Opportunistic Flooding to Improve TCP Transmit Performance in Virtualized Clouds

Sahan Gamage, Ardalan Kangarlou, Ramana Rao Kompella, Dongyan Xu

Department of Computer Science
Purdue University
West Lafayette, Indiana 47907, USA
{sgamage,ardalan,kompella,dxu}@cs.purdue.edu

ABSTRACT

Virtualization is a key technology that powers cloud computing platforms such as Amazon EC2. Virtual machine (VM) consolidation, where multiple VMs share a physical host, has seen rapid adoption in practice with increasingly large number of VMs per machine and per CPU core. Our investigations, however, suggest that the increasing degree of VM consolidation has serious negative effects on the VMs' TCP transport performance. As multiple VMs share a given CPU, the scheduling latencies, which can be in the order of tens of milliseconds, substantially increase the typically sub-millisecond round-trip times (RTTs) for TCP connections in a datacenter, causing significant degradation in throughput. In this paper, we propose a light-weight solution called vFlood that (a) allows a TCP sender VM to opportunistically flood the driver domain in the same host, and (b) offloads the VM's TCP congestion control function to the driver domain in order to mask the effects of VM consolidation. Our evaluation of a vFlood prototype on Xen suggests that vFlood substantially improves TCP transmit throughput with minimal per-packet CPU overhead. Further, our application-level evaluation using Apache Olio, a web 2.0 cloud application, indicates a 33% improvement in the number of operations per second.

Categories and Subject Descriptors

D.4.4 [**Operating Systems**]: Communications Management—*Network communication*

General Terms

Design, Performance, Measurement

Keywords

Virtualization, Cloud Computing, TCP, Datacenters

SOCC'11, October 27–28, 2011, Cascais, Portugal.
Copyright 2011 ACM 978-1-4503-0976-9/11/10 ...$10.00.

1. INTRODUCTION

Recent advances in cloud computing [12] and datacenter technologies have significantly changed the computing landscape. Enterprises and individual users are increasingly migrating their applications to public or private cloud infrastructures (e.g., Amazon EC2, GoGrid, and Eucalyptus [38]) due to their inherent economic and management benefits. The key technology that powers cloud computing is *virtualization*. By breaking away from the traditional model of hosting applications in physical machines, virtualization enables the "slicing" of each physical machine in a cloud infrastructure into multiple virtual machines (VMs), each individually hosting a server, service instance, or application component. This practice, commonly known as *VM consolidation* (or server consolidation), allows dynamic multiplexing of physical resources and results in higher resource utilization and scalability of the cloud infrastructure.

The practice of VM consolidation exploits the availability of modern commodity multi-core and multi-processor systems that facilitate easy allocation of resources (e.g., memory and CPU) across multiple VMs. Recent trends indicate a *rapid increase in VM density*—in a recent survey, about 25% of the enterprises in North America indicate that they already deploy 11-25 VMs per server, and another 12% indicate as high as 25 VMs per server [8]. We believe that technological advances such as techniques for the sharing of CPU [22, 45] and memory [23, 36] among VMs will only catalyze this trend further, leading to even higher degrees of VM consolidation in the future.

Meanwhile, many cloud applications tend to be *communication intensive*. The reason lies in the wide adoption of distributed computing techniques such as MapReduce [18] for large-scale data processing and analysis. In addition, many scalable online services (e.g., e-commerce, Web 2.0) hosted in the cloud are often organized as multi-tier services with server load balancers redirecting clients to web frontend servers, which in turn interact with backend servers (e.g., database, authentication servers). These applications involve communication across multiple VMs in the datacenter, and thus the application-level performance is directly dependent on the network performance between the VMs.

Unfortunately, our recent investigations [28] suggest that the two trends above are directly at odds with each other. Specifically, we observe that *sharing of CPU by multiple VMs negatively impacts the VMs' TCP transport performance*, which in turn affects the overall performance of many cloud applications. In particular, with multiple VMs sharing the same CPU, the latency experienced by each VM to obtain its CPU time slices increases. Furthermore, such CPU access latency (tens/hundreds of milliseconds) can be *orders of magnitude higher* than the typical (sub-millisecond) round-trip time (RTT) between physical machines within a datacenter. Consequently, the CPU access latency dominates the RTT between two VMs, significantly slowing down the progress of TCP connections between them.

Due to the closed loop nature of TCP, VM CPU sharing can negatively impact both the *transmit* and *receive* paths of a TCP connection. On the receive path, a data packet may arrive at the physical host from a remote sender in less than a millisecond; but the packet needs to wait for the receiving VM to be scheduled to process it and generate an acknowledgment (ACK). In our prior work [28], we have proposed a solution called vSnoop, in which we place a small module within the driver domain that essentially generates an ACK for an in-order data packet on behalf of the receiving VM under safe conditions, thus causing the TCP sender to ramp up faster than otherwise. We demonstrated that this approach can effectively improve the performance of network-oriented applications.

In this paper, we focus on the *transmit path* that has not been addressed in our prior work. On the sender side, because of CPU scheduling latency, the sender VM can get delayed in processing the TCP ACKs coming from the remote receiver, which causes its congestion window (and hence the sending rate) to grow slowly over time. At first glance, it may appear that this problem is quite similar to that on the receive path, and hence, we could devise a solution similar to vSnoop. However, notice that for the receive path, vSnoop could easily fabricate the ACKs in the driver domain since they are generated in response to data packets that already contain all the information necessary (mainly, sequence number) for generating the ACKs. Unfortunately, the same cannot be done on the transmit path since the driver domain cannot fabricate new data packets on behalf of the VM and only the VM can undertake this task. Thus, a straightforward extension of vSnoop will not work on the transmit path.

Of course, the principle of getting help from the driver domain is still logical even on the transmit path. However, given we cannot change the fundamental fact that only the VM can generate the data packets, the only other recourse then is to create a situation that encourages the VM to generate a lot of data packets and transmit them quickly. Achieving this is not easy though, since the VM itself will generally adhere to the standard TCP congestion control semantics such as slow start and congestion avoidance, and hence, cannot send too many data packets at the beginning. This behavior will continue even if the receiver advertises a large enough window, since a normal TCP sender is programmed to behave nicely to flows sharing the network path.

To address this problem, we propose a solution called *vFlood*, in which we make a small modification to the sending VM's TCP stack that essentially "offloads" congestion control functionality to the driver domain. Specifically, we install a small kernel module that replaces the TCP congestion control functionality in the VM with one that just "floods" the driver domain with a lot of packets that are subsequently buffered in the driver domain. The driver domain handles congestion control on behalf of the VM, thus ensuring the compliance of TCP semantics as far as the network is concerned.

There are two other challenges that still remain. First, buffer in the driver domain is a finite resource that needs to be managed *across* different VMs and connections in a fair fashion. Thus, no one connection should be able to completely occupy all the buffer space, which would prevent other connections from taking advantage of vFlood. Second, there has to be a flow control mechanism between the VM and the driver domain that would prevent the VM from continuously flooding the driver domain. This is especially important for connections that have low bottleneck network capacity. To solve the first problem, we propose to use a simple buffer allocation policy that ensures some free space to be always available for a new connection. We solve the second problem with the help of a simple backchannel through which the driver domain can easily communicate with the VM about when to stop/resume the flooding.

We have developed a prototype of vFlood in Xen [13]. Our implementation of vFlood required only about 1500 lines of code, out of which 40% was reused from Xen/Linux code base. Using this prototype, we performed extensive evaluation at both TCP flow and application levels. For the flow-level experiments, vFlood achieves about 5× higher median TCP throughput for 100 KB flows compared to the vanilla Xen. Our application-level evaluation with the Apache Olio [2, 42] Web 2.0 benchmark shows that vFlood improves its performance by 33% over the default Xen.

While we have so far discussed the receive and transmit *independently*, in general, a given VM may have some TCP connections that are transmit-intensive, some receive-intensive, and some which involve both simultaneously. Thus, in a real system, we need an integrated version of vFlood and vSnoop. A curious question here is how an integrated version compares with vFlood or vSnoop alone, i.e., whether the resulting benefits are cumulative or not. We have also implemented an integrated version of vSnoop and vFlood in our prototype system, and our evaluation (with vSnoop only, vFlood only, and vSnoop+vFlood configurations) shows that they do indeed yield orthogonal, non-counteracting performance improvements, with the integrated system improving the performance of the Apache Olio benchmark by almost 60% compared to about 33% by vFlood alone and 26% by vSnoop alone.

2. vFlood MOTIVATION

We motivate the problem and the need for vFlood using an example shown in Figure 1. We first focus on the "vanilla" case shown in the figure on the left. In this scenario, we consider three VMs labeled VM1-VM3 sharing a CPU. Assume a TCP sender in VM1 is transmitting packets to a remote TCP receiver not in the same physical host. In this case, according to the standard TCP semantics, the TCP sender will start conservatively with one (or a few depending on the TCP implementation) packet at the beginning of the connection. In many VMMs (e.g., Xen), a data packet passes via a buffer that is shared between the VM and the driver domain (e.g., the ring buffer in Xen). Once the driver domain is scheduled, it will transmit the packet on the wire towards the TCP receiver.

