than the guest VMs and, as a result, the congestion window of the sender can be advanced a lot faster. The outcome of such an acknowledgement offload is a much faster progress of TCP connections – most notably for small flows; and a higher utilization of the high-speed network infrastructure (e.g., 10 Gigabit Ethernet, Infiniband) common in Grids, clouds, and datacenters.

However, offloading TCP acknowledgement to the driver domain must be performed judiciously, as one needs to preserve TCP's end-to-end semantics. Moreover, such offloading is applicable to scenarios where CPU is *not* the bottleneck for the consolidated VMs. If the CPU is the bottleneck, then obviously no improvement at the network I/O path can lead to more efficient execution of the guest VM. In the next section, we present the design of our solution, called vSnoop, that embodies the idea of acknowledgement offloading.

III. vSnoop Design

In this section we present the design of vSnoop. To show the applicability of vSnoop to a wide class of virtualization platforms (where either the driver domain or the VMM provides access to physical devices), we keep the description as platform-agnostic as possible and leave the platform-specific details to Section IV. Guided by our analysis in Section II, we place a new component called vSnoop inside the driver domain that performs early TCP acknowledgement on behalf of guest VMs. vSnoop is transparent to the VMs and does not require any modification to the guest operating system. As its name indicates, vSnoop *snoops* on all incoming and outgoing packets to/from the VMs and maintains the necessary state critical to *safe* early acknowledgement. More specifically, vSnoop maintains a minimal, per-flow state throughout the lifetime of a TCP connection to a VM and uses it to decide whether early acknowledgement for packets destined to a VM may lead to violation of end-to-end TCP semantics. In particular, vSnoop must avoid the scenario where the TCP sender receives an ACK for a packet without the packet ever reaching the receiver VM.

A. Overview

Figure 3 illustrates vSnoop's placement within the driver domain and its position relative to the guest VMs. vSnoop has two main criteria for safe early acknowledgement: (1) For a given TCP connection, vSnoop only acknowledges in-order packets. To keep vSnoop scalable, vSnoop does not buffer out-of-order packets which may arise as a result of packet losses or packets taking a different route. Instead, vSnoop simply passes all out-of-order packets to the receiver VM and let the VM handle them as it normally would in the absence of vSnoop. (2) vSnoop acknowledges in-order packets only when the shared buffer between the driver domain and the guest VM is not full. vSnoop takes this precaution so that all packets acknowledged by vSnoop are guaranteed to be delivered to the target VM and hence, TCP semantics are preserved at all times. In addition to acknowledging all in-order packets, vSnoop suppresses all (empty) ACKs coming from the VM if the ACKs correspond to packets already acknowledged by vSnoop. With one exception (to be discussed in Section III-C), vSnoop takes this measure to prevent unnecessary duplicate ACKs from reaching the sender.

vSnoop identifies TCP flows based on their source and destination IP addresses and port numbers and maintains a small hash table to store information about each flow. This mechanism is similar to how TCP/IP stack at end-host maintains per-flow TCP control information. For each flow, vSnoop maintains (1) the sequence number of the in-order packet expected to be received by vSnoop (NEXT_SEQ), (2) the sequence number expected to be received by the VM (VM_SEQ), (3) TCP receive window size (RCV_WIN), and (4) the current mode of operation for this flow (FL_MODE). Next we will show how this per-flow state is maintained and used to realize early acknowledgement.

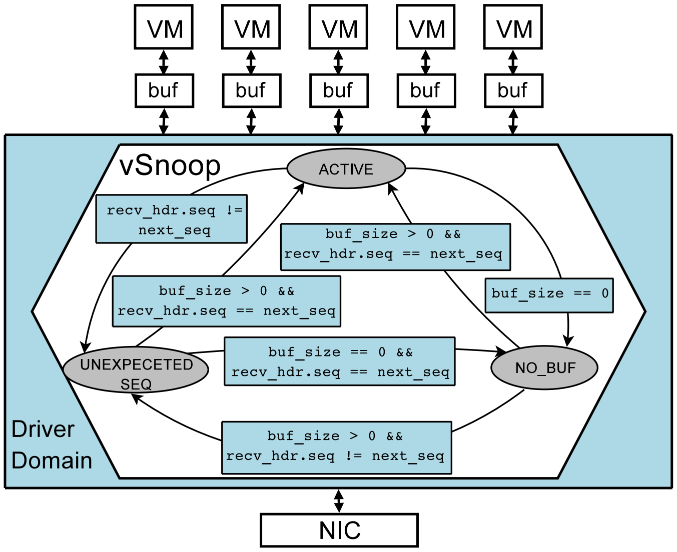

VM
VM
VM
VM
VM
buf
buf
buf
buf
buf
vSnoop
ACTIVE
recv_hdr.seq != next_seq
buf_size > 0 && recv_hdr.seq == next_seq
buf_size > 0 && recv_hdr.seq == next_seq
buf_size == 0
UNEXPECETED SEQ
buf_size == 0 && recv_hdr.seq == next_seq
NO_BUF
buf_size > 0 && recv_hdr.seq != next_seq
Driver Domain
NIC

Fig. 3. Overview of vSnoop and its per-connection state machine.

B. vSnoop State Machine

Figure 3 also depicts vSnoop's state machine that determines early acknowledgement "safety" on a per-flow basis. If FL_MODE is ACTIVE for a given flow, vSnoop will perform early acknowledgement for all in-order packets in the flow. In this state, vSnoop discards empty ACKs (i.e. ACKs with no data payload) coming from the VM to prevent delivery of duplicate ACKs to the sender. However, if FL_MODE is in either UNEXPECTED_SEQ or NO_BUF state (i.e. the packet is out-of-order or there is no space in the shared buffer), vSnoop will go offline for that flow and let the VM handle acknowledgement.

