Virtual TCP Offload: Optimizing Ethernet Overlay Performance on Advanced Interconnects

Zheng Cui[†] Patrick G. Bridges[†] John R. Lange[*] Peter A. Dinda[‡]

[†]Department of CS
University of New Mexico
Albuquerque, NM 87131, USA
{cuizheng,bridges}@cs.unm.edu

[*]Department of CS
University of Pittsburgh
Pittsburgh, PA 15260 USA
jacklange@cs.pitt.edu

[‡]Department of EECS
Northwestern University
Evanston, IL 60208 USA
pdinda@northwestern.edu

ABSTRACT

Ethernet overlay networks are a powerful tool for virtualizing networked applications. Their performance suffers on advanced interconnects such as Infiniband, however, because of differences between the semantics of Ethernet and the underlying network. In this paper, we demonstrate that providing a virtual TCP offload Ethernet device to the guest operating system dramatically improves overlay network performance on advanced interconnects like Infiniband. The virtual offload device enables the overlay system to leverage the semantics and performance characteristics of the underlying network to maximize overlay performance. Our evaluation shows that this approach allows applications to achieve near-native microbenchmark bandwidth and dramatically improved application performance compared to alternative overlay approaches when running on an Ethernet virtual overlay on a QDR Infiniband fabric.

Categories and Subject Descriptors

D.4.4 [**Software**]: OPERATING SYSTEMS

Keywords

Overlay Networks; Virtualization; HPC; InfiniBand

1. INTRODUCTION

Data centers and scientific clouds require clusters and supercomputers interconnected with advanced networks, such as InfiniBand, SeaStar, and Gemini interconnects. Such hardware resources are increasingly being integrated with virtualization as a means of deploying and managing large-scale computing systems with the "Infrastructure as a Service" (IaaS) cloud computing model. The combination of virtual machines and virtual overlay networking provides a powerful model to realize virtual distributed and parallel computing with strong isolation, portability, and recoverability properties.

This project is made possible by support from the United States National Science Foundation (NSF) via grants CNS-0707365 and CNS-0709168, and by the Department of Energy (DOE) via grants DE-SC0005050 and DE-SC0005343.

A virtual *Ethernet overlay network* supports broad classes of applications and software stacks by presenting a uniform Ethernet communication environment. This simple, yet powerful abstraction is provided regardless of the underlying networking hardware, which may be quite different from Ethernet, and which may span multiple different data center and supercomputer networks. The abstraction is a form of software-defined networking (SDN), albeit the implementation is accomplished purely in end-system software via tunneling. This has the benefit that it does not require hardware support (e.g., OpenFlow), nor cooperation among data center/supercomputer networks, and can extend anywhere where IP networking is available.

Past work has shown that this model can provide near-native performance on high-speed Ethernet networks [6,17]. Providing the same Ethernet abstraction on high-end data center, cluster, and supercomputer networks provides a number of additional advantages, such as reducing the effort porting applications designed for Ethernet to other heterogeneous networks.

Current virtual overlay networks, however, are unable to deliver near-native network performance on advanced interconnects such as Infiniband due to the *semantic gap* between the Ethernet overlay network and underlying physical network. Such a semantic gap [8] is inevitable in virtual overlays whenever the semantics of the underlying physical network are different from that of the overlay network. In the case of an Ethernet overlay on top of Infiniband, for example, performance problems from the semantic gap arise from differences in overlay and network MTUs, or unnecessary protocol overheads from providing reliability semantics in both the guest protocol stack and in the host network adapter. While substantial work has been done bridging the semantic gap between the VM and the VMM in general [12–15,21,23], comparatively little work has been done on bridging this gap for virtual overlay networks.

This paper proposes enhancing the Ethernet virtual overlay network with virtual TCP offload support to bridge the semantic gap between the guest application, the overlay network, and the underlying interconnect. TCP offload capabilities in the virtual Ethernet device allow the guest to better communicate its desired network semantics to the overlay, improving its ability to meet guest network demands. We demonstrate the viability of this approach by enhancing the VNET/P virtual network overlay with offload device capabilities and running it on top of a high-performance Infiniband network fabric. Our results show that adding offload capabilities to the Ethernet overlay network substantially increases network performance on sophisticated data center networks while preserving the advantages of Ethernet overlay networks.

The remainder of this paper is organized as follows. In Section 2, we provide basic background on virtual overlay networks in general, the VNET overlay we use in this paper, and the Infiniband interconnect on which we evaluate our proposed overlay enhancement. Section 3 then follows with a discussion of the challenges the semantic gap presents when running an Ethernet overlay on top of an advanced interconnect such as Infiniband. Section 4 describes our proposed mechanism for spanning this semantic gap, Virtual TCP Offload Engines (VTOEs), and Section 5 provides details of the implementation of this mechanism for Infiniband networks with Linux guests. Sections 6 and 7 then provide microbenchmark and application benchmarks demonstrating the advantages of this approach. Finally, Section 8 concludes and describes directions for future work.

2. BACKGROUND AND RELATED WORK

In this section, we provide a brief introduction to virtual overlay networks. The implementation and evaluation of our approach to optimizing overlay performance for advanced interconnects was conducted using the VNET/P high-performance virtual overlay on an InfiniBand network. Because of this we also provide key architectural details of VNET/P and the Infiniband network architecture.

2.1 Virtual Overlay Networks

Current adaptive cloud computing systems use software-based overlay networks to carry inter-VM traffic. For example, the user-level VNET/U system [18, 24, 25] combines a simple networking abstraction within VMs with location-independence, hardware-independence, and traffic control. Specifically, it exposes a layer 2 abstraction that lets the user treat his VMs as being on a simple LAN, while allowing the VMs to be migrated seamlessly across resources by routing their traffic through the overlay. By controlling the overlay, the cloud provider or adaptation agent can control the bandwidth and the paths between VMs over which traffic flows. Such systems [22, 24] and others that expose different abstractions to the VMs [26] have been under continuous research and development for several years.

2.2 VNET/P Implementation

VNET/P [17] is an in-VMM, overlay-based layer-2 virtual networking system for the Palacios VMM [19]. VNET/P consists of a virtual NIC in each guest OS, an extension to the VMM (the VNET/P Core) that handles packet routing and interfacing to virtual NICs, and a Linux kernel module (the VNET/P Bridge) for interacting with the host's network interfaces and remote systems. For high performance applications, as in this paper, the virtual NIC conforms to the virtio interface, but several virtual NICs with hardware interfaces are also available in Palacios.

In operation, the virtual NIC moves Ethernet packets between the application VM and Palacios, and includes receive and transmit rings. Interrupts are injected into the guest via a virtual IOAPIC/APIC interrupt controller structure. Routing and packet forwarding occur in the VNET/P Core. Routing is based on MAC addresses with a hash-based cache system that allows for constant time lookups in the common case. A packet routed by the VNET/P Core to a guest is handed to a virtual NIC, while a packet routed to an external network or machine is routed to the VNET/P bridge. The VNET/P bridge, which is embedded in the host kernel, encapsulates guest Ethernet packets into UDP datagrams and sends them out through host Ethernet devices.

Our experiments in this paper include portions of the VNET/P+ (note the "+") optimizations that improve the performance of VNET/P for high speed networks [6]. They allow VNET/P to achieve near-native performance for a wide range of microbenchmarks and MPI application benchmarks on high-speed Ethernet networks, including 10 Gigabit/second Ethernet networks. In particular, we utilize zero-copy data overlay data movement techniques, but have not yet merged in support for optimistic interrupt techniques.