Since each CPU scheduling slice is typically in the *order of milliseconds* (e.g., 30ms in Xen), and network RTTs in a datacenter are typically sub-millisecond, the ACK packet

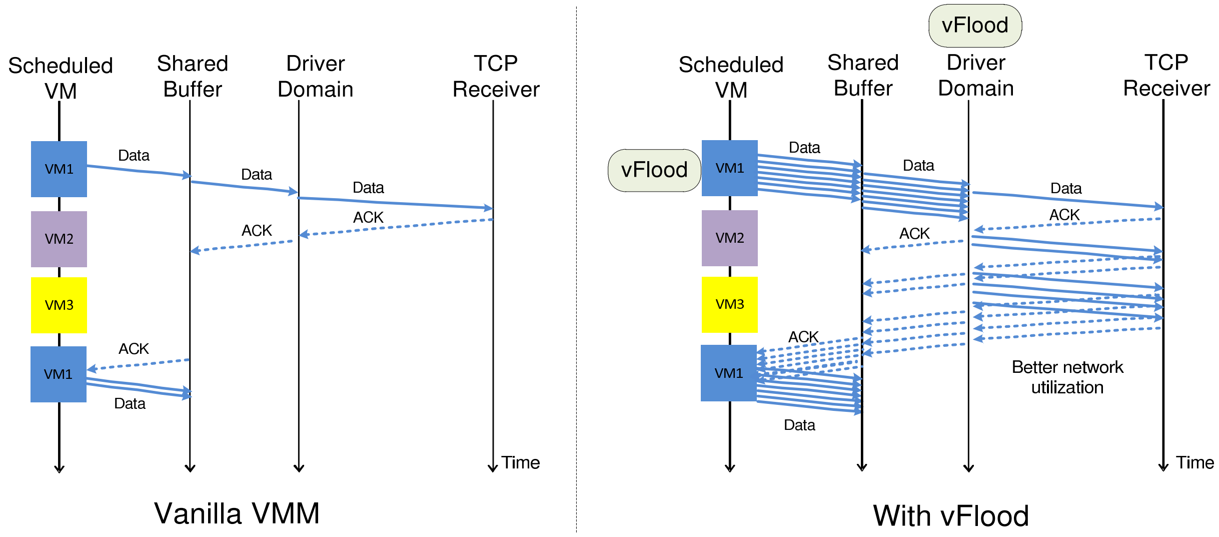


Figure 1: Illustration of TCP connection progress with vanilla VMM and with vFlood-enabled VMM.

may arrive quite quickly but may not find VM1 running at that instance. Consequently the packet will be buffered by the driver domain. Later when VM1 gets scheduled, this packet will be consumed. Unfortunately, as shown in the figure, this delay could be as high as 60ms if both VM2 and VM3 use up the entire slice of 30ms (as in Xen). Once the ACK packet arrives at VM1, VM1 will increase its congestion window according to the TCP slow start semantics and will send two new packets. Assuming network RTT is 1ms, the ACKs for these two packets may arrive 1ms later from the network, but they may have to get buffered until VM1 gets scheduled to process them further.

As a result of this additional scheduling latency, the progress of the TCP connection is severely hampered. Had there been no virtualization, the TCP sender would have doubled the congestion window every 1ms during the slow start phase thus ramping up quickly to the available bandwidth. Under the VM consolidation scenario with 3 VMs and CPU slice of 30ms, we can see that in the worst case the TCP sender doubles congestion window potentially every 60ms. Extending this argument to the TCP congestion avoidance phase, we can find that every 1ms (true network RTT), TCP will grow its congestion window by 1 MSS, whereas the same may happen every 60ms due to the additional latency of CPU scheduling among the VMs. The slow ramp up of the connection will negatively affect the overall TCP throughput, especially for small flows which spend most of their lifetime in TCP slow start. Recent studies on datacenter network characteristics [27, 14] indicate that the majority of flows in datacenters are small flows, suggesting that the impact of CPU sharing on TCP throughput is particularly acute in virtualized datacenters.

2.1 Possible Approaches

We now discuss some possible approaches to address this problem. We group them into three categories depending on the layer at which the approach resides.

TCP At the TCP layer, one possible approach is to turn off TCP slow start completely, and start with a reasonably high value for the congestion window. While it may mitigate the problem to some extent, this approach will lead to a congestion collapse in the network as each connection, irrespective of the congestion state in the network, will start flooding a large number of packets. This is especially undesirable in datacenter networks that often employ switches with shallow buffers for cost reasons [11]. For this reason, we choose not to disable TCP slow start. More generally, it takes decades to perfect protocols such as TCP, and a cursory fix to TCP such as shutting off slow start may result in undesirable and even unpredictable consequences. Of course, it may be interesting to conduct a more careful investigation to see if we can make changes at the TCP layer to address this problem.

VM Scheduler At the VMM layer, one option could be to modify the scheduler to immediately schedule the VM for which an ACK packet arrives, so that it can quickly respond to the ACKs by sending more data. Unfortunately, this option is prone to severe context switch overheads as the scheduler needs to keep swapping the VMs in response to the packets on the wire, making it practically infeasible. A variation of this idea is to make the scheduler communication-aware and prioritize network intensive VMs when making scheduling decisions. This approach does not incur aforementioned overhead of additional context switching. Indeed, Govindan *et al.* have proposed modifications to Xen's Simple Earliest-Deadline First (SEDF) CPU scheduler to make it more communication-aware by preferential scheduling of receiver VMs and anticipatory scheduling of sender VMs to improve the performance of network-intensive workloads [19]. While their scheduling mechanism achieves high performance for VMs with network-intensive workload, the improvement may come at the expense of VMs running latency-sensitive applications with little network traffic [39]. We do however believe that a communication-aware VM scheduler will create more *favorable* conditions for vFlood; we will investigate such an integration in our future work.

Hardware (TOEs) One other possible approach adopted by modern TCP offload engines (TOEs) offered by different vendors (e.g., [1, 3]) is to implement TCP in the network interface cards directly. While TOEs are actually designed for a different purpose, to reduce the CPU overhead involved in TCP processing, they do mitigate our problem to some extent. However, TOEs come with several limitations such as requiring to modify the existing applications in order to achieve improved performance [41], lacking flexibility in protocol processing (such as pluggable congestion control, netfilter and QoS features available in Linux), and being potentially prone to bugs that cannot be fixed easily [37]. Interestingly, popular virtualization platforms, such as VMware ESX and Xen, still *do not* fully support offloading complete TCP processing to hardware [21, 9]. Linux also does not natively support full TCP stack offload due to various reasons such as RFC compliance, hardware-specific limitations, and inability to apply security patches by the community (due to the closed source nature of TOE firmware) [7]. An alternative approach, motivated by the presence of many cores in the modern processors, is TCP onloading [40, 41], where TCP processing is dedicated to one of the cores. Since onloading requires extensive modifications to a guest VM's TCP stack, it is also not widely adopted.

2.2 Key Intuition behind vFlood

Based on the above discussion, we opt for a solution that (1) does not change the TCP protocol itself, (2) does not require hardware-level changes such as TOEs, and (3) does not modify the VMM-level scheduler. Our approach relies on the *key observation* that some components of a virtualized host get scheduled more frequently than the VMs. For instance, the driver domain in Xen (also the VMkernel in VMware ESX, the parent partition in Microsoft Hyper-V, etc.) is scheduled very frequently in order to perform I/O processing and other management tasks on behalf of all the VMs that are running on the host. Although not shown in Figure 1 for clarity, the gaps between VM slices are essentially taken by the driver domain. (In Xen, the driver domain can get scheduled every 10ms – even though the scheduling slice time may be 30ms – so that it can process any pending I/O from the VMs.) This observation suggests the following idea: If the driver domain, upon the arrival of an ACK packet from the TCP receiver, can push the next data segment(s) *on behalf* of the sending VM, the connection will make much faster progress and be largely decoupled from the scheduling/execution of the VM. vFlood is proposed exactly to realize this idea.

Moreover, since the driver domain itself cannot generate data on behalf of the sending VM, the data generated by some application in the VM must first be pushed to the driver domain. (We show one convenient approach to achieve this in Section 3.) The resulting progress of the TCP connection is shown on the right side of Figure 1: The driver domain, on behalf of the VM, can emulate the same TCP slow start and congestion avoidance semantics in response to ACKs that are arriving from the TCP receiver, achieving faster TCP connection progress and higher TCP throughput, which approaches the throughput achieved by a non-virtualized (but otherwise the same) TCP sender. The figure also shows that the interactions between the driver domain and the TCP receiver are compliant with the TCP standards.

3. vFlood DESIGN

vFlood essentially offloads the TCP congestion control functionality from the sender VM to the driver domain of the same host. Under vFlood, the sender VM is allowed to opportunistically flood data packets at a high rate to the driver domain during its CPU time slice, while the driver domain will perform congestion control on behalf of the VM to ensure TCP congestion control semantics are followed across the network. To realize congestion control offloading, vFlood needs to accomplish three main tasks:

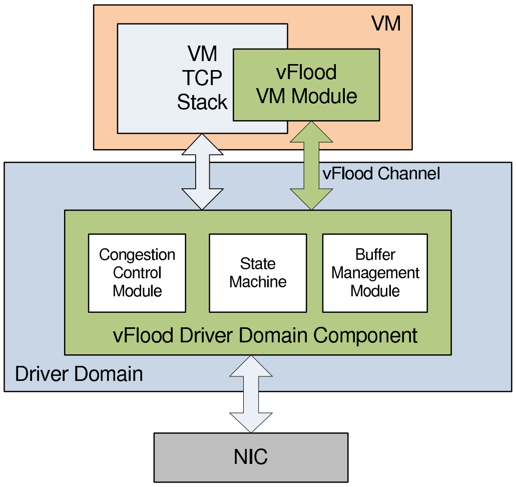


Figure 2: vFlood architecture

(1) Enable the VM adopt an *aggressive* congestion control strategy during slow start and congestion avoidance phases of a TCP flow;
(2) Implement a standards-compliant congestion control strategy in the driver domain on behalf of the vFlood-enabled VMs;
(3) Manage buffer space for the flooded data in the driver domain so that tasks (1) and (2) are performed in a coordinated way.

vFlood accomplishes the above tasks using three main modules as shown in Figure 2: (1) a *vFlood VM module* that resides within the VM and its main responsibility to shut off the default congestion control in the VM and flood the driver domain as fast as allowed; (2) a *congestion control module* in the driver domain that performs TCP congestion control on behalf of the VM; and (3) a *buffer management module* in the driver domain that controls the flooding of packets so that the buffer space for flooded packets is used fairly across all connections and VMs. We will discuss each of these modules in detail for the remainder of this section. We note that the generic design of these modules demonstrates vFlood's applicability to a wide class of virtualization platforms (e.g., Xen, VMware ESX, and Hyper-V). We defer the discussion of platform-specific implementation details to Section 4.