While vSnoop is offline, it uses the ACKs coming from the VM to update the per-flow VM_SEQ and NEXT_SEQ values. Meanwhile, the sender keeps sending packets until unacknowledged data reaches the minimum of the sender's congestion window or receiver's advertised window. At some point the sender would not send any new packets unless it receives an ACK from the VM. Hence, with high likelihood, the VM will receive all in-flight packets the next time it gets scheduled. Subsequently, it is very likely that the VM generates ACKs for all these packets and vSnoop receives them in one batch once the driver domain gets scheduled. After this point, if the sender sends a new packet, since the sequence number of this packet is going to be equal to NEXT_SEQ, FL_MODE becomes ACTIVE again and early acknowledgement resumes. This is predominantly the way vSnoop becomes online as vSnoop usually receives all the ACKs in one batch prior to the sender receiving them. The other less frequent scenario in which vSnoop gets back online is through TCP retransmission. More specifically, if VM scheduling intervals become too long such that they trigger timeout for unacknowledged packets, the sender starts retransmissions from the packet whose sequence number is NEXT_SEQ. This packet brings vSnoop online again and early acknowledgement resumes.

C. Technical Challenges and Solutions

There are two main challenges in the development of vSnoop: (1) To keep vSnoop online most of the time; (2) To make vSnoop behave just like a standard TCP implementation. To address the first challenge, vSnoop bounds the advertised receive window of ACKs generated by itself or by the receiver VM to the shared buffer size (buf_size). Bounding the number of outstanding packets in this fashion greatly reduces the likelihood of retransmission when vSnoop is offline, thus increasing the likelihood of vSnoop being online most of the time. Our measurements in Section V-C show that setting an upper bound for the advertised window does *not* make vSnoop any less efficient than the original Xen (*i.e.*, no vSnoop in the driver domain). With this simple design, vSnoop can perform early acknowledgement for the vast majority of packets, effectively keeping the shared buffer between the driver domain and the VM full most of the time. We note that, for "large" flows where the receive window of a connection has grown large enough to fill up the shared buffer, the benefit of vSnoop narrows relative to its benefit for "small" flows, as buffer exhaustion leads to vSnoop going offline more frequently. Nonetheless, our evaluation shows that vSnoop always outperforms the original Xen for all flow sizes.

To address the second challenge, we identify an important issue in keeping vSnoop's behavior consistent with TCP semantics. It concerns the receive window value advertised by vSnoop during early acknowledgement. As described in Section II, the main objective of vSnoop is to make TCP slow-start for connections to a VM behave more like TCP-slow start for connections to the driver domain. Therefore, similar to the TCP layer in the driver domain, vSnoop increments the receive window (RCV_WIN) by twice the maximum segment size (MSS) upon acknowledging each packet until the receive window reaches buf_size. The outcome is the exponential growth in the congestion window of the sender as defined by TCP standards. Also, upon receiving an ACK from a VM, vSnoop updates RCV_WIN with the value advertised in the ACK. Therefore, it just takes one VM scheduling slice for vSnoop to synchronize RCV_WIN with the value advertised by the VM. We noted in Section III-A there is one exception with respect to dropping the empty ACKs generated by a VM. vSnoop does not drop the ACK coming from a VM that acknowledges the last packet acknowledged by vSnoop. This behavior is consistent with RFC 793 [12] in order to notify the sender of the most recent receive window size.

There are two more subtle issues worth mentioning: (1) vSnoop cannot drop ACK packets coming from a VM that have data payload. vSnoop must pass these packets to their destination so that a connection can progress in both directions. Given that these packets may acknowledge packets that have already been acknowledged by vSnoop, some TCP implementations may discard these packets or cause complications. Therefore, for this type of packets, vSnoop rewrites the acknowledgement number to (NEXT_SEQ - 1) to ensure their delivery. (2) vSnoop's rewriting of the receive window and the

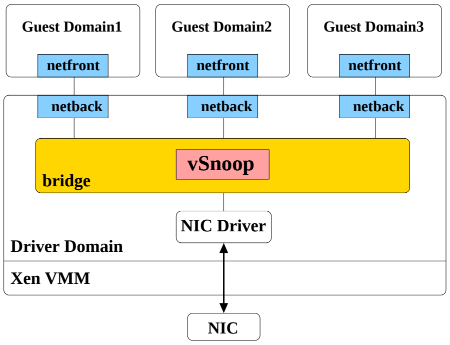


Fig. 4. Xen I/O architecture and vSnoop on Xen

acknowledgement number for packets from a VM invalidates their TCP checksum. Hence, after modifying a TCP header, vSnoop sets the TCP checksum field with the correct value. Alternatively, checksum calculation can be delegated to NICs with checksum offloading support [13].

Finally, to preserve end-to-end TCP semantics, vSnoop requires that no packet be lost between the driver domain and the TCP layer in the VM. Fortunately, the following factors collectively guarantee such a condition: (1) Packet transfer between the driver domain and the target VM is merely a memory copy operation which is deemed reliable. (2) Since vSnoop bounds the advertised receive window and acknowledges packets only if there is adequate space in the shared buffer, vSnoop greatly reduces the possibility of exhausting kernel resources in the guest VM. (3) Most importantly, one particular aspect of Xen I/O networking (to be discussed in Section IV) guarantees that intermediary buffers and resources in the guest VM are never exhausted. Considering all these factors, the presence of vSnoop does not require special tuning of the guest operating system or the network drivers.

IV. vSnoop Implementation

We have implemented vSnoop for the paravirtualized Xen platform as the paravirtualized devices are more efficient and portable than the emulated devices and they do not require any hardware support[3]. Before describing the implementation details, we briefly describe Xen's network device virtualization.

A. Background and Overview

Xen uses a *split driver* model for paravirtualized devices where each driver has a back-end component in the driver domain (i.e. dom0) and a front-end component in each guest domain (VM). These two components interact with each other via ring buffers, shared memory, and event channels. A ring buffer holds all I/O *request* and *response* operations corresponding to a specific device. Typically, each *request* corresponds to an I/O activity from the front-end to the back-end while a *response* corresponds to an I/O activity in the reverse direction. Both *request* and *response* operations reside on the ring buffer and both point to the actual data to exchange. The data, which can be a network packet or a disk block, is located on the shared memory pages between dom0 and a guest VM so that it is accessible by both back-end and front-end drivers. Finally, the event channel acts like an interdomain interrupt mechanism between dom0 and a guest VM.