2.3 InfiniBand

InfiniBand [1] is a standard switched fabric that supports high bandwidth (up to 120Gb/s) and low latency. Processor nodes connect to the fabric through Host Channel Adapters (HCA). High-performance connections on these devices, particularly Reliable Connected (RC) modes, provide zero-copy RDMA data transfer which bypasses OS involvement in data movement. Infiniband NICs also provide other network interfaces, for example the Unreliable Datagram (UD) mode, whose semantics are similar to that of traditional Ethernet and are generally used for administrative tasks.

The InfiniBand specification defines an HCA interface called Verbs. Upper layer protocols are implemented on top of Verbs. This interface is asynchronous: a consumer posts Work Requests (Send, Receive, RDMA Write, RDMA Read and Atomic operations) to the HCA. The HCA optionally signals their completion and can schedule a completion notification (through event queues or interrupt).

In addition, Infiniband implementations generally provide IP-over-Infiniband support (IPoIB) [16]. The IPoIB functionality of the device driver allows the TCP/IP stack of the host to use the NIC to transport IP packets. Network overlay systems such as VNET/P can use IPoIB functionality to get basic overlay functionality on Infiniband networks. For reasons discussed later in Section 3, this approach has significant performance problems.

2.4 InfiniBand Virtualization

Currently two approaches are very popular in virtualizing InfiniBand with high performance: VMM-bypass [20] and Passthrough [2, 6, 19]. VMM-bypass I/O extends the idea of OS-bypass originated from user-level communication, and allows time-critical I/O operations to be carried out directly in guest VMs without involvement of the VMM and/or a privileged VM. However, the user-space application in the guest has the direct access to the physical IB device resources. In the Passthrough model, the VM has direct access to the InfiniBand devices via VMM's passthrough mechanism.

Both of these approaches can significantly improve I/O and communication performance for VMs, in some cases even without sacrificing safety or isolation. However, they lock the VM to the specific InfiniBand infrastructure, losing the portability of virtual networks. This makes checkpointing and migration more difficult because when a VM is restored from a previous checkpoint or migrated to another node, the corresponding state information on the device needs to be restored also, which requires a similar or identical device.

3. CHALLENGES

Virtual Ethernet overlay mechanisms like VNET effectively support high-performance applications on physical networks with semantics similar to Ethernet. On more advanced interconnects, however, the semantic gap between overlay features and physical interconnect features presents difficult performance challenges. In particular, guest OSes see only a relatively simple Ethernet interface and so do not provide the overlay with higher-level semantic information about desired network semantics. This lack of knowledge about the guest-level communications can lead to performance degradation.

When deploying a virtual Ethernet overlay on top of advanced interconnects, there are two straightforward approaches to addressing the semantic gap between the virtual overlay and underlying networks, like Infiniband, with more advanced features: (1) Using minimal interconnect features to minimize the semantic gap, or (2) using more advanced features without guest knowledge. Each has significant performance problems as we describe below.

3.1 Using Minimal Features

The first alternative is to use minimal interconnect features to transport guests' traffic. The interconnect may be able to provide reliability to applications on demand, while the overlay delivers packets without any guarantees.

As an example, overlays could use the InfiniBand Unreliable Datagram (UD) transport service to minimize the gap between overlay semantics and physical network semantics. Unfortunately, datagram-based transport services in most advanced interconnect implementations are limited to Maximum Transmission Unit (MTU) sizes which are usually less than 4KB. The Infiniband MLX4 NICs used in this paper, for example, impose a 2KB MTU. Small MTUs dramatically reduce network throughput on high-speed networks by increasing the required number of network headers, routing decisions in the routers, protocol processing and device interrupts [5].

In addition, minimal-feature interconnect modes generally do not bypass the OS and require significant interrupt processing. Such interrupts are even more expensive in virtualized operating systems than in non-virtualized hosts [6]. For example, virtual interrupt emulation introduces overhead for virtual device register handling (such as APIC, IOAPIC, and NIC), guest context switches, and the VM/guest stack switch. Moreover, the virtual machine monitor (VMM) has to maintain the emulation state for each trap, which significantly increases virtual network latency and decreases throughput by more than 30%.

3.2 Translating to Advanced Features

Alternatively, the overlay system can use advanced interconnect features that provide more complex semantics (e.g., connected reliable streams) while hiding these features from the guest. However, guests cannot assume these semantics will be provided since the overlay exports a simple Ethernet interface to VMs. As a result, guests must provide such semantics themselves when they are necessary. This can introduce duplicated guest/overlay protocol processing overheads.

For example, using a Linux guest and Infiniband RC connections in the overlay causes the guest to unnecessarily send TCP connection establishment requests and acknowledgments over the reliable Infiniband connection. The guest also unnecessarily checksums the incoming packets and performs congestion and flow control activities. These increase packet latency and guest CPU processing requirements, reducing application performance.

4. VIRTUAL TCP OFFLOAD

To address these semantic gap issues, we supplement the virtual Ethernet NIC exported to the guest by the overlay with a Virtual TCP Offload Engine (VTOE). In this section, we give an overview of our the general VTOE approach, show the architecture of VNET/P as enhanced with VTOE capabilities, and describe the overall architecture of the VNET VTOE NIC.

4.1 Overview

TCP Offload Engine (TOE) Ethernet devices offload the processing of the entire kernel TCP/IP stack to the network controller. They are primarily used with high-speed network interfaces such as 10 Gbps Ethernet, where the processing overheads of the network stack are significant [4]. Most modern operating systems support TCP offload engines, though Linux has no generic support for TCP offload.

By exporting a virtual TCP offload engine to the guest, the overlay enables guests that support TCP offload to designate both reliable and unreliable traffic at the Ethernet level. This reduces the semantic gap between the guest and overlay. For connections that span only interconnects that guarantee reliable transport, this results in lower virtualization overhead and achieves better network performance, as shown in Section 7. For connections that cross networks that do not guarantee reliable transport, the overlay itself provides reliability using TCP; existing overlays such as VNET already support overlay-level TCP tunneling. By exposing VTOE to the guest, the overlay promises to either run TCP itself or to use a network that obviates the need for it.

4.2 Virtual TCP Offload in VNET/P

Figure 1 shows the overall architecture of VNET/P supplemented with a Virtual TCP Offload Engine, which we denote VNET+VTOE. In this system, guests run in application VMs. The VMM provides a *virtual (Ethernet) NIC* with an offload engine to each guest. Basic Ethernet virtual NIC functionality is used to transport non-TCP Ethernet packets between the application VM and the overlay implementation inside the VMM, while the VTOE carries TCP traffic.

Inside the virtual machine monitor, the VNET Core and Host Connection Agents (*VNET_Core_CA* and *VNET_Host_CA*) are respectively responsible for interacting with the guest VNIC and the host physical NIC. Specifically, the VNET_Core_CA supplies the guest with a VNET Socket ID for each connection it creates to use to make requests to the overlay. The VNET_Host_CA creates and manages shadow connections over the underlying high performance

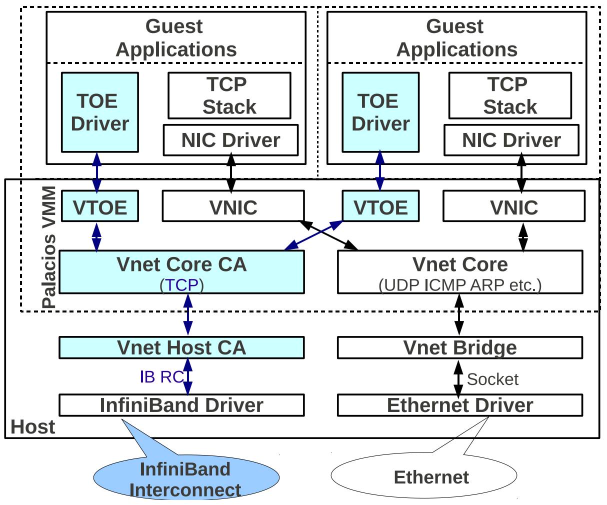


Figure 1: VNET+VTOE architecture with Linux VM and Palacios VMM.

fabric, which the VNET_Core_CA references using a shadow connection ID (shadow CID).