3.1 vFlood VM Module

The vFlood VM module resides in the VM and its main responsibility is to opportunistically flood the driver domain with TCP packets when the VM is scheduled. For packet flooding, vFlood modifies the standard congestion control behavior of the VM so that transmissions are done in a more aggressive fashion as long as such a strategy does not exhaust driver domain and network resources. This task can be conveniently implemented by installing a kernel module within the VM that effectively replaces the VM's congestion control function with a customized one that sets the congestion window to a high value (e.g., $cong\_win_{vm}$=512 segments) whenever desired (detailed conditions in Section 3.2). We note that, for most operating systems, such replacement is quite straightforward as the TCP congestion control functionality is usually implemented as a pluggable module so that an operating system can easily be configured with different congestion control implementations (e.g., Reno, Vegas [15], CUBIC [24], FastTCP [26]). In our Linux-based implementation, vFlood leverages the same interface as other popularly-used congestion control implementations in Linux to interact with the kernel. Note that only the congestion window management functions, such as growing and shrinking window in response to ACKs and packet losses, are replaced and offloaded to the driver domain; whereas the VM's TCP stack still implements all other functionalities required for reliable transmission such as timeout and retransmission.

As discussed earlier, such a minimal change to VMs is unavoidable for vFlood to realize opportunistic flooding and congestion control offloading. However, we deem such a design quite reasonable in virtualized cloud platforms where the guest kernel of a VM can be *customized* for better performance, security, and manageability. In paravirtualized VMMs such as Xen, the guest kernel is already patched for optimized interactions with the underlying VMMs; so intuitively, one can think of our approach as *paravirtualizing the TCP stack* to some extent. Our approach is also somewhat similar to the VMware tools [10], in which a set of system tools are installed in a VM to improve the VM's performance.

Note that our approach requires *no modifications* either to the applications or to the TCP protocol itself – all we require is installing a small kernel module in the guest OS that essentially dumbs down the congestion control portion of TCP, and a module within the driver domain for congestion control offloading. Thus, our approach is not as radical as TOEs or implementing a new variant of TCP.

3.2 Congestion Control Module

The vFlood congestion control module in the driver domain mainly performs the offloaded TCP congestion control function. Upon arrival of ACKs from a TCP receiver, the congestion control module will transmit packets that have already been flooded from the sender VM. However, unlike the artificial congestion window set in the VM ($cong\_win_{vm}$), the congestion window maintained by the driver domain ($cong\_win_{drv}$) grows and shrinks according to TCP standards and appears to the network and the receiver as the *actual value* used for the end-to-end connection. With this design, one can see that the presence of vFlood does not lead to any violation of end-to-end TCP semantics and would yield an approach that is at most as aggressive as a TCP sender from the driver domain (or from a non-virtualized sender).

vFlood relies on the *assumption* that the driver domain has sufficient memory and computation resources to buffer packets flooded by the VMs and to perform congestion control functions on behalf of them. Given that TCP processing overheads are typically dominated by the checksum computation and segmentation, and that these tasks are increasingly delegated to hardware in modern NICs, we believe that this assumption is quite reasonable. However, vFlood still has to carefully manage the finite amount of buffer space for flooded data among multiple connections. (We discuss this issue in Section 3.3.) In addition, vFlood's design requires an additional *communication channel* (referred to as *vFlood_channel*) between the congestion control module in the driver domain and the vFlood VM module. This channel is used by the driver domain to set $cong\_win_{vm}$ (Section 3.3), which effectively throttles the VM's transmission rate based on available buffer space. Meanwhile, the VM module uses the channel to signal the congestion control module when to go offline/online.

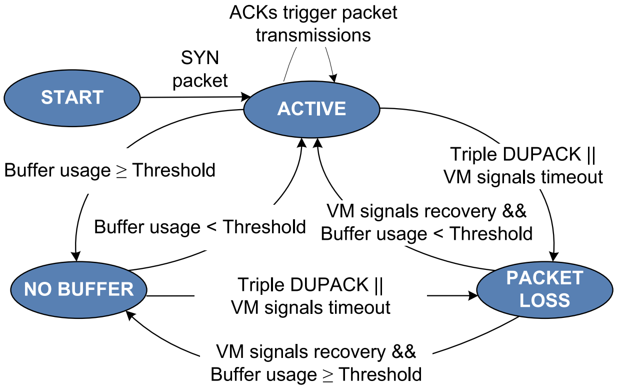

ACKs trigger packet transmissions
START
SYN packet
ACTIVE
Buffer usage ≥ Threshold
Triple DUPACK || VM signals timeout
Buffer usage < Threshold
VM signals recovery && Buffer usage < Threshold
NO BUFFER
Triple DUPACK || VM signals timeout
PACKET LOSS
VM signals recovery && Buffer usage ≥ Threshold

Figure 3: vFlood state machine

vFlood State Machine vFlood effectively de-synchronizes the TCP sender and receiver that are usually tightly coupled (and hence, susceptible to VM scheduling latencies) with the help of a state machine (Figure 3). For each flow, vFlood maintains a small amount of state so that it can enable or disable congestion control offloading depending on the state. In addition to the standard congestion control state maintained, notably congestion window and slow start threshold, vFlood's congestion control module keeps track of a few other variables: (1) the highest sequence number acknowledged by the receiver, and the number of times a packet with this sequence number has been acknowledged (for counting duplicate ACKs), (2) the count of unacknowledged, transmitted packets, (3) the advertised receive window of the receiver, (4) the window scaling factor, and (5) buffer usage of a flow. These values collectively determine when the congestion control module should actively engage in congestion control on behalf of the sender VM, and when it should go offline and let the VM re-take the congestion control responsibility. The driver domain initializes the per-flow state upon receiving a SYN or SYN-ACK packet and sets the flow's state to ACTIVE.

Online Mode While a flow is in the ACTIVE state, vFlood buffers all packets coming from the VM (subject to the buffer-management policy described in Section 3.3) and performs congestion control and packet transmission. The flow remains in the ACTIVE state until the flow experiences one of the two conditions: First, the network bottleneck capacity of the flow may decrease, in which case, the flow may start to exceed its share of the buffer usage (i.e., the per-flow buffer occupancy threshold defined in Section 3.3) leading to buffer space overflow. In this case, the vFlood VM module cannot continue to assume a large window size and pump more packets to the driver domain. Second, depending on the severity of the network congestion, one or more packets may be dropped. In this case, the TCP receiver will continue to send duplicate acknowledgements for each of the subsequent packets received after the dropped packet. In case of three dup-ACKs, standard TCP senders would trigger a fast retransmit and cut the congestion window by half, and in case of a timeout, they would trigger a retransmission but will cut the congestion window to 1MSS and switch to slow start.

Buffer space overflow is easy to detect in the driver domain. Once the congestion control module becomes aware of such an overflow, it will go offline by setting the state to NO BUFFER and signaling the vFlood VM module to switch to standard TCP congestion control. While detecting triple duplicate ACKs can also be done easily based on the state maintained, it is unfortunately not easy to detect timeouts since it requires timers and RTT estimators. Additionally, to perform retransmissions, all unacknowledged packets need to be buffered. To keep vFlood's design lightweight, we resort to the sender VM's TCP stack for handling these pathological situations: Once the VM's TCP stack detects a timeout, it will notify the vFlood VM module, which will in turn notify the congestion control module in the driver domain via *vFlood_channel*. The congestion control module will then go offline (PACKET LOSS state) and transfer congestion control back to the sending VM.

Offline Mode When vFlood goes offline (i.e., when it switches to NO BUFFER or PACKET LOSS state), the congestion control module in the driver domain operates in a pass-through mode, where it does not transmit any new packets and instead lets the sending VM do that. However, notice that the sender VM's congestion control has been replaced by the vFlood VM module that sets a larger congestion window than the driver domain. To solve this problem, once congestion control responsibility is transferred back to the sender VM, it will stop using the more aggressive congestion window. Instead, the vFlood VM module will function like the original TCP stack without vFlood. To achieve this effect, vFlood basically employs a shadow variable $cong\_win_{vm}'$ that, similar to $cong\_win_{drv}$, grows and shrinks according to TCP semantics. This variable is in addition to $cong\_win_{vm}$ that is used when vFlood is online (i.e., in ACTIVE state). Note that $cong\_win_{vm}'$ may not be exactly the same as $cong\_win_{drv}$ at any given instance due to the slight lag in TCP processing, but are going to be roughly similar since the driver domain and the VM see the same sequence of events.