Figure 4 presents an overview of network device virtualization in Xen and its *netback* and *netfront* components. This figure also shows that vSnoop is implemented as part of the bridge module inside dom0. To better understand vSnoop's functionality, we first examine the way Xen handles packet arrival had vSnoop not been deployed. Upon the arrival of a packet at the host's physical NIC, the driver domain receives the packet and determines the receiver VM using the bridge module. Once the receiver is determined, the bridge module hands the packet to the corresponding *netback* instance which in turn picks a *request* from the ring buffer and places a *response* in its place. Once all *responses* corresponding to incoming packets to a VM are placed, *netback* notifies *netfront* by sending an event.

When the receiver VM gets scheduled, the corresponding *netfront* starts consuming the *responses* placed by *netback* and starts placing new, empty *requests* in the ring for future incoming packets. What is particularly important is that before placing new *requests* in the ring buffer, *netfront* allocates memory for all packets that would be associated with these new *requests*. This guarantees that, once a packet reaches *netback*, no shortage of memory in a guest VM would lead to the packet's dismissal. As we briefly alluded to in Section III, this behavior particularly suits vSnoop, as all packets acknowledged by vSnoop are guaranteed enough resources in advance. Each network interface in a Xen VM has a separate RX and TX ring buffer and they all interact with *netback* in a similar fashion.

Over the course of our experiments, we realize that Xen uses a dynamic algorithm that places variable amount of new requests in the ring. However, we find out that this algorithm does not always perform as well as it was intended and places relatively few requests in the ring. Therefore, in order for vSnoop to perform early acknowledgment for a larger number of packets, we make one change to the *netfront* driver so that it can use a larger portion of the ring for *requests*. More specifically, we tune *netfront* so that it can use at least up to 75% of the 256 slots in the ring buffer for placing new *requests*. We will refer to our optimization as 'Xen+tuning' in Section V.

With the background above, vSnoop is implemented as two main hook functions attached to the bridge. Based on the direction of packets relative to a VM (incoming or outgoing), vSnoop engages either *vSnoop_egress* or *vSnoop_ingress* hook function. Both *vSnoop_egress* and *vSnoop_ingress* process packets by operating on socket buffer (*sk_buff*) kernel structures. With the placement of vSnoop in the bridge module, these functions receive *sk_buff* structures with L2 headers. Therefore, to identify TCP flows, both functions need to extract IP and TCP header fields from *sk_buff*. vSnoop identifies TCP flows based on their source and destination IP addresses

[3] We believe vSnoop's early acknowledgement methodology can also be applied to emulated devices.

and port numbers and maintains a small hash table to store information about each flow. For the remainder of this section we show how *vSnoop_egress* and *vSnoop_ingress* maintain the per-flow state (e.g., NEXT_SEQ, VM_SEQ, RCV_WIN, and FL_MODE) and implement the state machine defined in Section III.

B. Implementation Details

Handling outgoing packets. The primary function of *vSnoop_egress* is to intercept all packets from a VM and set up/maintain the per-flow information. The per-flow information is usually set up during TCP handshake when the VM sends SYN or SYN-ACK packet. The state can also be initialized after the TCP handshake if no per-flow information is present. The latter enables vSnoop's operation in the presence of live VM migration (discussed later in this section). Once the state is initialized, *vSnoop_egress* updates the per-flow VM_SEQ, RCV_WIN, and FL_MODE values based on the ACK packets it receives from the VM. The other functionality of *vSnoop_egress* is to drop unnecessary duplicate ACKs from the VM. This process involves examining whether a packet received from the VM is an empty ACK, if the packet has already been acknowledged by *vSnoop_ingress* and if the packet has no control flags set (such as SYN or FIN bit). If a packet satisfies all the above conditions, then it will be dropped.

As described in Section III, when the advertised receive window by the VM exceeds buf_size, *vSnoop_egress* sets the window to buf_size to limit the number of outstanding packets so that vSnoop can remain online or become online soon. Rewriting the acknowledgment number for ACKs with data and re-calculating a packet's TCP checksum are two other functions of *vSnoop_egress*. Finally, *vSnoop_egress* removes the information associated with a flow in the hash table once a connection is terminated by FIN or RST packets.

Handling incoming packets. The main function of *vSnoop_ingress* is to perform early acknowledgement as described in Section III. Upon receiving a TCP packet, *vSnoop_ingress* first determines the corresponding flow. If the sequence number of the packet matches NEXT_SEQ for that flow, then this packet becomes a candidate for early acknowledgement. *vSnoop_ingress* acknowledges a candidate packet when the following conditions are met: no control flag in the TCP header is set; receive window (RCV_WIN) is non-zero; and the ring buffer is not full. Every time *vSnoop_ingress* acknowledges a packet, it increases RCV_WIN by (2 × MSS) after confirming that there is enough buffer space in the ring buffer. While packets do not necessarily get dropped when the ring buffer is full, this check guarantees early acknowledged packets are never dropped en route to the receiver VM.

Handling live VM migration. vSnoop can also handle live VM migration. In Xen live migration [14], memory pages belonging to a VM are copied from the source host to the destination host in multiple iterations while the VM is running. The problem that may arise with vSnoop during migration is that a VM may complete migration before an early-acknowledged TCP packet reaches the receiver VM. Such a scenario would result in an inconsistent state where the sender receives an ACK from vSnoop for a packet that is not going to be delivered to the VM. To handle live VM migration, we adopt a straightforward yet effective solution where vSnoop gets disabled for all the flows involving the migrating VM prior to the first iteration of memory page copying. Since the duration of VM migration is magnitudes longer (typically a few seconds or more), the VM will receive all the early-acknowledged packets during this period. Once the VM moves to the destination, vSnoop at the destination host, if deployed, will initialize the hash table entries for the VM's active flows and early acknowledgement resumes.