More specifically, VNET_Core_CA:

1. Maps between guest VNET Socket IDs (SIDs) and host shadow Connection IDs (CIDs),
2. Provides receive buffers to the VNET_Host_CA based on buffers supplied by the guest, and
3. Translates events and interrupts from the underlying physical device to VTOE events and interrupts as necessary.

For example, when the guest requests the creation of a new offload TCP connection through the VTOE NIC, the VNET_Core_CA allocates a unique Socket ID for the guest and returns this to the guest using the VTOE NIC. When a connection is later established between two VNET sockets, the VNET_Host_CAs on each hosts create a unique shadow Connection ID. Each guest uses its local Socket ID when enqueuing buffers to the overlay, and VNET_Core_CAs maps between the VNET Socket ID and the shadow Connection ID when interacting with the VNET_Host_CA.

The VNET_Core_CA memory allocator manages direct memory access (DMA) from buffers posted to the guest SID to the underlying shadow CID, and the VNET_Core_CA event dispatcher handles virtual interrupt and asynchronous event delivery to virtual offload engines based on the physical interrupts raised by local devices and events signaled by the underlying physical device. These components are also responsible for handling the memory mapping and interrupt processing for zero-copy cut-through forwarding.

4.3 VTOE NIC Architecture

The VTOE NIC provides a simple offload interface between the guest and the VNET_Core_CA. It supports two main classes of operations between the guest and the overlay in the virtual machine monitor. Specifically, I/O ports and an event queue are used for managing connection creation and state changes, while interrupts and send and receive work queues are used to manage data movement between the guest, the overlay, and the network.

The guest and overlay manage connection creation and state management using I/O ports and a memory-mapped event queue. Guests request the creation of new connections by programming VTOE I/O ports, and the overlay communicates connection state information back to the guest using the event queue. All event queue entries are tagged with the corresponding socket ID, and event queue includes events for all TCP-relevant state changes, including CONNECT_REQUEST, CONNECT_ESTABLISHED, DISCONNECTED, ADDRESS_ERROR, UNREACHABLE, and CONNECT_REJECTED,

Likewise, the guest uses send and receive work queues (SWQ and RWQ) to enqueue data buffers to the overlay for transmission or receiving incoming data. Each queue entry is tagged with the relevant SID, enabling the VNET_Host_CA to map to the relevant underlying shadow connection. Once data has arrived into a guest buffer, the overlay raises a virtual interrupt into the guest at the appropriate time, either eagerly or using more complex optimization such as optimistic interrupts [6].

5. IMPLEMENTATION

Our initial implementation of VTOE support in VNET/P has focused on Infiniband networks, but VTOE could also be used to support overlay functionality on other high-performance network fabrics such as Cray SeaStar or Gemini systems. In addition, we have implemented VTOE NIC support for Linux guests; because general Linux support for TCP offload is somewhat lacking, this required special measures on Linux guests. We detail the specific work done to support VNET+VTOE on Infiniband with Linux guests in the remainder of this section.

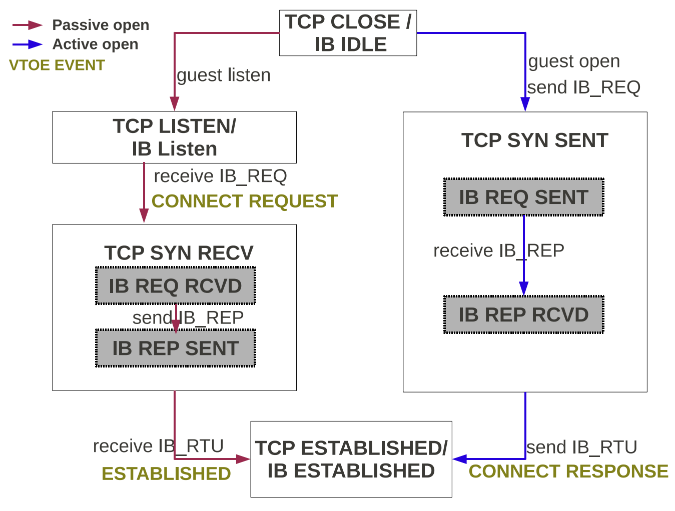
5.1 Infiniband Support

VNET+VTOE Infiniband support has two main elements: connection management and data movement. Connections are established in a multistep handshake process before entering the data transfer phase. After data transmission is completed, the connection termination closes established virtual circuits and releases all allocated resources. In the remainder of this subsection, we describe the mapping between TCP and Infiniband RC connection states, and detail management of data movement for Infiniband and VNET+VTOE.

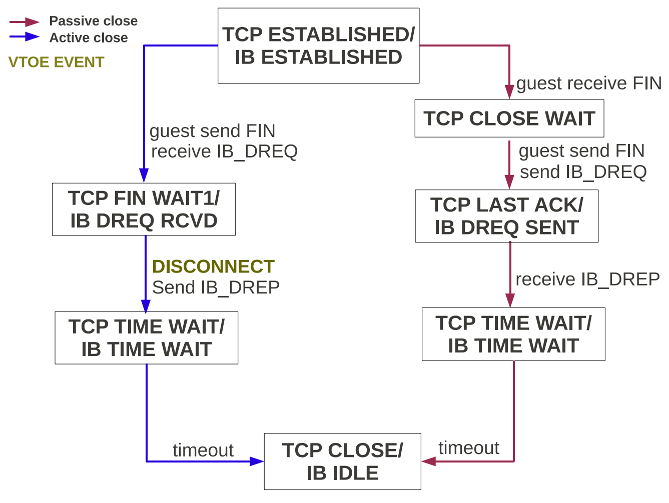
5.1.1 Connection Management

For connection management, the VNET_Core_CA must manage three different sets of states: the guest TCP state, the Infiniband connection state as per the IB Connection Management standard, and the underlying Infiniband queue pair state. To do this, it makes requests to the VNET_Host_CA in response to guest connection changes, maps event notifications from the VNET_Host_CA to TCP event notifications to the guest, and communicates with the local Infiniband connection manager. We use Active/Passive (also referred as client/server) mode to establish a connection. In the client/server model, the shadow server side listens for connection requests with a service ID; the client shadow side initiates a connection request with a matching service ID.

Connection Establishment: Figure 2(a) shows the TCP state machine of the VNET socket and the IB state of the



(a) Connection establishment



(b) Connection termination

Figure 2: State machines for both frontend VNET sockets and shadow CIDs, during connection establishment and termination.

shadow connection during connection establishment. The right half of figure Figure 2(a) corresponds to an active open by the guest (i.e. a `connect()` system call), while the left half of the figure corresponds to a passive open by the guest (i.e. a **listen()** system call). Note the close, but not exact, correspondence between the TCP and IB connection state machines.