Maintaining the extra shadow variable above does not cause much overhead as it only entails growing $cong\_win_{vm}'$ by 1 MSS for every ACK during slow start and by 1 MSS for each RTT during congestion avoidance. Therefore, when the driver domain module goes offline, the vFlood VM module can seamlessly take over. In some sense, the vFlood VM module still retains the full-fledged TCP congestion control mechanism; however, when conditions are right, it will switch back to the flooding (online) mode and offload congestion control back to the driver domain. Upon receiving a notification from the sender VM that the lost packets have been recovered or upon detecting available buffer space, the vFlood driver domain module will become online again and resume its congestion control duty.

Adjusting Receive Window One issue that we have not discussed so far is the value of the advertised receive window sent in the ACKs. Given that a TCP connection's *send window* is the minimum of its *congestion window* and the *receiver advertised window*, we also have to modify the receiver advertised window in order to make the VM TCP stack flood packets to the driver domain. One solution is to rewrite ACK packets' receive window field at the driver domain module while vFlood is online, so that the small receiver advertised window does not inhibit flooding. However, rewriting receive window invalidates a packet's TCP checksum, so vFlood either has to recalculate the checksum or require checksum validation at the physical NIC. This solution might also lead to retransmission of more packets than expected by the receiver in a situation where the VM detects packet losses and retransmits based on the inflated value written by the driver domain. Hence, similar to the congestion window maintenance, vFlood's VM module maintains two variables. One variable corresponds to the actual receiver advertised window as read from incoming ACKs and is used when vFlood is offline (i.e., vFlood enters PACKET LOSS or NO BUFFER state). The other variable corresponds to the buffer size in the driver domain and is used when vFlood is online (i.e., in ACTIVE state).

Choice of Congestion Control Algorithm Another issue concerns the exact congestion control algorithm used in the driver domain out of many possible variants. In order to ensure maximum flexibility in a cloud environment, we assume the driver domain implements almost all the standard algorithms that any stock Linux or Windows kernel implements (e.g., Reno, NewReno, Vegas, CUBIC). Also we have modularized the design of the congestion control module in the driver domain such that plugging-in a new TCP congestion control algorithm is fairly easy. That way, an appropriate congestion control algorithm can be configured easily when vFlood is enabled as an option. We can even run different congestion control algorithms simultaneously depending on what the VMs desire thus ensuring functional equivalence between the online and offline modes of vFlood.

3.3 Buffer Management Module

Finally, vFlood uses the buffer management module to regulate the buffer usage across different flows, both within and across VMs. The need for this module stems from the fact that, in the ACTIVE state, the sender VM acts as the *producer* and the network becomes the *consumer* of packets for a given flow (according to TCP semantics). If the consumer process (network) becomes too slow, it would lead to exhaustion of buffer resources in the driver domain. This problem becomes more acute when there are multiple guest VMs, each with multiple active TCP flows, contending for the same buffer space. Note that this problem does not arise in our receive-path solution vSnoop [28]. More specifically, on the receiver side, the buffer space is limited by the amount of space in the shared ring buffer (in Xen), or whatever other mechanism a given VMM uses to transfer between the VMM and the receiver VM (whether vSnoop is used or not). However, in vFlood, without care, one slow flow (perhaps with a bottleneck in the network) can occupy all the buffer space thus denying the benefits of congestion control offloading to other flows—the scenario we wish to avoid.

In vFlood, we provide two levels of isolation. First, in order to guarantee complete isolation for flows belonging to *different VMs*, vFlood assigns to each VM its own dedicated buffer space (e.g., a 4MB buffer) that is not shared with other VMs. This is similar in spirit to the approach many VMMs (e.g., Xen) take to provision ring buffers for network transfers between the driver domain and the VMs.

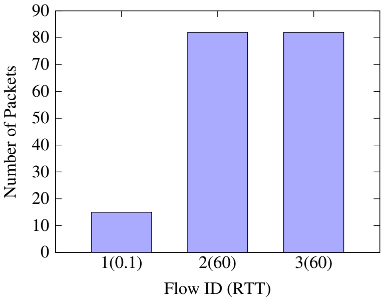


Figure 4: Unfairness in buffer occupancy when no buffer management policy is in place

As a result, no one VM can greedily occupy all the available buffer at the expense of other VMs. Second, we provide a mechanism to manage buffer usage across flows that belong to the *same VM*. Note that it does not make sense to strictly partition the buffer across flows since the number of active flows may be large and not all flows benefit equally from vFlood. In particular, we note that high RTT connections and low bottleneck bandwidth connections benefit less from vFlood, and also consume more buffer space since the discrepancy between the rates at which the sender VM produces and network drains is the most for such flows.

To understand how RTTs affect the buffer usage, we conduct a simple experiment where a VM sends a 2MB file to three different receivers simultaneously. The first flow has 0.1ms RTT while the other two flows have 60ms RTT. We ensure that all three flows have *similar bottleneck capacity*. Figure 4 shows the buffer occupancy of the vFlood per-VM buffer space for all three flows. As illustrated, flows 2 and 3 occupy a larger share of the buffer space compared to flow 1. The larger buffer occupancy stems from the fact that $cong\_win_{drv}$ grows more slowly for the high-RTT flows. As a result, it takes longer to transmit buffered packets for flows 2 and 3. These high RTT connections will prevent a low RTT high bandwidth connection from taking better advantage of vFlood by occupying more available buffer space.

Buffer Allocation Algorithm Interestingly, the high-level problem faced by vFlood buffer management module is conceptually similar to the problem of buffer allocation confounded by network routers, where different flows compete for the same set of buffer resources. While many ideas have been proposed in that context, a simple yet elegant scheme that we can borrow is due to Choudhary and Hahne [17], where the amount of available free space serves as a dynamic threshold for each flow. Specifically, in this scheme, the buffer usage threshold T_i for flow i is defined as

$$T_i = \alpha_i \cdot (B - Q(t)) \tag{1}$$

where B is the total buffer size, $Q(t)$ is the total buffer usage at time t, and α_i is a constant which can be set according to the priority of flows. Thus, if only a single flow is present, it can occupy up to one half of the total buffer space (assuming $\alpha_i = 1$). As soon as a new flow arrives, both flows can occupy only 1/3rd of the total buffer space, with the remaining 1/3 reserved for future flows. The main advantage of this scheme lies in the fact that some amount of buffer is always reserved for future flows, while the threshold dynamically adapts based on the number of active flows.

In vFlood's buffer management module, we implement a similar scheme that determines how many more segments the driver domain can receive for flow i ($pkt\_in^i_{VM}$) from the VM module as:

$$pkt\_in^i_{VM} = \underbrace{cong\_win^i_{drv} - pkt\_out^i_{drv}}_{\theta_i} + \underbrace{T_i - Q_i(t)}_{\phi_i} \tag{2}$$

where $cong\_win^i_{drv}$ is the congestion window for flow i at the driver domain, $pkt\_out^i_{drv}$ is the number of transmitted, unacknowledged packets for flow i as maintained by the driver domain, T_i is the buffer threshold as defined by Equation 1, and $Q_i(t)$ is the buffer usage of flow i at time t. In Equation 2, the term θ_i refers to the number of segments that can be sent immediately, without any buffering, while the term ϕ_i refers to available buffer space for flow i. Consequently, the congestion window for flow i at the VM module is defined as:

$$cong\_win^i_{VM} = pkt\_in^i_{VM} + pkt\_out^i_{VM} \tag{3}$$

where $pkt\_out^i_{VM}$ is the number of transmitted, unacknowledged packets for flow i as maintained by the VM module while vFlood is online. To implement a *fair buffering* policy among all VM flows, it suffices to pick the same value for α_i (Equation 1) for all flows. With this policy, flow 1 in Figure 4 would have more buffer space and would benefit more from the presence of vFlood.

Prioritized Buffering Policy In general, however, not all flows benefit equally from the buffer allocation. We can broadly categorize flows into four classes depending on their RTT (high/low) and bottleneck capacity (high/low). Flows with high RTT or low bottleneck capacity will occupy significant buffer space. However, the benefit of allocating buffer for these type of flows is relatively minimal as: (1) For high RTT flows, the VM CPU scheduling-induced extra latency is not dominant with respect to the already high network RTTs. (2) For low bottleneck capacity flows, pushing the congestion control to the driver domain is not likely to improve TCP throughput anyway. Therefore, flows with low RTT and high bottleneck capacity would mainly benefit from vFlood. Fortunately, much of the traffic in datacenters fall into this category.

Given this classification, we devise a *prioritized buffering* policy where vFlood favors low-RTT flows by picking larger values for α_i for them. While we can do similar prioritization for high-bandwidth flows, it is not practical due to two reasons. First, detecting bottleneck bandwidth is harder to do in the driver domain and requires keeping additional state. One could implement rate estimation as in TCP Vegas [15], but Vegas assumes rate increases by 1 MSS every other RTT to ensure stable rate calculation which is different from Reno semantics. Further, it requires additional timing variables making it, although not completely impossible, a bit more tedious to perform. A bigger problem, however, is that the rate estimation during slow start – arguably the stage where vFlood benefits the most – is unreliable since the window size has not stabilized yet. Thus, we advocate classifying flows mainly based on their RTTs without considering their bottleneck capacities.