V. Evaluation

In this section we present evaluation results on vSnoop: Section V-B evaluates the overhead of vSnoop itself; Section V-C focuses on the TCP performance achieved by vSnoop; and Section V-D demonstrates the effect of vSnoop on application-level performance.

A. Testbed Setup

The experiments are performed in our virtualized cloud computing testbed connected by Gigabit Ethernet. Each VM-hosting server runs Xen 3.3 with Linux 2.6.18 as the operating system for both the driver domain and the paravirtualized guest VMs. (1) The experiments in Sections II, V-B, and V-C involve a client machine and a server. The client machine has a 2.4GHz Intel Core 2 Quad CPU with 2GB of RAM and an Intel Pro Gigabit network card and runs Linux 2.6.19. The server hosts the VMs and has a dual-core 3GHz Intel Xeon CPU with 3GB of RAM and a Broadcom NetXtreme 5752 Gigabit Ethernet card. The VMs each have 256MB of RAM. (2) The experiments in Section V-D involve multiple server hosts, each being a PowerEdge Dell server with a 3.06GHz Intel Xeon CPU, 4GB of RAM, and a Broadcom NetXtreme 5704 Gigabit Ethernet card. TCP Reno is used in all experiments.

B. Profiling vSnoop Overhead

In the design and the implementation of vSnoop, we strive to keep vSnoop as light-weight as possible by only including the minimal functionality of the TCP layer at vSnoop that is essential to TCP acknowledgement offload. To better understand the overhead associated with vSnoop, we use the Xenoprof [15] toolkit for system profiling. Xenoprof supports profiling at the fine granularity of individual processes and routines executed in the Xen VMM, driver domain, and guest VMs. We use Xenoprof to measure the overhead associated with different vSnoop routines in terms of the CPU cycles/percentage they consume. We additionally instrument vSnoop routines to record the number of packets they process. This information helps us to obtain the per-packet cost or the cost incurred by vSnoop routines at a given point in time.

Table I presents the average vSnoop overhead for 10-second Iperf [16] transfers for two scenarios: (1) In the "single stream" scenario, there is one connection from the client to a VM.

vSnoop Routines	Single Stream		Multiple Streams	
	Cycles	CPU %	Cycles	CPU %
vSnoop_ingress()	509	3.03	516	3.05
vSnoop_lookup_hash()	74	0.44	91	0.51
vSnoop_build_ack()	52	0.32	52	0.32
vSnoop_egress()	104	0.61	104	0.61

TABLE I
PER-PACKET CPU UTILIZATION FOR VSNOOP ROUTINES

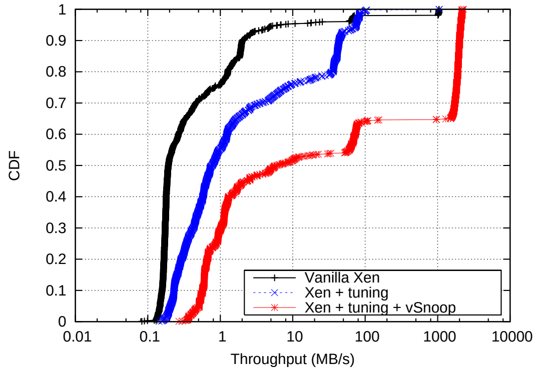


Fig. 5. CDFs for TCP throughput of 1000 successive 100KB transfers for vanilla Xen, Xen with our tuning, and Xen with our tuning and vSnoop

Our measurements show vSnoop adds about 4.5% to the CPU utilization of the driver domain. Much of this cost is associated with the *vSnoop_ingress* routine. The routine that looks up a flow's state in the hash table and the routine that builds the ACK each incur negligible overhead. (2) In the "multiple stream" scenario, we have 100 concurrent connections to the 5 VMs running in the server. The per-packet cost or the vSnoop overhead at a given point of time remains largely unchanged from the single stream scenario. The only routine that incurs slightly higher cost for the multiple stream scenario is *vSnoop_lookup_hash()*. This is intuitive as vSnoop has to search a larger hash table to retrieve information about a particular flow.

C. TCP Throughput Evaluation

In this section, we test vSnoop for transport-level performance. To gain full control over the experiment setup, we develop our own TCP application, *TCP-app*, that works similarly to Iperf. TCP-app involves sending data of various sizes from the client to the VM. We set up a variety of scenarios to assess the TCP throughput achieved by vSnoop. For each scenario, we compare TCP throughput under (1) the vanilla Xen, (2) Xen with our *netfront* tuning (Section IV-A), and (3) Xen with our tuning and vSnoop. With the exception of one scenario, we enable only one core in the server host so that the impact of VMs' CPU sharing/scheduling can be studied without interference.

To better understand the nature of the experiments we present Figure 5. This figure shows the cumulative distribution functions (CDFs) for 1000 successive 100KB transfers from the client to the VM for vanilla Xen, Xen with *netfront* tuning, and Xen with *netfront* tuning and vSnoop. In this experiment, the server VM is co-located with two other non-idle[4] guest VMs. This figure shows that vSnoop (with tuning) yields significant and in some cases orders of magnitude of improvement in TCP throughput. In particular, the median throughput values for 'vanilla Xen', 'Xen+tuning', and 'Xen+tuning+vSnoop' are 0.192 MB/s, 0.778 MB/s, and 6.003 MB/s, respectively.