When the guest performs an active open on the VNET socket and changes its socket state to TCP_SYN_SENT, the overlay sends of an IB_REQ (request) packet which includes the Socket ID and number of available receive buffers. It also changes the IB connection state changes to IB_REQ_SENT and changes the underlying queue pair (not shown) to the READY_TO_RECEIVE state.

Note that during this process, overlay routing tables and existing IB infrastructure are used to handle address resolution mapping guest Ethernet addresses to underlying Infiniband addresses in the overlay. As part of this process, the overlay delivers ROUTE_RESOLVED events to the guest as necessary.

When the remote side responds with an IB_REP (reply) message, the VNET Core Connection Agent changes the shadow connection state to IB_REP_RECEIVED, sets the queue pair to READY_TO_SEND state, sends an IB_RTU (ready-to-use) message, and transitions to IB_ESTABLISHED state. It then notifies the guest of the response to its connection request. The guest transitions the guest socket to the TCP_ESTABLISHED state in response.

The process is similar on passive opens. The overlay sends CONNECT_REQUEST events to the guest upon receipt of an IB_REQ packet, initializes the underlying queue pair (not shown), and sends IB_REP message. Similarly, it sets the queue pair to READY_TO_SEND and delivers the guest an ESTABLISHED event on the receipt of an IB_RTU message.

Connection Termination: Figure 2(b) illustrates the procedure for connection teardown. When a guest wishes to stop its half of the connection, it send a TCP FIN packet to its peer and changes state from TCP_ESTABLISHED to TCP_FIN_WAIT1.

On receiving the FIN message, the peer changes state from TCP_ESTABLISHED to TCP_CLOSE_WAIT. It continues sending any outstanding data, however, before notifying the overlay to close its half of the connection. After sending out all the outstanding packets, the peer sends a FIN packet to its peer, sends a disconnect command to the overlay, and transitions to state TCP_LAST_ACK. In the overlay, the shadow CID sends a disconnection request message to the peer host and changes IB connection state to IB_DREQ_SENT. Upon receiving IB_DREP from the active-closing host, the shadow CID changes queue pair state to ERROR, and transitions to state IB_TIME_WAIT.

When the active-closing shadow CID receives the disconnection request from the passive-closing host, it transits to state DREQ_RCVD. It delivers all the incoming packets to the guest VTOE buffers, and raises a DISCONNECT event to the guest. It then changes queue pair state to ERROR, sends an IB disconnection reply message to the peer host, and changes IB connection state to IB_TIME_WAIT. When the guest either gets the FIN packet from the peer, or the DISCONNECT event from the overlay, it processes all the incoming data and transitions to state TCP_TIME_WAIT. Finally, timeouts are used to transition from time wait to idle connection states.

5.1.2 *Data Transfer*

Data transfers leverage two optimization techniques.

Transmission with Zero Overlay Copies: The guest OS in the VM includes the device driver for the virtual NIC and the VTOE. The socket initiates packet transmission by posting a SEND request with Socket ID and data source to the send work queue in the VTOE NIC. In the overlay, the packet dispatcher sends the packet of type TCP_OFFLOAD to the VNET Core Connection Agent. The VNET_Core_CA maps the packet Socket ID to the appropriate shadow

CID and posts the packet to the appropriate Infiniband RC queue pair.

While the packet is handed off multiple times, there is no copy from the guest's socket buffer to the host's NIC. We adopt the zero-copy data forwarding technique to avoid any data copies in the overlay. Note, however, that the guest may include a copy from the application's data buffers to the VNET socket's private buffer.

Reception with Zero Overlay Copies: As in the transmit case, guests use the receive work queue in the VTOE NIC to post receive buffers to different connections. The VNET_Core_CA and VNET_Host_CA work together to post these buffers to the queue pair associated with the shadow connection. As in the send case, the receive datapath to the guest OS does not require any copies, using the zero-copy data forwarding technique. Also note that the receive path does not need to route packets in the overlay, since each shadow CID is associated with a unique guest socket ID.

5.2 Interfacing With Linux Guests

Interfacing VNET+VTOE with Linux is somewhat complicated due to the lack of general TCP offload support in Linux. We worked around this problem similarly to how Infiniband and other Linux TCP offload implementations do. In particular, we use the Infiniband SDP [3] approach to dynamically change the application address family into AF_INET_VNET using a preload library. This address family then redirects to new offload drivers in the guest. This code has two elements, the *VNET socket provider* and the *VTOE socket module*.

The VNET socket provider is user-mode shared library code that provides socket direct extensions to the TCP/IP stack and determines which connections to redirect, based on protocol type, to the AF_INET_VNET address family. These socket direct extensions are completely transparent to the higher-layer protocols and applications that run on top of them. Applications interact in the same way with a VNET+VTOE stack as they would with a standard TCP/IP stack.

For the connection establishment calls, the provider makes a routing and policy decision and decides whether a TCP or VNET socket should be created. If a TCP socket is required, all calls on the socket are redirected to the Linux socket chain. If a VNET socket is required, the calls are redirected to the kernel VNET socket module.

The VNET socket module handles the socket operations redirected from the TCP socket by the VNET socket provider, updating kernel socket state and interfacing the VTOE device as necessary. and responding asynchronous events.

Although in this work the user is responsible for inserting the kernel module into the guest, and for assuring that the application uses the preload library, this not strictly necessary. In related work, we have shown how both of these steps can be done without user or guest cooperation through VMM-based code injection [9].

6. MICROBENCHMARKS

We first studied the effects of VTOE on a set of simple TCP and MPI throughput and latency microbenchmarks. Application benchmarks are described in Section 7.

6.1 Testbed

Our testbed, which is used both here and in the next section, consists of 6 physical machines each with dual quad core 2.3 GHz 2376 AMD Opteron "Shanghai" processors (8 cores total), 32 GB RAM, and a Mellanox MT26428 InfiniBand NIC in a PCI-e slot. The Infiniband NICs were connected via a Mellanox MTS 3600 36-port 20/40 Gbps InfiniBand switch.

We compared the performance of four different configurations, all mapped to underlying Infiniband RC connections:

- **Native+SDP/Uverbs:** Infiniband Socket Direct Protocol to offload TCP connections or MPI directly using Infiniband user-level verbs.
- **Native+IPoIB:** In-kernel TCP over the Infiniband in-kernel IP-over-IB implementation.
- **VNET+VTOE:** VNET Ethernet overlay with virtual TCP offload support.
- **VNET+IPoIB:** In-kernel TCP over VNET on top of the host IP-over-IB implementation.

For VNET+VTOE and VNET+IPoIB measurements, we ran a simple Linux 2.6.32 host with a minimal BusyBox configuration, and the Palacios VMM. The guest used was a Linux 2.6.30 kernel also with a minimal BusyBox running on a virtual machine with a single virtio network interface, 4 cores, and 2.5 GB of memory. In the VNET+VTOE configuration, the guest is provided with a single virtual TOE. Unless otherwise specified, the virtio NIC provided to the guest was configured to use 9000 byte MTUs. For native measurement, we ran a Linux 2.6.30 kernel also with a minimal BusyBox. For Native+IPoIB and VNET+IPoIB configurations, MTUs are set to 65520.

Performance measurements were made between identically configured machines. To assure accurate time measurements in the virtualized case, each guest was configured to use the CPU's cycle counter, and Palacios was configured to allow the guest direct access to the underlying hardware cycle counter.

The CPU utilization is reported by TTCP by dividing the total of user mode time + guest OS kernel time by real used wall-clock time, so it includes both the guest OS and VMM CPU costs, and is not averaged for single-thread TTCP.