We note that in typical cloud environments, due to over-provisioning of datacenter networks (to provide full bisection bandwidth within a datacenter), intra-data center capacities are much higher compared to the lower-bandwidth cross-datacenter links. Further, intra-datacenter RTTs are typically much lower than across data centers; thus, high

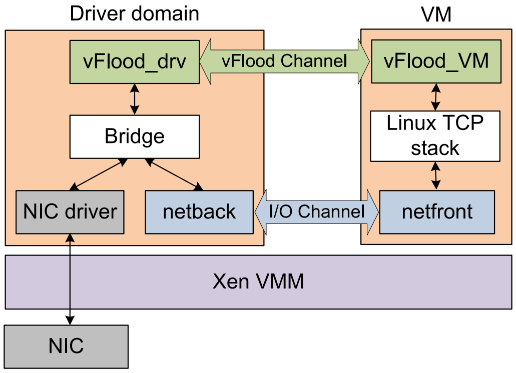


Figure 5: vFlood implementation on Xen

(low) bandwidths are positively correlated with low (high) RTTs. Thus, focusing on RTT-based prioritization alone works reasonably well in our settings. In our evaluations, we consider 1.0ms as a reasonable threshold to distinguish high and low RTTs.

4. vFlood IMPLEMENTATION

We have implemented a prototype of vFlood with paravirtualized Xen 3.3 as VMM and Linux 2.6.18 as the guest OS kernel. One of our main implementation goals is to minimize the code base of vFlood and maximize the reuse of existing Xen and Linux code. In fact, out of approximately 1500 lines of vFlood code (according to cloc tool [4]), 40% is directly reused from existing Xen/Linux code. The reused code includes part of Linux's Reno congestion control implementation and flow hashing functionality; and Xen's I/O ring buffer management.

As shown in Figure 5, Xen adopts a *split driver* model for paravirtualized devices. Each virtual device (e.g., a virtual network or block device) has a front-end interface in the VM and a back-end interface in the driver domain (dom0). The communication between the two interfaces takes place via the following modules: (1) a *ring buffer* that holds descriptors of I/O activities in one direction (i.e., from front-end to back-end or vice versa), (2) a *shared page* referenced by ring buffer that holds the exchanged data (e.g.. a network packet, disk block, etc.), and (3) an *event channel* that serves like an interrupt mechanism between the two ends.

When a VM transmits a packet, the packet gets placed on the shared page between the VM and dom0 and an *event* (a paravirtual IRQ) is sent to dom0. Upon receiving the event from the VM, dom0 constructs a socket buffer (*sk_buff*) kernel structure for the packet and passes it to the Linux bridge module en route to the network interface card (NIC). Similar type of activities takes place on the receive path but in the reverse order, where an *sk_buff* structure arrives at the bridge from the NIC, the bridge identifies the destination VM and passes it to the corresponding back-end interface. Finally the back-end interface delivers the packet to the VM via the front-end interface. We next describe the implementation of different modules of vFlood outlined in Section 3.

vFlood VM Module Replacing the default congestion control module in the VM is the only modification we made to the VM. As we briefly alluded to in Sections 3.1 and 3.2, the vFlood VM module maintains two congestion windows for each flow. $cong\_win_{vm}$ is used when vFlood is online for flooding packets to the driver domain; while $cong\_win_{vm}'$ is maintained according to the TCP Reno specification and will be used when vFlood goes offline.

The vFlood VM module interacts with the guest OS kernel through the same interface used by the standard congestion control modules, hence it does not require any modifications to the TCP/IP stack of the VM. Additionally, this module interacts with the driver domain via the communication channel *vFlood_channel*, whose implementation will be described shortly. Both the VM and driver domain modules of vFlood maintain a control structure for each TCP flow to store the per-flow state. Each control structure is accessed by a hash function that takes as input the source/destination IP addresses and port numbers of a given flow.

Congestion Control Module This driver domain module of vFlood is implemented as two hook functions to the Linux bridge module. *vFlood_tx* intercepts all packets on the transmit path and performs congestion control for VM flows for which vFlood is online. More specifically, upon receiving a packet from the sender VM, *vFlood_tx* transmits the packet immediately if allowed by the Reno congestion control algorithm (i.e., based on the congestion window, advertised receive window, and the number of transmitted, unacknowledged packets); otherwise, it buffers the packet. *vFlood_rx* intercepts all packets on the receive path and performs three main tasks. First, as ACKs arrive, *vFlood_rx* updates $cong\_win_{drv}$ per TCP Reno semantics. Second, if allowed by the congestion control algorithm, it fetches packets from the per-flow buffer and transmits more packets. Third, it notifies the vFlood VM module via *vFlood_channel* of the available buffer allocation so that the vFlood VM module can adjust its congestion window accordingly.

vFlood_channel is implemented like a standard Xen device, similar to the virtual network device described earlier. One set of ring buffer and event channel is used for communication from a VM to dom0. Upon receiving an event on this channel, the event-handler at the congestion control module takes vFlood offline or online based on the command passed from the vFlood VM module. The other set of ring buffer and event channel is used for communication from dom0 to the VM. Upon receiving an event on this channel, the event-handler at the vFlood VM module adjusts $cong\_win_{VM}$ based on the buffer threshold and allocation passed from dom0. The only difference between *vFlood_channel* and a typical Xen device is that we use the ring buffer itself, not a separate shared page, for passing commands between the VM and dom0.

Buffer Management Module The main task of the buffer management module is to manage the per-VM buffer space in dom0. This module maintains a FIFO queue of *sk_buff* structures for each flow. Additionally, it keeps track of per-VM and per-flow buffer allocation and usage. As we described earlier, this module interacts directly with vFlood's congestion control module in dom0.

5. EVALUATION

In this section, we present a detailed evaluation of vFlood using our prototype implementation. Our evaluation is focused on answering the following key questions: (1) By how much does vFlood improve TCP throughput? (2) How does the TCP throughput gain translate into application-level improvements? (3) How much overhead does vFlood incur?

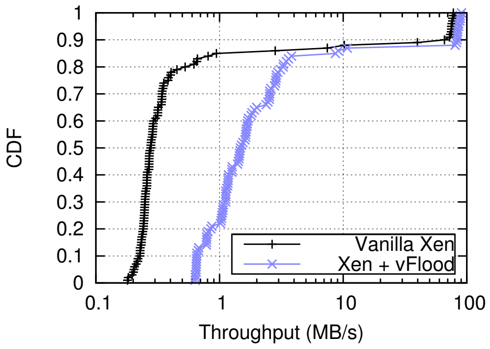


Figure 6: CDFs for 100 successive 100KB transfers with and without vFlood

Before we answer these questions, we first describe the experimental setup.

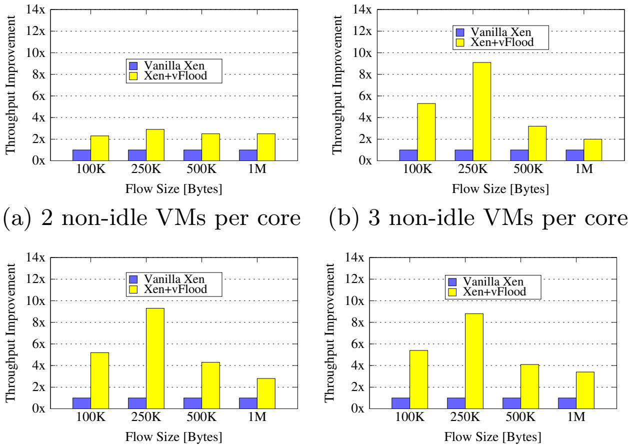
Experimental Setup Each server runs Xen 3.3 as the VMM, and Linux 2.6.18 as the operating system for the VMs and driver domains. For the experiments described in Section 5.1, we use a machine with dual-core 3GHz Intel Xeon CPU, 3GB of RAM as the server, while the client is a 2.4GHz Intel Core 2 Quad CPU machine with 2GB of RAM. For application-level experiments in Section 5.2, we use Dell PowerEdge servers with a 3.06GHz Intel Xeon CPU and 4GB of RAM. All machines are connected via commodity Gigabit NICs. In all experiments, we configure VMs with 512MB of memory and use TCP Reno implementation. In order to keep the CPU utilization at determined levels in our experiments, we use a load generator utility which can make the CPU busy by performing simple CPU bound operations. We selected 30ms as the duty cycle of this utility as it aligns with the VM scheduling time slice of Xen.

5.1 TCP Throughput Evaluation

This section presents our evaluation of TCP throughput improvement under a variety of scenarios. For experiments in this section, we allocate a 2048-segment buffer for each VM in the driver domain to support vFlood operations and use a custom application that makes data transmissions of different size (similar to Iperf [6]) over TCP sockets. For the rest of this section, we compare TCP throughput of the vFlood setup with the vanilla Xen/Linux setup.

Basic Comparison with Xen Due to small sizes of the flows we are experimenting with and the VM scheduling effects, the throughput results are subject to high variation during different runs of the experiment. Figure 6 shows the CDF of TCP throughput for one hundred 100 KB transfers from a VM to a non-virtualized machine with and without vFlood. In this experiment, the sending VM is sharing a single core with two other VMs with 60% CPU load (i.e. all VMs show 60% CPU utilization). The results indicate the high variability exists for both vanilla Xen and vFlood setups; however, vFlood consistently outperforms the vanilla Xen configuration. The median throughput achieved by vFlood is almost 5× higher than that of the vanilla Xen. For all the remaining experiments, we conduct 100 runs of each experiment and compare the median throughputs across the vFlood and vanilla Xen setups.