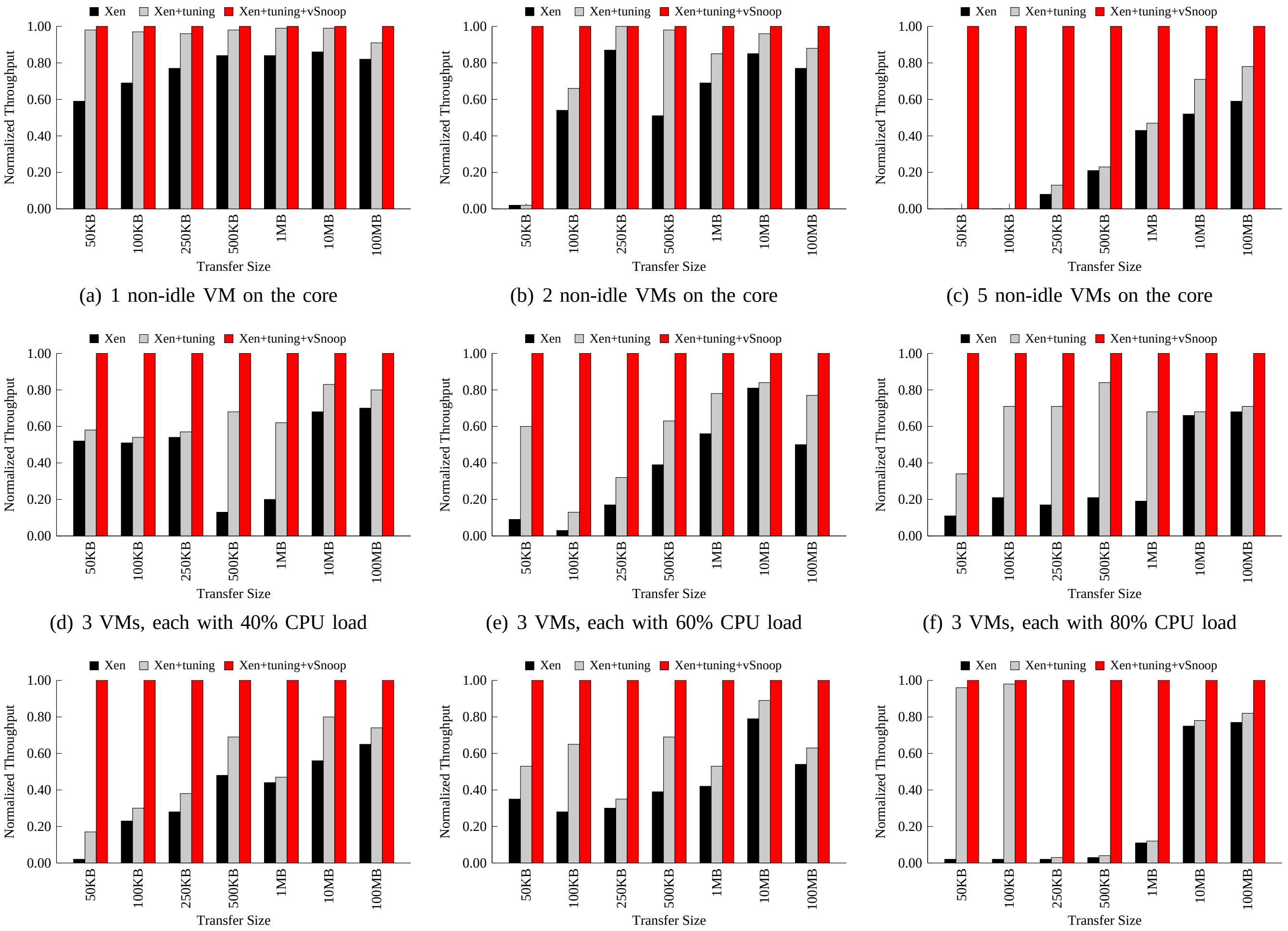
It is interesting to point out that for about a third of measurements in 'Xen+tuning+vSnoop', the TCP throughput exceeds the link rate between the client and server hosts. The reason is: vSnoop's presence leads to a large number of packets getting buffered in the driver domain. Since memory copying between the driver domain and the receiver VM is much faster than the link rate, the TCP throughput observed by the VM appears as if it had exceeded the network link rate. Similar phenomenon was reported for UDP transport between Amazon EC2 instances [6]. Another observation we make is that simply comparing the average throughput values for the three configurations is not the best way to evaluate vSnoop. However, due to space constraint we cannot present CDFs for all the experiments. For the rest of this section we only compare the median throughput values for the three configurations. While this type of comparison in many cases under-represents the benefits of vSnoop, we overall find it a suitable way of assessing vSnoop's performance.

Figure 6 presents the results for different transfer sizes under a variety of scenarios. All median throughput values (of 1000 runs) for a specific transfer size are normalized based on the value for the 'Xen+tuning+vSnoop' configuration.

1. Varying the number of VMs. Figures 6(a), 6(b), 6(e), and 6(c) show the effectiveness of vSnoop when 1, 2, 3 and 5 non-idle VMs (including the receiver VM itself) are running on the same core respectively. These figures show that vSnoop constantly outperforms vanilla Xen. More importantly, the benefit of vSnoop increases with higher degree of VM consolidation and with smaller transfers. Higher VM consolidation worsens the impact of VM scheduling, thus widening the gap between vanilla Xen and vSnoop; Short transfers are particularly susceptible to VM scheduling (Section II) and hence benefit more from vSnoop. It is also worth noting that capping the advertised receive window by vSnoop (Section III) does not hurt TCP throughput. vSnoop outperforms other configurations even for the large transfers in the 1-VM scenario, where the advertised receive window can get very large in the absence of vSnoop.

2. Varying CPU load. To understand the effect of VM CPU load, we fix the number of VMs on a core to three and vary the VM CPU load. Figures 6(d), 6(e), and 6(f) show vSnoop outperforms other configurations under different CPU load in the VMs. Higher VM workload makes CPU scheduling more detrimental to TCP throughput, thereby making vSnoop more

[4] All references to non-idle VMs entail 60% CPU load on the VM, unless otherwise specified.



(a) 1 non-idle VM on the core

(b) 2 non-idle VMs on the core

(c) 5 non-idle VMs on the core

(d) 3 VMs, each with 40% CPU load

(e) 3 VMs, each with 60% CPU load

(f) 3 VMs, each with 80% CPU load

(g) 10 concurrent connections to a VM

(h) Sender subject to VM scheduling

(i) driver domain run on a separate core

Fig. 6. TCP throughput measurements under a variety of scenarios.

useful.