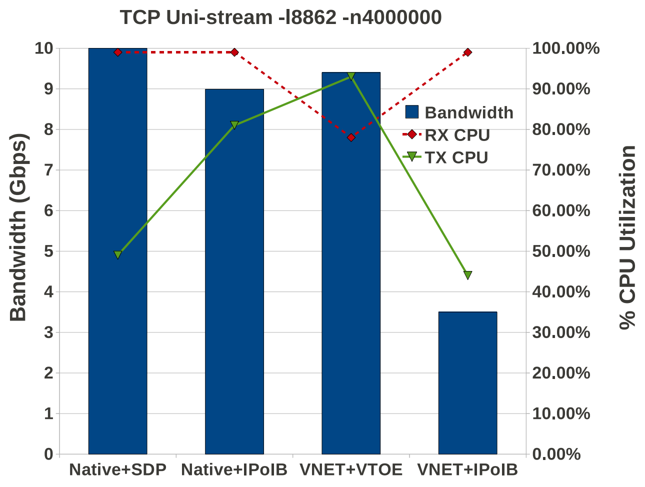
6.2 Microbenchmarks

We used simple two-node TCP and MPI benchmarks to provide an initial characterization of the impact of our proposed VTOE infrastructure. TCP throughput was measured using *ttcp-1.10*. For simple MPI tests, we used the Intel MPI Benchmark Suite (IMB 3.2.2) [11] running on OpenMPI 1.3 [7], focusing on the point-to-point messaging performance. For each test case, we ran 10 times and report the average as the result.

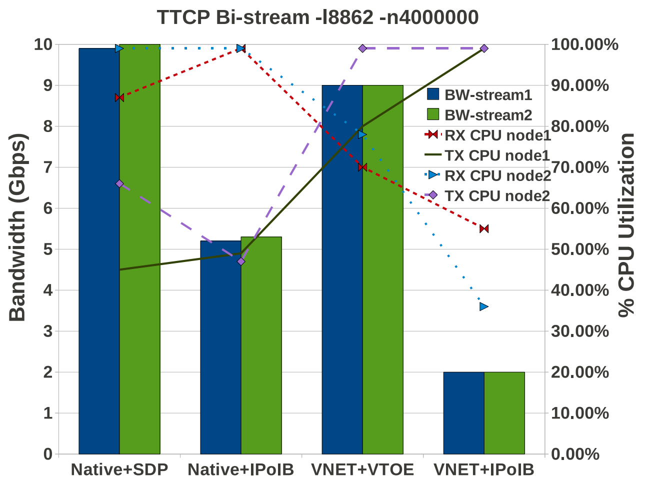
6.2.1 TCP Uni-stream Bandwidth

Figure 3(a) shows uni-stream bandwidth performance for VNET+VTOE running over a single connection along with CPU utilization.

VNET+VTOE achieves near-native micro-benchmark performance of 9.4 Gbps, compared to the 10 Gbps in the Native+SDP case. This is higher bandwidth than Native+-IPoIB performance, and Virtual TCP offload offers nearly 2.7 times the performance of VNET using IP-over-IB.



(a) TCP Uni-stream Bandwidth



(b) TCP Bi-stream Bandwidth

Figure 3: End-to-end TCP throughput and CPU utilization of Native+SDP, Native+IPoIB, VNET+VTOE, and VNET+IPoIB on InfiniBand Interconnect. VNET+VTOE performs more than 2.5 times better then VNET+IPoIB on the InfiniBand Interconnect

In terms of CPU usage, Native+IPoIB and Native+SDP have the same receive-side CPU usage, while Native+IPoIB has more transmit-side CPU utilization than Native+SDP. In the virtualization cases, VNET+VTOE has 76% receive-side usage compared to 99% for VNET+IPoIB. On the transmit side, VNET+IPoIB has less CPU usage than VNET+-VTOE.

Analysis: First, in the native cases, the more transmit-side CPU utilization in Native+IPoIB, mainly comes from the interrupt processing triggered by large amount of ACKs generated in the receive-side TCP stack.

Second, in the virtualization cases, on the receive side, the higher CPU usage in VNET+IPoIB comes from two-level overheads: 1) On the first level, sender's TCP processing, and overlay's data copies, encapsulation, de-encapsulation, and routing all slower down the packet delivering rate (virtual link speed), and thus more virtual interrupts are generated to the receive side, which have the receive side spend more time on virtual interrupt processing; and 2) on the second level, each incoming packet goes through the whole guest TCP stack, which makes the receive side busier.

On the transmit side, in VNET+IPoIB, each incoming packet has to go through the TCP stack, so the receive-side TCP is slower in generating ACKs to the sender; moreover, the virtual link is slower in delivering ACKs, and virtual interrupt handling takes time, thus the flow control window in the sender is exhausted faster, and therefore the sender slows down.

6.2.2 *TCP Bi-Stream Bandwidth*

In addition to unidirectional TCP performance, we also examined bi-stream bandwidth performance to measure the duplex capability of VNET+VTOE. In this test, we use two machines and two threads on each machine. Each thread connects to its partner on the other machine, thus two connections are established between the machines. On each connection, the basic TTCP bandwidth test is performed. The throughput and CPU usage are shown in Figure 3(b).

Native+SDP shows good duplex performance, delivering 10 Gbps bandwidth for each stream. In contrast, Native+-IPoIB hits a bottleneck on bi-directional data transfer, with each stream dropping to half of the wire capacity. This is due to the TCP acknowledgment processing, which increases CPU interrupt processing overhead.

In the virtual overlay configurations, VNET+VTOE also fully utilizes the physical interconnect's full duplex features. Similar to the native case, VNET+IPoIB does not utilize the interconnect's full-duplex capabilities. This again mainly comes from the guest-level duplicated reliability processing and virtual interrupts triggered by ACKs from the TCP stack.

The CPU utilization is also presented for each test configuration. The benchmark reveals that Native+SDP and VNET+VTOE can not only achieve high aggregated bandwidth, they also show reduced overall CPU utilization. Specifically, Native+SDP reduces receive-side average CPU utilization compared with Native+IPoIB, and VNET+VTOE reduces the transmit-side average CPU usages compared to VNET+IPoIB.

6.2.3 *CPU Utilization*

There are two important observations regarding the measurements shown in Figure 3.

1. Receive-side CPU utilization in the virtualized configurations is lower than for the native configurations, while the opposite is true for the transmit-side.

2. In both native cases, receive-side CPU utilization higher than transmit-side CPU utilization, while the opposite is true for both virtualized configurations.

In the native cases, the real physical link is fast enough to keep the receive-side CPU busy with incoming packets, and the NIC speed is faster than the CPU, thus the application cannot consume data from the buffer as fast as it is filled. The receive-side flow control window is quickly exhausted and the sender has to slow down. In the native cases, the network performance is bound to the receive-side CPU utilization.

In contrast, in the virtualized cases, the virtual link provided by the overlay is slower, reducing load on the receiver. The receiver now buffers data slower than the application can consume it. The receive window is open, the sender delivers data as fast as possible, and thus the network performance is bound to the overlay virtual link data-transfer rate.

6.2.4 MPI

Figure 4 shows the IMB MPI point-to-point performance with VNET+VTOE. For small messages, VNET+VTOE has more than two times lower message delay than VNET+-IPoIB, but two times higher message delay than Native+-IPoIB. For medium-sized messages, VNET+VTOE approaches Native+IPoIB performance. For large messages, Native+-IPoIB achieves about 47% of Native+Uverbs throughput, while VNET+VTOE achieves 60% of Native+Uverbs performance. VNET+IPoIB delivers about 28% of Native+-Uverbs bandwidth.