Number of VMs Per-Core In this experiment, we vary the total number of VMs running on the same core as the sender VM from 2 to 5 (including the sender VM) and fix the CPU load of each VM to 60%. The normalized performance gains of vFlood are shown in Figures 7(a), 7(b), 7(c) and 7(d), where 2, 3, 4 and 5 VMs share the same core. As we can see, vFlood results in significant improvement for all transfer sizes for different number of VMs per core.



(a) 2 non-idle VMs per core (b) 3 non-idle VMs per core

(c) 4 non-idle VMs per core (d) 5 non-idle VMs per core

Figure 7: TCP throughput improvement with different number of VMs per core.

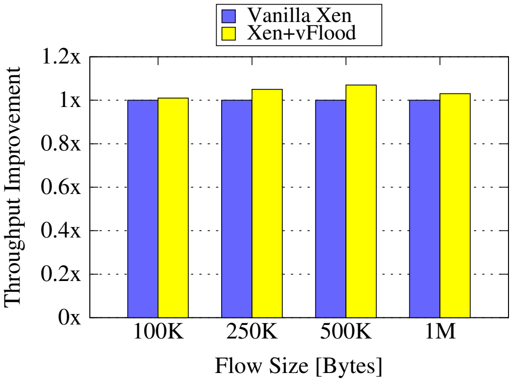
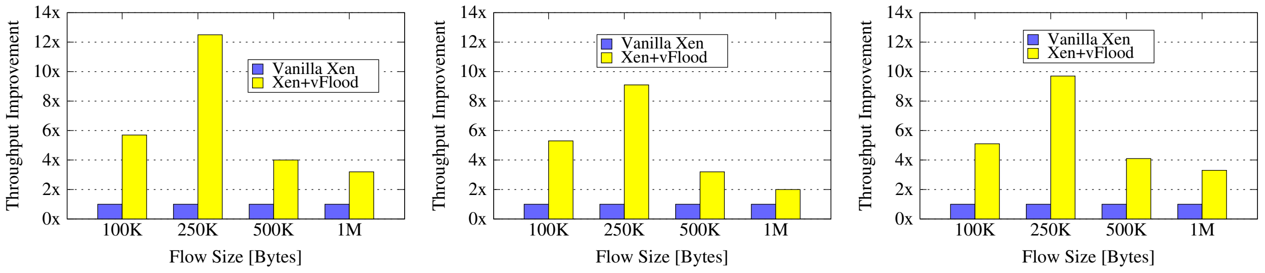


Figure 8: TCP throughput improvement for the 1-VM scenario

While it is indeed expected that vFlood performs well when there are a lot of VMs per core, one may guess that there will be no benefits of vFlood when one VM is running. However, as we can observe in Figure 8, even for the case where only one VM is running, vFlood is able to perform slightly better than vanilla Xen. We attribute this slight improvement to the fact that even in the 1-VM scenario the driver domain and the VM compete with each other to run on the same CPU. Therefore, while the scheduling delay for this scenario is not as high as the multi-VM scenarios, the driver domain can still process incoming ACKs more quickly and consequently can make faster transmissions compared to the vanilla Xen scenario.

Large Transfers Similarly, we also expect the most gains to be for short flows (which as we pointed out dominate data center environments). To study the benefits of vFlood for large transfers, we experimented with two sizes – 10MB and



(a) 3 VMs, 40% load

(b) 3 VMs, 60% load

(c) 3 VMs, 80% load

Figure 10: Performance of vFlood with varying load. We fix the number of VMs per core to 3.

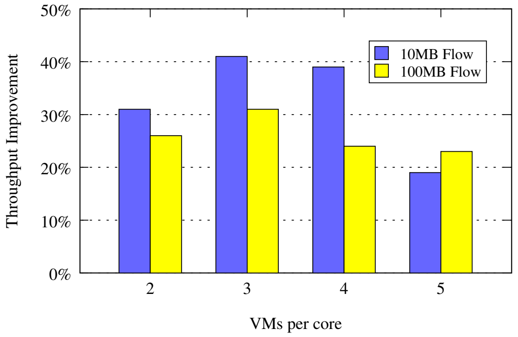


Figure 9: vFlood throughput improvement for large flows

100MB. Our results in Figure 9 shows that, even for large transfers, vFlood improves TCP throughput by 19% to 41%.

Varying CPU Load To study the benefits of vFlood across different CPU loads, we fix the number of VMs sharing the same core to 3 and set CPU load of each VM to 40%, 60% and 80%. Figure 10 shows the normalized throughput gain across these different loads, and shows vFlood outperforms the vanilla Xen setup significantly and consistently across all configurations. We observe that improvements are particularly high for 250KB transfers, with up to 12× for the 3-VM 40% load scenario.

To investigate further the cause behind this special case, we select one of the configurations (3 VMs sharing the same core, each with 40% CPU load) and study TCP throughput values when we vary the flow size all the way from 50KB to 1GB. Figure 11 shows the results. Interestingly, this figure shows when transfer size is about 340KB we obtain the maximum improvement. This phenomenon corresponds to the number of slots in Xen's ring buffers (240 slots), which leads to a maximum transfer of about 240 segments (of size 1500 bytes) within one VM scheduling interval.

Scalability of vFlood Most of the experiments described above consist of only one flow at a particular instance. In order to verify that vFlood scale well with the number of flows, we measured the throughput gains of vFlood when there are 10 and 100 concurrent flows from the same VM (with 3 VMs sharing the same core running 40% load). Figure 12(a) shows that as we scale up the number of flows, gains drop marginally, but still vFlood is able to produce significant throughput improvements compared to the vanilla Xen.

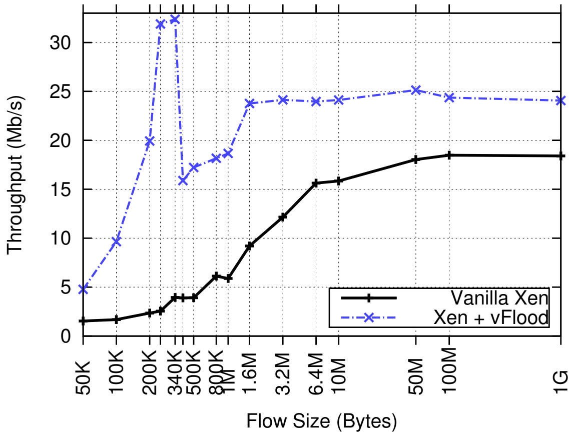
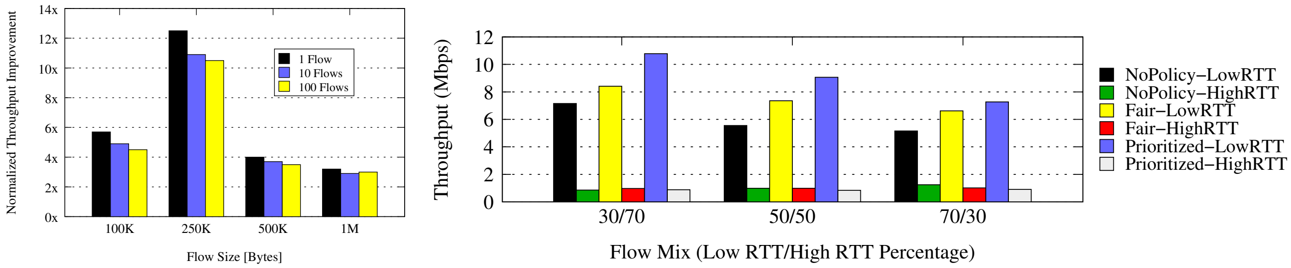


Figure 11: vFlood improvement for different flow sizes

Effectiveness of Buffer Management Policies In Section 3.3, we discussed the importance of having a buffer management policy, specially when high-RTT (low throughput) and low-RTT (high throughput) flows share the same per-VM buffer space. This section presents a comparison of the three buffering policies presented earlier, namely *no policy*, *fair policy* (i.e., with same α_i) and *prioritized policy* (with higher α_i for low RTT connections). For these experiments we run the sender VM and the low-RTT receiver in the same local area network, while for high-RTT connections we place the receiver on a remote PlanetLab node (*planetlab1.ucsd.edu*). In our implementation, we designate flows with RTT less than 1ms as low-RTT and other flows as high-RTT. Additionally, we dedicate a per-VM buffer of size 2048 segments for vFlood operations.