3. Concurrent connections. Thus far, all the results are based on a single connection from the client to the VM. Figure 6(g) shows that vSnoop is also effective when there are 10 concurrent connections to the VM. The results presented are for a setup where 3 non-idle VMs run on the same core.

4. Sender subject to VM scheduling. In this scenario, we investigate the effectiveness of vSnoop when the sender is also virtualized and subject to VM scheduling like the receiver. This is a quite common scenario inside a cloud or datacenter where hosted VMs communicate with each other. In fact, our application-level experiments (Section V-D) reflect such a scenario. Figure 6(h) presents the results for a setup where the client VM and the server VM are each co-located with two other non-idle VMs in their respective hosts.

5. Driver domain on a separate core. While the previous results show solid improvement by vSnoop, we wonder whether vSnoop would be even more effective when the driver domain does not have to compete with the guest VMs for CPU. Figure 6(i) presents the results from a scenario where the driver domain runs on a separate core from the one that supports 3 non-idle guest VMs. The results indicate that, for short transfers (up to 1MB), vSnoop outperforms vanilla Xen by a significantly large margin. This is because the driver domain and the receiver VM now process incoming packets on two separate cores. As a result, whenever the VM is scheduled, it is likely to process *all* the packets that the driver domain has passed to the VM or will pass while the VM is running, thus increasing the TCP throughput. However, the benefit margin narrows for large transfers for the same reason mentioned in Scenario 1.

D. Application-Level Evaluation

Experiment with RUBiS. To demonstrate the effectiveness of vSnoop for real-world applications running in a cloud or datacenter, we run the Rice University Bidding System (RUBiS) [17] in our testbed. RUBiS is a benchmark that evaluates application server performance for an auction site that resembles eBay [18]. RUBiS implements the core functionality of on-line auction such as browsing and searching for items, bidding, and selling. We use the PHP version of RUBiS which has two tiers: an Apache webserver and a MySQL database

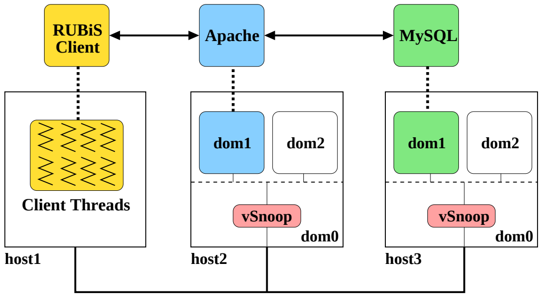


Fig. 7. RUBiS experiment setup

RUBiS Operation	Count w/o vSnoop	Count w/ vSnoop	% Gain
Home	359	396	10.3%
Browse	421	505	19.9%
BrowseCategories	288	357	23.9%
SearchItemsInCategory	3498	4747	35.7%
BrowseRegions	128	141	10.1%
BrowseCategoryInRegion	124	136	9.6%
SearchItemsInRegion	690	749	8.5%
ViewItem	2892	3776	30.5%
ViewUserInfo	732	846	15.6%
ViewBidHistory	339	398	17.4%
BrowserBackOperation	2750	3511	27.7%
EndOfSession	16	23	43.7%
Total	12237	15585	27.4%
Average Throughput	29 req/s	37 req/s	27.5%

TABLE II
RUBiS BENCHMARK RESULTS WITH "BROWSING MIX"

server. Figure 7 shows our setup: The VMs hosting Apache and MySQL (dom1s in the figure) are *each* co-located with a VM (dom2s) with 30% CPU load. Each of the VMs has 768MB of RAM. Since vSnoop is deployed in all server hosts in the testbed, it will benefit the TCP connections between the client and Apache and between Apache and MySQL.

We run the RUBiS benchmark for a 7-minute period (1) without vSnoop (*i.e.*, 'Xen+tuning') and (2) with vSnoop (*i.e.*, 'Xen+tuning+vSnoop'). We turn on our 'tuning' enhancement in both scenarios. In this experiment, 180 client threads perform operations such as browsing web pages, viewing items, searching for items in a geographical region, etc. We use the "browsing mix" workload where clients trigger read requests to the Web and database servers. The goal of this experiment is to assess how the TCP-level improvement translates into application-specific performance improvement – in RUBiS's case, the number of user requests handled per second. To stress test both 'with vSnoop' and 'without vSnoop' setups, we make one slight change to the RUBiS client implementation such that, right after an operation is done, the client thread starts the next operation (*i.e.*, no sleep time between operations). Table II shows the counts of various operations performed as well as the overall system throughput. With vSnoop, RUBiS performs higher number of each type of operations, which translates into a higher number of user requests (15585 vs. 12237) and throughput (37 req/s vs. 29 req/s), a 27% improvement.

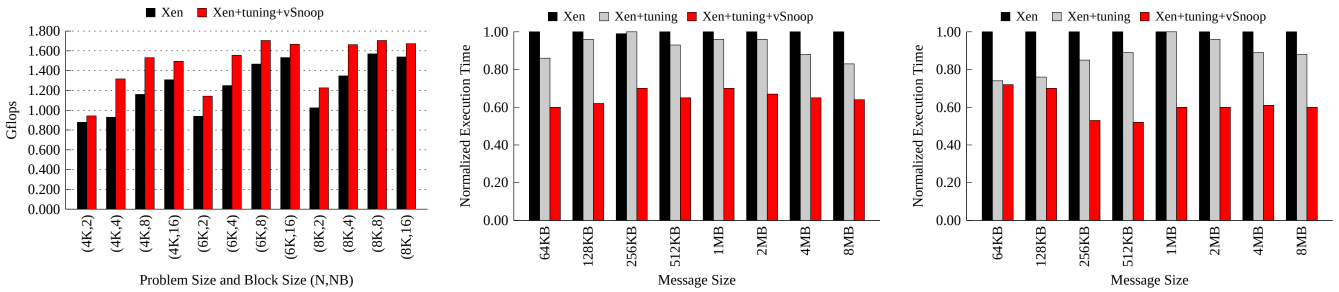
Experiments with MPI benchmarks. In these experiments, we assess the benefit of vSnoop for executing MPI programs in VMs. Our experiments use (1) the High-Performance Linpack (HPL) benchmark [19] and (2) the Intel MPI benchmark (IMB) [20]. HPL is *computation*-intensive and is primarily used to find the maximum floating-point operations (flops) per second achieved by a cluster; whereas the IMB is more *communication*-intensive and evaluates the efficiency of various communication patterns in a cluster.