7. APPLICATION BENCHMARKS

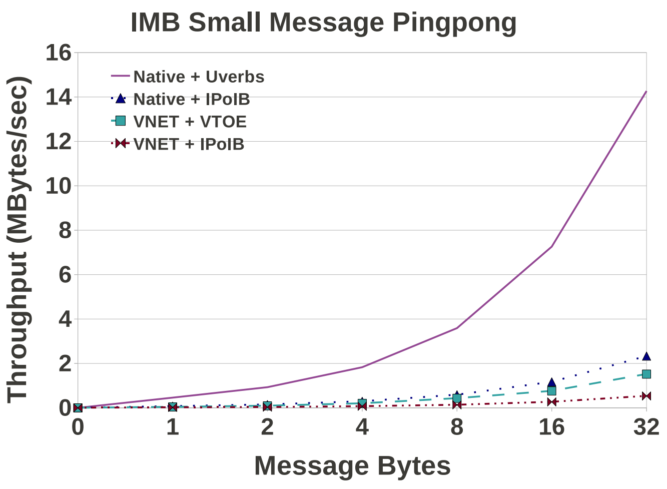
Beyond the microbenchmarks we described in the previous section, we also evaluated VNET+VTOE using the HPC Challenge benchmarks, with the goal of characterizing the performance impact of the VTOE optimization on communication-intensive applications.

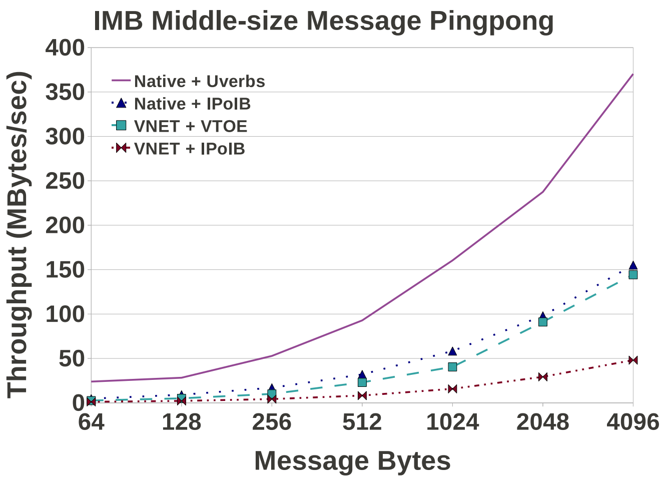
7.1 HPC Challenge Benchmarks

The HPC Challenge (HPCC) benchmarks [10] are a set of macro and application benchmarks for evaluating various aspects of the performance of high performance computing systems. We used the communication-oriented macrobenchmarks and application benchmarks to compare the performance of VNET+VTOE with Native+Verbs, Native+-IPoIB, and VNET+IPoIB. For these tests, each VM was configured with 4 virtual cores, 2.5 GB RAM, and a virtio NIC. For VNET+VTOE, each VM is also configured with a virtual TOE. For VNET testing, each host had one VM running on it. We ran tests with 2, 3, 4, 5, and 6 VMs with 4 HPCC processes started on each VM. Thus our performance results are based on HPCC with 8, 12, 16, 20, and 24 processes. In the native cases, no VMs are used and the processes ran directly on the host.

7.1.1 Latency-Bandwidth Benchmark

This benchmark consists of the ping-pong test and the ring-based tests, where the former measures the latency and bandwidth between all distinct pairs of processes. The ring based tests arrange the processes in a ring topology and then engage in collective communication among neighbors in the ring, measuring bandwidth and latency. The ring-based tests model the communication behavior of multi-dimensional domain-decomposition applications. Both naturally ordered rings and randomly ordered rings are evaluated. Communication is done with MPI non-blocking sends and receives, and MPI SendRecv. Here, the bandwidth per process is defined as total amount of message data divided





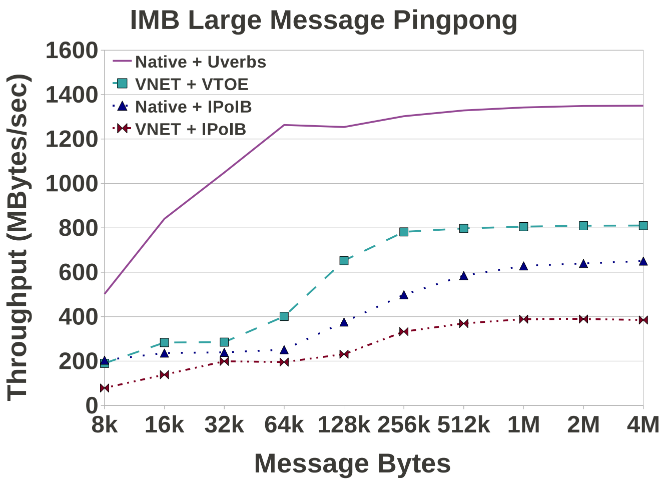


Figure 4: Intel MPI PingPong microbenchmark showing bidirectional throughput as a function of message size on InfiniBand Interconnect

by the number of processes and the maximum time needed in all processes.

Figure 5 shows the results of the HPCC Latency-Bandwidth benchmark for different numbers of test processes. Ping-Pong Latency and Ping-Pong Bandwidth results are consistent with the previous microbenchmarks: Native+IPoIB generally has 5–12 times higher latency than Native+Uverbs, and 40–60% bandwidth of Native+Uverbs. In VNET+-VTOE, bandwidths are within 60% of Native+Uverbs, and latencies are about 2 times that of Native+IPoIB. In VNET+-IPoIB, bandwidths are within 20–30% of Native+Uverbs, and latencies are about 4 times that of Native+IPoIB latencies. The results show that our VTOE can substantially enhance the performance of a software-based overlay virtual network like VNET/P on InfiniBand.

7.1.2 HPCC application benchmarks

We considered the three application benchmarks from the HPCC suite that exhibit the largest volume and complexity of communication: MPIRandomAcceess, PTRANS, and MPIFFT.

In MPIRandomAccess, random numbers are generated and written to a distributed table, with local buffering. Performance is measured in billions of updates per second (GUPs) that are performed. Figure 6(a) shows the results of MPI-RandomAccess, comparing the Native+Uverbs, Native+-IPoIB, VNET+VTOE, and VNET+IPoIB cases. Native+-IPoIB achieves 90–100% of Native+Uverbs performance in cases of 8 and 12 processes. However, when the scale increases, Native+IPoIB only delivers 40–75% of Uverbs performance. For the overlay, VNET+VTOE delivers full Native+-IPoIB performance at 8 and 12 processes and 60% of Native+-Uverbs performance as the scale increases.VNET+IPoIB achieves 60–70% of Native+Uverbs performance at scale of 8 and 12 processes, while delivers 40–45% of Native+Uverbs performance at greater scales.

PTRANS does a parallel matrix transpose, exercising the simultaneous communications between pairs of processors. The performance is measured in the total communication capacity (GB/s) of the network. Figure 6(b) shows the result of PTRANS for the Native+Uverbs, Native+IPoIB, VNET+VTOE, and VNET+IPoIB cases. Native+IPoIB achieves 63–80% of Native+Uverbs performance. VNET+-VTOE achieves 100% of Native+IPoIB performance and outperforms Native+IPoIB performance as the scale of the application gets bigger, while VNET+IPoIB frequently delivers 5–10% of the Native+IPoIB performance.