Figure 12(b) shows the median TCP throughput values for different buffering policies when the sender VM repeatedly (for a 2-minute period) starts 20 concurrent flows to local and remote receivers and transmits 500KB blocks of data. Our evaluation compares throughput values for the aforementioned policies under different flow mixes (i.e., different ratio of low-RTT to high-RTT flows). For low-RTT flows, we see improvements by going from *no policy* to *fair policy* and from *fair policy* to *prioritized policy* for all flow mixes. The benefits for low-RTT flows are the highest for the 30/70 mix where a naive policy would let the majority (70%) high-RTT flows steal buffer space from the minority (30%) low-RTT flows. We also note that different buffering



(a) Concurrent flows

(b) Buffer management policies

Figure 12: Subgraph (a) shows TCP throughput gains with concurrent flows. Subgraph (b) shows the comparison between the three buffer management policies

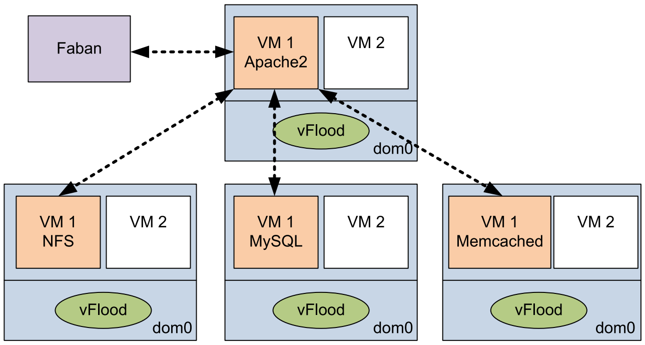


Figure 13: vFlood Apache Olio test setup

Operation	Count Vanilla Xen	Count vFlood	Count vSnoop	Count vFlood + vSnoop
HomePage	2544	3271	3416	4215
TagSearch	3290	4281	4020	5550
EventDetail	2363	3077	3135	3925
PersonDetail	219	331	312	410
AddPerson	53	96	71	123
AddEvent	156	245	178	257
Total	9512	12642	11940	15167
Rate(ops/sec)	31.7	42.1	39.8	50.5
Percentage Improvement	-	**32.9%**	**25.5%**	**59.5%**

Table 1: Apache Olio benchmark results

policies do not make any difference to high-RTT flows as for them network RTT dominates VM scheduling-induced RTT (Section 3.3).

5.2 Apache Olio Benchmark

We use the Apache Olio benchmark [2, 42] to show the effectiveness of vFlood for typical cloud applications. Apache Olio is a social-event calendar Web 2.0 application, where users can create, RSVP, rate, and review events. We use the PHP implementation for our experiments which includes four components: (1) An Apache Web server which acts as the request processor and web front-end, (2) A MySQL server that stores user information and event details, (3) An NFS server that stores user files and event specific data, and (4) A Memcached server to cache rendered page fragments.

Figure 13 shows our testbed configuration. We use four VMs on four distinct physical hosts to run each component of the Olio system. We also run one other VM (with 30% load) per physical server sharing the same core with Olio VM to trigger VM scheduling. vFlood is deployed on all physical servers so that the communication between each component as well as the communication between clients and the Apache web server can benefit from congestion control offloading. We allocate a 4096-segment buffer in the driver domain for each VM to support vFlood operations and use Faban [5] as the client load generator. Faban is configured to run for 6 minutes (30-second ramp up, 300-second steady state, and 30-second ramp down) during which 200 client threads generate different types of requests.

We evaluate the number of operations performed by Olio for three different configurations: (1) Vanilla Xen, (2) Xen with vFlood only, (3) Xen with vSnoop (Section 1) only, and (4) Xen with vFlood and vSnoop (whose integration is discussed in Section 6). Table 1 shows the total count of different operations performed by Olio. When vFlood alone is deployed in the system, we see total throughput rising from 31.7 ops/sec in the vanilla Xen configuration to 42.1 ops/sec (a 32.9% improvement). When vSnoop is deployed we see a 25.5% increase in total throughput. When both vFlood and vSnoop are deployed, TCP throughput improves on both receive and transmit paths and we see throughput rising to 50.5 ops/sec (a 59.5% improvement). Our results indicate that the effects of vFlood and vSnoop *complement* each other and the performance gains they achieve are cumulative.

5.3 vFlood Overhead

vFlood Routine	CPU Cycles	CPU %
vFlood_tx()	65	0.62
vFlood_rx()	370	3.05
vFlood_hash_lookup()	78	0.73
vFlood_update_VM()	59	0.56
vFlood_process_threshold()	57	0.92

Table 2: vFlood per-packet CPU usage

In order to understand the runtime overhead of vFlood, we use Xenoprof [33] to profile vFlood's overhead at both the VM and driver domain. We specifically use Xenoprof to measure CPU cycles consumed by different vFlood routines. Also we instrument main vFlood routines to record the number of packets they process. Table 2 shows *per-packet* CPU cycles consumed by different vFlood routines and their percentage of CPU usage when Iperf transmits for a 20-second period. From the table, we can see much of the overhead is associated with *vFlood_rx()* routine. This routine is responsible for intercepting acknowledgements destined to VMs, calculating congestion window, releasing buffered packets and notifying VMs about their buffer usage. On the other hand, the overhead caused by *vFlood_tx()* is minimal because this routine's primary responsibility is to queue packets coming from the VM. Our queuing mechanism (we reuse Linux sk_buff queuing mechanism *skb_queue_tail()*) also incurs negligible overhead. The function which is called by the driver domain whenever it needs to update the buffering threshold (*vFlood_update_VM()*) and the function which called by the VM to process buffer threshold information sent by the driver domain (*vFlood_process_threshold()*) also do not add much overhead.

6. DISCUSSION

VM Migration Given that the vFlood VM module also runs a fully functional standard congestion control algorithm in the background, the VM state is very much self-contained and can be migrated to other hosts. If we are to move a VM from a vFlood-enabled host to one without vFlood support, we only need to switch the vFlood VM module to the original congestion control mode before the migration. Moving a VM from a vFlood-enabled host to another vFlood-enabled host requires some state initialization at the driver component of the destination host for the existing flows. While the state needed for initialization can be migrated from the source host to the destination host, either by modifying the VM migration protocol, or by leveraging the *vFlood_channel* to transfer the state from VM module to the driver domain, we suspect that the benefits would be typically marginal as most flows in datacenter environments are fairly short-lived [27, 14]. Therefore, our current implementation supports live VM migration in a limited, yet effective fashion by taking vFlood offline for those active flows established before the migration.

Buffer Space Management Typically, if the number of VMs is small, the buffer space in the driver domain may not be an issue. In environments where the number of VMs may be large (say, 30-40), buffer space may become an issue. Thus, instead of making the buffer space increase proportional to the number of VMs, we can potentially allocate the per-VM vFlood buffer from the VM's own memory. In such a scheme, a VM can share one or multiple pages with the driver domain for buffering purposes (e.g., through the Grant Table facility in Xen) thus reducing vFlood's dependency on driver domain resources. Another advantage of this scheme is that during VM migration, the buffered regions can also be migrated with the VM as they are now part of the VM's address space.

vFlood and vSnoop Integration In section 5.2, we presented some promising results using a preliminary integration of vSnoop and vFlood. We found that the integration effort is non-trivial as they both operate on the same set of packets, and rely on some shared data structures for their operation. For example, an incoming ACK packet with data payload can trigger acknowledgement from vSnoop and a packet transmission from vFlood. Our preliminary implementation is based on a pipelined architecture where on the receive (transmit) path packets gets processed by vSnoop (vFlood) first and then by vFlood (vSnoop). This approach, however, does not implement features such as ACK piggybacking—combining pro-active vSnoop's ACKs with vFlood's data packets to reduce the number of packet transmissions. We are currently working on a more efficient solution based on an integrated state-machine that would collapse the different actions that vSnoop and vFlood would take, thus ensuring functional equivalence with a non-virtualized TCP stack in terms of number of packets on the wire.

Interplay with Emerging Hardware A few techniques have been proposed to give VMs direct access to specialized networking hardware (e.g., use of IOMMU-based SR-IOV in Xen 4.0 and VMDirectPath in VMware vSphere). While these techniques lower the network virtualization overhead by bypassing the driver domain or the hypervisor, they still do not address the significant increase in RTT due to VM CPU scheduling. In such settings, we envision implementing vFlood (except the vFlood VM module) combined with vSnoop in the hardware itself, thus eliminating the VMM overheads completely. We believe that the vFlood state machine described Section 3.2 should lend itself to a scalable hardware implementation. We will pursue this vision in our future work.

7. RELATED WORK

We have already discussed most of the work that is directly related to vFlood in Section 2.1. We now discuss other related efforts that fall into the general area of performance improvement for virtualized environments. We group them into three categories: (1) reducing virtualization overheads along the I/O path, (2) improving VMM I/O scheduling, and (3) optimizing TCP for datacenters.

Reducing Virtualization Overheads There exists substantial research focusing on optimizations that reduce overheads induced by virtualization along the I/O path. For instance, Menon *et al.* have proposed several optimizations to improve device virtualization using techniques such as packet coalescing, scatter/gather I/O, checksum offload, segmentation offload, and offloading device driver functionality to hypervisor [35, 32, 34]. vFlood is quite complementary to these techniques. By addressing the interplay between VM consolidation and network transport protocol, vFlood operates one level higher than those optimization techniques. XenSocket [46], XenLoop [44], Fido [16] and Xway [31] specialize in improving inter-VM communication when the VMs are all on the same physical host; vFlood is more general as it improves transport protocol performance regardless of where the other end of a connection is located. IVC [25] is another effort in this direction that targets high performance computing platforms and applications.

Improving VMM I/O Scheduling I/O scheduling for VMs has received significant attention. Some recent efforts include mClock [20] and DVT [29, 30]. mClock provides proportional-share fairness with limits and reservations to schedule I/O requests from VMs. DVT proposes the differential virtual time concept to ensure that VMs experience less variability in I/O service time. These solutions focus on modifying the VMM I/O scheduler, whereas vFlood is agnostic to the VMM's CPU and I/O schedulers.

Optimizing TCP for Datacenters Alizadeh *et al.* show that the traditional TCP falls short of handling flows requiring small predictable latency and flows requiring large sustained throughput due to TCP's demand on the limited buffer space available in datacenter network switches [11]. They propose DCTCP for datacenter networking, which leverages ECN capability available in the switches. Vasudevan *et al.* observe the "in-cast" problem where multiple hosts send bursts of data to a barrier-synchronized client, thus causing overflows in Ethernet switch buffers [43] and TCP performance degradation. Their mechanism focuses on desynchronizing retransmissions by adding randomness to the TCP retransmission timer. Both of these approaches essentially modify the TCP protocol to adapt to the new environments; whereas for vFlood, we do not change TCP's behavior but merely re-architect it across the VM and driver domain to improve TCP throughput.