In the HPL experiment, we set up a 4-VM MPICH2 [21] execution environment, with each VM hosted by a distinct physical server. Each VM has 256MB of RAM and is co-located with another non-idle VM with 30% load. Figure 8(a) presents the results under various problem sizes ($N \in \{4000, 6000, 8000\}$) and block sizes ($NB \in \{2, 4, 8, 16\}$). The results show that vSnoop improves the HPL performance (Gflops) in all runs compared with that achieved by the vanilla Xen. We do notice that the percentage of improvement is less than those seen in the TCP benchmark and RUBiS experiments. In fact, the lower performance gain is expected as HPL is more CPU-bound than I/O-bound. As such, the communication time saved by vSnoop usually gets dominated by the much longer computation time that precedes or follows the communications. Moreover, synchronization and inter-dependencies among nodes for an MPI execution to proceed is another factor that offsets some of the transport efficiency brought by vSnoop. However, even in such an unfavorable scenario, vSnoop constantly yields benefit of varying degree. Finally, we note that our results from the 'Xen+tuning' configuration are almost identical to those from the vanilla Xen so we do not present them in the figure. The reason is that the number of messages in our HPL runs are too small to benefit from our *netfront* tuning enhancement.

Our experiment with the IMB shows the effectiveness of vSnoop in reducing the execution duration of many MPI communication primitives. We use almost the same setup as the one for the HPL experiment. The only change we make is that, using a synthetic load generator, we increase the CPU load on the VMs to 60% as the IMB does not incur sufficient computation to study the CPU scheduling impact. Figures 8(b) and 8(c) show the results (normalized execution time) under "one-to-many" (*Broadcast*) and "many-to-many" (*Alltoall*) communication patterns, with varying message size. The results show that vSnoop leads to notably shorter execution time for IMB's *Broadcast* and *Alltoall* benchmarks. Results from IMB's other communication patterns also show the benefit of vSnoop and are omitted for lack of space.

VI. Related Work

In recent years, researchers have proposed various solutions to alleviate the overhead of network device virtualization for virtualized environments. These efforts can roughly be classified into three main categories: (1) optimizing the virtualized I/O path, (2) improving communication among VMs on the same host, and (3) making VM scheduling algorithms aware of VM communication.



(a) HPL benchmark

(b) Intel MPI benchmark - Broadcast

(c) Intel MPI benchmark - Alltoall

Fig. 8. MPI benchmark results

Menon et al. have proposed several improvements to the network device virtualization in [22], [23], and [24]. [22] shows that much of the virtualization overhead is due to per-packet operations between the guest VM and the driver domain and proposes packet aggregation (i.e. coalescing multiple TCP packets of the same connection into one big packet) to reduce per-packet overhead. In [23], the authors propose using scatter/gather I/O, TCP/IP checksum offload, and TCP segmentation offload for improving network performance of Xen VMs. TwinDrivers [24] is a framework that moves some of the device driver functionality from a guest VM to the VMM for better performance. By addressing a new problem (*i.e.*, TCP throughput degradation due to VM consolidation) *not* identified by the above efforts, vSnoop complements these techniques and can be integrated with them.

Many research efforts have tried to improve communication throughput between VMs on the *same* physical host. XenSocket [25], XenLoop [26], Fido [27], and XWAY [28] use shared memory primitives provided by Xen to bypass the driver domain and create efficient communication channels between VMs on the same host. While XenSocket introduces a new type of socket to the application-layer, XWay, XenLoop and Fido are transparent to applications as the inter-domain communication channel is placed underneath the network stack. IVC [29] is another effort in this direction that targets the high performance computing (HPC) domain. More specifically, the authors design a VM-aware MPI library which enables HPC applications to transparently benefit from efficient inter-VM communication channels between co-located VMs. vSnoop complements these approaches as it is transparent to the applications and communication libraries running inside the VMs. Moreover, it is applicable to communications between VMs on *different* hosts.

Extensions to Xen's SEDF scheduler are proposed in [30], which makes the VM scheduling in Xen communication-aware. More specifically, the authors propose preferential scheduling of the recipient VM and anticipatory scheduling of the sender VM to improve the performance of network-intensive workloads. vSnoop is designed as a driver domain-level technique that is *agnostic* to a specific VM scheduling algorithm in the VMM. As a result, vSnoop can be deployed on a virtualization platform with such an enhanced VM scheduler. Further, we believe a communication-aware VM scheduler, like the one in [30], creates favorable conditions for vSnoop as it can lead to faster consumption of packets on the shared buffer which keeps vSnoop online most of the time.

The idea of snooping on packets to improve TCP throughput in lossy and high bit-error rate wireless networks was proposed in [31]. In that work, the wireless access point caches packets and performs local retransmissions to wireless nodes whenever needed. Despite the conceptual similarity between their approach and vSnoop, the network and end-host characteristics faced by the two are very different. Hence their design and implementation largely differs from that of vSnoop. For example, in vSnoop we offload TCP *acknowledgement* to the driver domain while in [31] TCP *retransmission* is offloaded to the base station.

VII. Conclusion

We have presented vSnoop as a technique that mitigates the impact of CPU sharing on the throughput of TCP connections to consolidated VMs. vSnoop is based on the observation that CPU scheduling among VMs adds a significant, last-hop latency to the RTT of TCP packets, resulting in TCP throughout degradation. Hence, the idea behind vSnoop is to offload TCP acknowledgment to the driver domain – whenever it is safe – to hide most of the VM scheduling-related latency from the sender. Evaluations of our Xen-based prototype, at both network transport and application levels, demonstrate the efficiency and effectiveness of vSnoop for virtualized cloud, Grid, and datacenter environments. As part of our future work, we will explore the development of vSnoop in the NIC hardware and study the interplay between vSnoop and communication-aware VM schedulers.