MPIFFT implements a double precision complex one--dimensional Discrete Fourier Transform (DFT). Its performance is measured in Gflop/s. Figure 6(c) shows the result of MPIFFT for the Native+Uverbs, Native+IPoIB, VNET+-VTOE, and VNET+IPoIB cases. Native+IPoIB achieves 65–85% of Native+Uverbs performance. VNET+VTOE achieves near Native+IPoIB performance, while VNET+IPoIB delivers around 19–50% of Native+IPoIB performance.

7.2 Discussion

As shown in the evaluation results, VTOE significantly improves bandwidth and reduces CPU utilization for bandwidth-intensive codes. For large messages and throughput-sensitive applications, VTOE outperforms Native+IPoIB. On the other hand, for the application benchmarks, the network communication consisted of a mixture of small and large packets, and so their performance was determined both by throughput and latency. Recall small-message latency in VNET+-VTOE is still high, about twice Native+IPoIB latency and 10–14x higher than Native+Uverbs, although it has been improved compared with that in VNET+IPoIB by more than 50%. This may explain why some application benchmarks cannot achieve native performance despite VNET+VTOE achieving native throughput in the microbenchmarks.

The long latency mainly comes from the virtual interrupt emulation overhead, and the virtualization overhead is more expansive than TCP kernel stack processing. From the results of application MPIRandomAccess, we can see the high latency has negative impacts on the overall performance. We expect that the optimistic interrupt techniques described elsewhere will reduce this overhead, but have not yet implemented these techniques in VNET+VTOE.

Considering the tradeoff between CPU overhead and network performance, it is again true that MPI applications mix communication and computation, and thus reduced CPU availability and thus more CPU-intensive communication handling may affect computation. However, when the communication is slow, the application cannot make progress even if sufficient CPU time is available. This is of particular concern for MPI applications that do significant collective communication and synchronization.

8. CONCLUSION AND FUTURE WORK

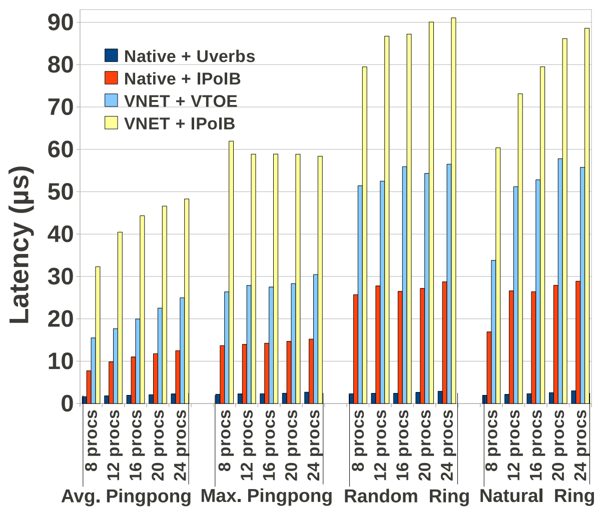
We analyzed the challenges in deploying virtual Ethernet overlays on advanced heterogeneous interconnects such as InfiniBand. The difficulties come from the semantic gap generated by virtualization. To reduce the semantic gap, we proposed, designed, and implemented a virtual TCP offload model to improve virtual Ethernet overlay performance, in terms of throughput, latency, and CPU utilization. This approach improves virtual Ethernet overlay TCP throughput by more than 2.5 times, cuts TCP latency by 50%, and improves TCP application performance.

Although VTOE has reduced VNET+IPoIB latency on InfiniBand by 50%, its latency is still high. Our previous work [6] did a quantitative analysis of virtual overlay overhead. The high overlay latency is due to the *delayed* virtual interrupt delivery into the guests. Optimistic interrupt allows the overlay delivers virtual interrupts to the guest prior to the overlay data processing, overlaps the overlay's processing with the virtual interrupt emulation. Merging this technique into the virtual TCP offload model should reduce latency. Additionally, current VTOE overhead still includes a memory copy between guest kernel space and application buffers. Since advanced interconnects have RDMA features, it should be possible enable remote user space memory copiess without the intervention of either guest kernels or VTOE modules, avoiding all data copies. We are currently implementing such functionality in the VTOE.

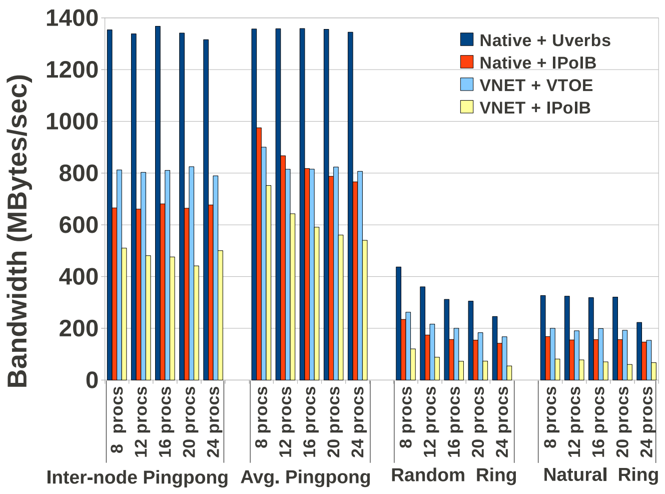
9. REFERENCES

[1] The InfiniBand architecture specification, release 1.2. www.infinibandta.org/specs.

[2] RDMA performance in virtual machines using QDR Infiniband on VMware vSphere 5. http://www.mellanox.com/pdf/whitepapers/RDMA_-Performance_in_Virtual_Machines_using_QDR_-InfiniBand_on_VMware_vSphere5.pdf.



(a) HPCC Latency on InfiniBand



(b) HPCC Bandwidth on InfiniBand

Figure 5: HPCC Latency-Bandwidth benchmark for all of Native+Uverb, Native+IPoIB, VNET+VTOE, and VNET+IPoIB. The results are generally consistent with the previous microbenchmarks, while the ring-based tests show that latency and bandwidth of VNET+VTOE scale and perform better than VNET+IPoIB.

[3] Socket Direct Protocol. en.wikipedia.org/wiki/Sockets_Direct_Protocol.

[4] TCP offload engine. www.networkworld.com/details/653.html.

[5] Clark, D., Jacobson, V., Romkey, J., and Salwen, H. An analysis of TCP processing overhead. *Communications Magazine, IEEE 27*, 6 (june 1989), 23 –29.

[6] Cui, Z., Xia, L., Bridges, P. G., Dinda, P. A., and Lange, J. R. Optimizing overlay-based virtual networking through optimistic interrupts and cut-through forwarding. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Los Alamitos, CA, USA, 2012), SC '12, IEEE Computer Society Press, pp. 99:1–99:11.

[7] Gabriel, E., Fagg, G. E., Bosilca, G., Angskun, T., Dongarra, J. J., Squyres, J. M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R. H., Daniel, D. J., Graham, R. L., and Woodall, T. S. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings of the 11th European PVM/MPI Users' Group Meeting* (September 2004).

[8] Garfinkel, T., and Rosenblum, M. When virtual is harder than real: security challenges in virtual machine based computing environments. In *Proceedings of the 10th conference on Hot Topics in Operating Systems - Volume 10* (Berkeley, CA, USA, 2005), HOTOS'05, USENIX Association, pp. 20–20.

[9] Hale, K., Xia, L., and Dinda, P. Shifting GEARS to enable guest-context virtual services. In *Proceedings of the 9th International Conference on Autonomic Computing (ICAC 2012)* (September 2012).

[10] Innovative Computing Laboratory. HPC challenge benchmark. http://icl.cs.utk.edu/hpcc/.