8. CONCLUSION

The main motivation of this paper stems from our investigations that reveal the negative impact of VM consolidation on transport protocols such as TCP. In virtualized cloud environments, TCP packets may experience significantly high RTTs despite sub-millisecond network latency, because of the VM CPU scheduling latency that is in the orders of tens of milliseconds. For many TCP connections, especially the small flows, such dramatic increase in RTTs leads to slower connection progress and lower throughput. To mitigate this impact, we have presented a solution called vFlood that effectively masks the VM CPU scheduling-induced latencies by offloading congestion control function from the sender VM to the driver domain and letting the sender VM opportunistically flood the driver domain with data to send. Our evaluation results indicate significant improvement in both TCP flow-level and application-level performance. Our experience with building a Xen-based prototype indicates that vFlood requires relatively small amount of code changes (about 1500 lines with 40% code reused from Xen/Linux), and its design is potentially portable to other VMMs.

9. ACKNOWLEDGEMENTS

We thank Ajay Gulati, and the anonymous reviewers for their insightful comments. This work was supported in part by the US NSF under grants 0546173, 0720665, 1054788 and 1017898. 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.

10. REFERENCES

[1] Alacritech corporation. http://www.alacritech.com.

[2] Apache Olio. http://http://incubator.apache.org/olio/.

[3] Chelsio communications. http://www.chelsio.com.

[4] CLOC. http://cloc.sourceforge.net.

[5] Faban. http://www.opensparc.net/sunsource/faban/www/index.html.

[6] The Iperf Benchmark. http://www.noc.ucf.edu/Tools/Iperf/.

[7] Linux Networking:TOE. http://www.linuxfoundation.org/collaborate/workgroups/networking/toe.

[8] Server Virtualization Landscape. http://events.1105govinfo.com/events/vcg-summit-2010/information/~/media/GIG/GIG%20Events/2010%20Enterprise%20Architecture/Presentations_0/VCG10_3%201_Oltsik%20Bowker.ashx.

[9] VMware Knowledge Base article. http://kb.vmware.com/kb/1006143.

[10] VMware Tools. http://kb.vmware.com/kb/340.

[11] Alizadeh, M., Greenberg, A., Maltz, D. A., Padhye, J., Patel, P., Prabhakar, B., Sengupta, S., and Sridharan, M. Data center TCP (DCTCP). In *ACM SIGCOMM* (2010).

[12] Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., Lee, G., Patterson, D. A., Rabkin, A., Stoica, I., and Zaharia, M. Above the clouds: A Berkeley view of cloud computing. Tech. Rep. UCB/EECS-2009-28, UC Berkeley, 2009.

[13] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. Xen and the art of virtualization. In *ACM SOSP* (2003).

[14] Benson, T., Anand, A., Akella, A., and Zhang, M. Understanding data center traffic characteristics. In *First ACM Workshop on Research on Enterprise Networking (WREN '09)* (2009).

[15] Brakmo, L. S., and Peterson, L. L. TCP Vegas: end to end congestion avoidance on a global Internet. *IEEE Journal on Selected Areas in Communications 13*, 8 (1995).

[16] Burtsev, A., Srinivasan, K., Radhakrishnan, P., Bairavasundaram, L. N., Voruganti, K., and Goodson, G. R. Fido: Fast inter-virtual-machine communication for enterprise appliances. In *USENIX ATC* (2009).

[17] Choudhury, A. K., and Hahne, E. L. Dynamic queue length thresholds for shared-memory packet switches. *IEEE/ACM Transaction on Networking 6* (1998).

[18] Dean, J., and Ghemawat, S. MapReduce: Simplified Data Processing on Large Clusters. In *USENIX OSDI* (2004).

[19] Govindan, S., Nath, A. R., Das, A., Urgaonkar, B., and Sivasubramaniam, A. Xen and Co.: communication-aware CPU scheduling for consolidated Xen-based hosting platforms. In *ACM VEE* (2007).

[20] Gulati, A., Merchant, A., and Varman, P. mClock: Handling throughput variability for hypervisor IO scheduling. In *USENIX OSDI'10* (2010).

[21] Guo, D., Liao, G., and Bhuyan, L. Performance characterization and cache-aware core scheduling in a virtualized multi-core server under 10GbE. In *IEEE IISWC* (2009).

[22] Gupta, D., Cherkasova, L., Gardner, R., and Vahdat, A. Enforcing performance isolation across virtual machines in Xen. In *ACM/USENIX Middleware* (2006).

[23] Gupta, D., Lee, S., Vrable, M., Savage, S., Snoeren, A. C., Varghese, G., Voelker, G. M., and Vahdat, A. Difference engine: Harnessing memory redundancy in virtual machines. In *USENIX OSDI* (2008).

[24] Ha, S., Rhee, I., and Xu, L. CUBIC: A new TCP-friendly high-speed TCP variant. *ACM SIGOPS Operating System Review 42*, 5 (2008).

[25] Huang, W., Koop, M. J., Gao, Q., and Panda, D. K. Virtual machine aware communication libraries for high performance computing. In *ACM/IEEE SC* (2007).

[26] Jin, C., Wei, D., and Low, S. FAST TCP: Motivation, Architecture, Algorithms, Performance. In *IEEE INFOCOM* (2004).

[27] Kandula, S., Sengupta, S., Greenberg, A., Patel, P., and Chaiken, R. The nature of data center traffic: measurements & analysis. In *ACM/USENIX IMC '09* (2009).

[28] Kangarlou, A., Gamage, S., Kompella, R. R., and Xu, D. vSnoop: Improving TCP throughput in virtualized environments via acknowledgement offload. In *ACM/IEEE SC* (2010).

[29] Kesavan, M., Gavrilovska, A., and Schwan, K. Differential Virtual Time (DVT): Rethinking I/O service differentiation for virtual machines. In *ACM SOCC* (2010).

[30] Kesavan, M., Gavrilovska, A., and Schwan, K. On disk scheduling in virtual machines. In *Second Workshop on I/O Virtualization (WIOV '10)* (2010).

[31] Kim, K., Kim, C., Jung, S.-I., Shin, H.-S., and Kim, J.-S. Inter-domain socket communications supporting high performance and full binary compatibility on Xen. In *ACM VEE* (2008).

[32] Menon, A., Cox, A. L., and Zwaenepoel, W. Optimizing network virtualization in Xen. In *USENIX ATC* (2006).

[33] Menon, A., Santos, J. R., Turner, Y., Janakiraman, G. J., and Zwaenepoel, W. Diagnosing performance overheads in the Xen virtual machine environment. In *ACM VEE* (2005).

[34] Menon, A., Schubert, S., and Zwaenepoel, W. TwinDrivers: semi-automatic derivation of fast and safe hypervisor network drivers from guest OS drivers. In *ACM ASPLOS* (2009).

[35] Menon, A., and Zwaenepoel, W. Optimizing TCP receive performance. In *USENIX ATC* (2008).

[36] Milos, G., Murray, D. G., Hand, S., and Fetterman, M. A. Satori: Enlightened page sharing. In *USENIX ATC* (2009).

[37] Mogul, J. C. TCP offload is a dumb idea whose time has come. In *USENIX HOTOS IX* (2003).

[38] Nurmi, D., Wolski, R., Grzegorczyk, C., Obertelli, G., Soman, S., Youseff, L., and Zagorodnov, D. The Eucalyptus open-source cloud-computing system. In *IEEE/ACM CCGrid* (2009).

[39] Ongaro, D., Cox, A. L., and Rixner, S. Scheduling I/O in virtual machine monitors. In *ACM VEE* (2008).

[40] Regnier, G., Makineni, S., Illikkal, R., Iyer, R., Minturn, D., Huggahalli, R., Newell, D., Cline, L., and Foong, A. TCP onloading for data center servers. *IEEE Computer 37* (2004).

[41] Shalev, L., Satran, J., Borovik, E., and Ben-Yehuda, M. IsoStack: Highly efficient network processing on dedicated cores. In *USENIX ATC* (2010).

[42] Sobel, W., Subramanyam, S., Sucharitakul, A., Nguyen, J., Wong, H., Klepchukov, A., Patil, S., Fox, O., and Patterson, D. Cloudstone: Multi-platform, multi-language benchmark and measurement tools for Web 2.0. In *First Workshop on Cloud Computing (CCA)* (2008).

[43] Vasudevan, V., Phanishayee, A., Shah, H., Krevat, E., Andersen, D. G., Ganger, G. R., Gibson, G. A., and Mueller, B. Safe and effective fine-grained TCP retransmissions for datacenter communication. In *ACM SIGCOMM* (2009).

[44] Wang, J., Wright, K.-L., and Gopalan, K. XenLoop: A transparent high performance inter-vm network loopback. In *ACM HPDC* (2008).

[45] Wood, T., Shenoy, P., Venkataramani, A., and Yousif, M. Black-box and gray-box strategies for virtual machine migration. In *USENIX NSDI* (2007).

[46] Zhang, X., McIntosh, S., Rohatgi, P., and Griffin, J. L. XenSocket: A high-throughput interdomain transport for virtual machines. In *ACM/IFIP/USENIX Middleware* (2007).