Acknowledgments

We thank the anonymous reviewers for their insightful comments. This work was supported in part by the US NSF under grants 0546173, 0720665, 0721680 and 0831647. Any opinions, findings, and conclusions or recommendations in this paper are those of the authors and do not necessarily reflect the views of the NSF.

References

[1] "Amazon Elastic Compute Cloud (Amazon EC2)," http://aws.amazon.com/ec2/.

[2] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov, "The Eucalyptus open-source cloud-computing system," in *Proceedings of the 9th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID '09)*, Washington, DC, 2009, pp. 124–131.

[3] "Nimbus Toolkit," http://www.nimbusproject.org/.

[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 *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing, NY, 2003.

[5] "Linux net:bridge," http://www.linux-foundation.org/en/Net:Bridge.

[6] G. Wang and T. S. E. Ng, "The impact of virtualization on network performance of Amazon EC2 data center," in *Proceedings of IEEE INFOCOM*, San Diego, CA, 2010.

[7] "Xen credit scheduler," http://wiki.xensource.com/xenwiki/CreditScheduler.

[8] J. R. Santos, G. Janakiraman, Y. Turner, and I. Pratt, "Netchannel 2: Optimizing network performance," in *Xen Summit*, 2007.

[9] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, "The nature of data center traffic: measurements & analysis," in *Proceedings of the 9th ACM Internet Measurement Conference (IMC '09)*, Chicago, IL, 2009, pp. 202–208.

[10] T. Benson, A. Anand, A. Akella, and M. Zhang, "Understanding data center traffic characteristics," in *Proceedings of the 1st ACM Workshop on Research on Enterprise Networking (WREN '09)*, Barcelona, Spain, 2009, pp. 65–72.

[11] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "VL2: a scalable and flexible data center network," in *Proceedings of the ACM SIGCOMM conference on Data communication (SIGCOMM '09)*, Barcelona, Spain, 2009, pp. 51–62.

[12] J. Postel, "Transmission control protocol," RFC 793, 1981.

[13] D. Minturn, G. Regnier, J. Krueger, R. Iyer, and S. Makineni, "Addressing TCP/IP processing challenges using the IA and IXP processors," *Intel Technology Journal*, vol. 7, 2003.

[14] C. Clark, K. Fraser, S. Hand, and J. G. Hansen, "Live migration of virtual machines," in *Proceedings of the 2nd ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI '05)*, San Diego, CA, 2005.

[15] A. Menon, J. R. Santos, Y. Turner, G. J. Janakiraman, and W. Zwaenepoel, "Diagnosing performance overheads in the Xen virtual machine environment," in *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments (VEE '05)*, Chicago, IL, 2005.

[16] "The Iperf Benchmark," http://www.noc.ucf.edu/Tools/Iperf/.

[17] "The RUBiS Benchmark," http://rubis.ow2.org.

[18] "eBay," http://www.ebay.com/.

[19] "The High-Performance Linpack Benchmark," http://www.netlib.org/benchmark/hpl/.

[20] "Intel MPI benchmark," http://software.intel.com/en-us/articles/intel-mpi-benchmarks/.

[21] "MPICH2," http://www.mcs.anl.gov/research/projects/mpich2/.

[22] A. Menon and W. Zwaenepoel, "Optimizing TCP receive performance," in *USENIX Annual Technical Conference*, Boston, MA, 2008, pp. 85–98.

[23] A. Menon, A. L. Cox, and W. Zwaenepoel, "Optimizing network virtualization in Xen," in *USENIX Annual Technical Conference*, Boston, MA, 2006, pp. 15–28.

[24] A. Menon, S. Schubert, and W. Zwaenepoel, "TwinDrivers: semi-automatic derivation of fast and safe hypervisor network drivers from guest OS drivers," in *14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '09)*, Washington, DC, 2009, pp. 301–312.

[25] X. Zhang, S. McIntosh, P. Rohatgi, and J. L. Griffin, "XenSocket: A high-throughput interdomain transport for virtual machines," in *ACM/IFIP/USENIX 8th International Middleware Conference (Middleware '07)*, Newport Beach, CA, 2007, pp. 184–203.

[26] J. Wang, K.-L. Wright, and K. Gopalan, "XenLoop: a transparent high performance inter-vm network loopback," in *17th International ACM Symposium on High Performance Parallel and Distributed Computing (HPDC '08)*, Boston, MA, 2008, pp. 109–118.

[27] A. Burtsev, K. Srinivasan, P. Radhakrishnan, L. N. Bairavasundaram, K. Voruganti, and G. R. Goodson, "Fido: Fast inter-virtual-machine communication for enterprise appliances," in *USENIX Annual Technical Conference*, San Diego, CA, 2009.

[28] K. Kim, C. Kim, S.-I. Jung, H.-S. Shin, and J.-S. Kim, "Inter-domain socket communications supporting high performance and full binary compatibility on Xen," in *Proceedings of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '08)*, Seattle, WA, 2008, pp. 11–20.

[29] W. Huang, M. J. Koop, Q. Gao, and D. K. Panda, "Virtual machine aware communication libraries for high performance computing," in *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing (SC '07)*, Reno, NV, 2007, pp. 1–12.

[30] S. Govindan, A. R. Nath, A. Das, B. Urgaonkar, and A. Sivasubramaniam, "Xen and co.: communication-aware CPU scheduling for consolidated Xen-based hosting platforms," in *Proceedings of the 3rd International Conference on Virtual Execution Environments (VEE '07)*, San Diego, CA, 2007, pp. 126–136.

[31] H. Balakrishnan, S. Seshan, E. Amir, and R. H. Katz, "Improving TCP/IP performance over wireless networks," in *Proceedings of the 1st Annual International Conference on Mobile Computing and Networking (MobiCom '95)*, Berkeley, CA, 1995.