[11] Intel. Intel Cluster Toolkit 3.0 for Linux. http://software.intel.com/en-us/articles/intel-mpi-benchmarks/.

[12] Jones, S. T. Implicit operating system awareness in a virtual machine monitor. http://citeseerx.ist.psu.edu/viewdoc/download?rep=-rep1&type=pdf&doi=10.1.1.143.6999, 2007.

[13] Jones, S. T., Arpaci-dusseau, A. C., and Arpaci-dusseau, R. H. Antfarm: Tracking processes in a virtual machine environment. In *Proceedings of the USENIX Annual Technical Conf* (2006).

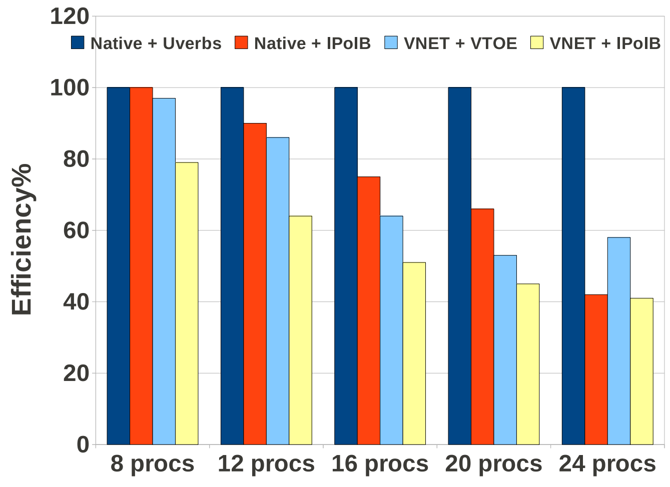
[14] Jones, S. T., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Geiger: monitoring the buffer cache in a virtual machine environment. *SIGARCH Comput. Archit. News 34*, 5 (Oct. 2006), 14–24.

[15] Jones, S. T., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. VMM-based hidden process detection and identification using Lycosid. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments* (New York, NY, USA, 2008), VEE '08, ACM, pp. 91–100.

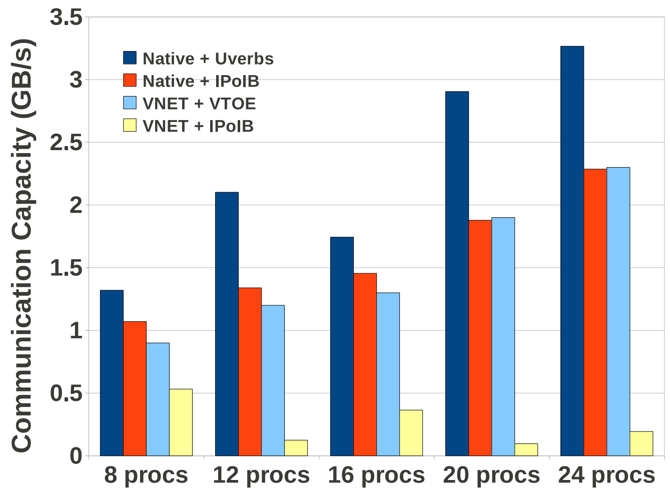
[16] Kashyap, V. IP over InfiniBand (IPoIB) architecture. IETF Network Working Group Request for Comments RFC 4392, April 2006.

[17] L. Xia and Z. Cui and J. Lange and Y. Tang and P. Dinda and P. Bridges. VNET/P: Bridging the cloud and high performance computing through fast overlay networking. In *Proceedings of the 21st ACM International Symposium on High-performance Parallel and Distributed Computing (HPDC)* (June 2012).

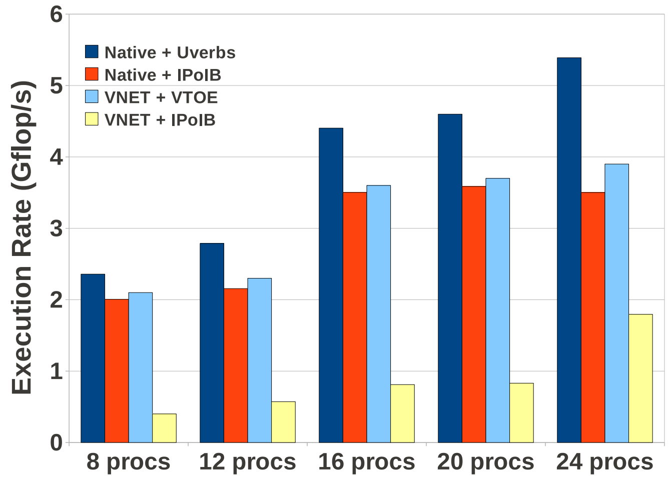
[18] Lange, J., and Dinda, P. Transparent network services via a virtual traffic layer for virtual machines.



(a) HPCC MPIRandomAccess



(b) HPCC PTRANS



(c) HPCC MPIFFT

Figure 6: HPCC application benchmark results. VNET+VTOE approaches Native+IPoIB performance and scalable application performance when supporting parallel application workloads on InfiniBand with rigorous network communication.

In *Proceedings of the 16th IEEE International Symposium on High Performance Distributed Computing (HPDC)* (June 2007).

[19] Lange, J., Pedretti, K., Hudson, T., Dinda, P., Cui, Z., Xia, L., Bridges, P., Gocke, A., Jaconette, S., Levenhagen, M., and Brightwell, R. Palacios and Kitten: New high performance operating systems for scalable virtualized and native supercomputing. In *Proceedings of the 2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)* (April 2010), pp. 1 –12.

[20] Liu, J., Huang, W., Abali, B., and Panda, D. K. High performance VMM-bypass I/O in virtual machines. In *Proceedings of the annual conference on USENIX '06 Annual Technical Conference* (Berkeley, CA, USA, 2006), ATC '06, USENIX Association, pp. 3–3.

[21] Menon, A., Cox, A. L., and Zwaenepoel, W. Optimizing network virtualization in Xen. In *Proceedings of the annual conference on USENIX '06 Annual Technical Conference* (Berkeley, CA, USA, 2006), ATC '06, USENIX Association, pp. 2–2.

[22] Ruth, P., Jiang, X., Xu, D., and Goasguen, S. Towards virtual distributed environments in a shared infrastructure. *IEEE Computer* (May 2005).

[23] Santos, J. R., Turner, Y., Janakiraman, G., and Pratt, I. Bridging the gap between software and hardware techniques for I/O virtualization. In *Proceedings of USENIX 2008 Annual Technical Conference* (Berkeley, CA, USA, 2008), ATC'08, USENIX Association, pp. 29–42.

[24] Sundararaj, A., and Dinda, P. Towards virtual networks for virtual machine grid computing. In *Proceedings of the 3rd USENIX Virtual Machine Research And Technology Symposium (VM 2004)* (May 2004). Earlier version available as Technical Report NWU-CS-03-27, Department of Computer Science, Northwestern University.

[25] Sundararaj, A., Gupta, A., and Dinda, P. Increasing application performance in virtual environments through run-time inference and adaptation. In *Proceedings of the 14th IEEE International Symposium on High Performance Distributed Computing (HPDC)* (July 2005).

[26] Wolinsky, D., Liu, Y., Juste, P. S., Venkatasubramanian, G., and Figueiredo, R. On the design of scalable, self-configuring virtual networks. In *Proceedings of 21st ACM/IEEE International Conference of High Performance Computing, Networking, Storage, and Analysis (Supercomputing 2009)* (November 2009).