Zheng Cui

Candidate

Computer Science

Department

This dissertation is approved, and it is acceptable in quality and form for publication:

Approved by the Dissertation Committee:

Patrick G. Bridges

, Chairperson

Dorian Arnold

Jedidiah R. Crandall

Peter A. Dinda

Nasir Ghani

Enhancing HPC on Virtual Systems in Clouds through Optimizing Virtual Overlay Networks

by

Zheng Cui

B.S., Computer Science, Zhengzhou University, 2003
M.S., Computer Science, University of New Mexico, 2007

DISSERTATION

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Doctor of Philosophy
Computer Science

The University of New Mexico

Albuquerque, New Mexico

July, 2013

Dedication

I dedicate this dissertation to my family; past, present, and future.

Acknowledgments

I would like to thank my advisers, Professor Patrick G. Bridges and Professor Peter A. Dinda, for their guidance and support.

Enhancing HPC on Virtual Systems in Clouds through Optimizing Virtual Overlay Networks

by

Zheng Cui

B.S., Computer Science, Zhengzhou University, 2003
M.S., Computer Science, University of New Mexico, 2007

Ph.D., Computer Science, University of New Mexico, 2013

Abstract

Virtual Ethernet overlay provides a powerful model for realizing virtual distributed and parallel computing systems with strong isolation, portability, and recoverability properties. However, in extremely high throughput and low latency networks, such overlays can suffer from bandwidth and latency limitations, which is of particular concern in HPC environments. Through a careful and quantitative analysis, I identify three core issues limiting performance: delayed and excessive virtual interrupt delivery into guests, copies between host and guest data buffers during encapsulation, and the semantic gap between virtual Ethernet features and underlying physical network features. I propose three novel optimizations in response: optimistic timer-free virtual interrupt injection, zero-copy cut-through data forwarding, and virtual TCP offload. These optimizations improve the latency and bandwidth of the overlay

network on 10 Gbps Ethernet and InfiniBand interconnects, resulting in near-native performance for a wide range of microbenchmarks and MPI application benchmarks.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Data centers and scientific clouds require clusters and supercomputers interconnected with advanced networks, such as high-speed 10 Gbps Ethernet, InfiniBand, and SeaStar interconnects. A virtual *Ethernet overlay network* supports broad classes of standard non-MPI and MPI applications by exposing a uniform Ethernet communication environment.

Current virtual overlay networks are inefficient for HPC applications, however. On high-speed 10 Gbps Ethernet, the performance overhead from the virtualization is 3 times higher latency and 60–70% throughput of native configurations. Additionally, latency exhibits a significant amount of variance. On advanced interconnects such as Infiniband, performance is further reduced due to the *semantic gap* [27] between the Ethernet overlay network and underlying physical network.

My thesis is that such virtual overlay networks can be optimized to deliver near-native HPC application performance. This dissertation analyzes the HPC performance challenges of virtual overlay networks, and identifies factors impacting overlay performance. These factors include context-switch overheads, delayed virtual interrupts, excessive virtual interrupts, high-resolution noise, and the semantic gap

between virtual Ethernet and heterogeneous interconnects. To address these issues, the dissertation proposes several optimizations, such as a Virtual Machine Monitor (VMM) level virtual overlay, optimistic interrupts, cut-through data forwarding, noise isolation, and virtual TCP offload. Performance results show that these optimizations allow virtual overlay networks to provide near-native performance to HPC applications.

The remainder of this chapter provides additional details on HPC in cloud system, virtual networking, and virtual overlay networking. It also summarizes the contributions of this work, and outlines the remainder of the dissertation.

1.1 HPC in Cloud Computing Systems

Cloud computing in the "infrastructure as a service" (IaaS) model has the potential to provide economical and effective on-demand resources for high performance computing. In this model, an application is mapped into a collection of virtual machines (VMs) that are instantiated as needed, and at the scale needed. As described in Section 2.3.3, such systems can also be adaptive, autonomically selecting appropriate mappings of virtual components to physical components to maximize application performance or other objectives. For loosely-coupled applications, this concept has readily moved from research [23, 65] to practice [60]. However, *tightly-coupled* scalable high performance computing (HPC) applications currently remain the purview of resources such as clusters and supercomputers. This dissertation seek to extend the adaptive IaaS cloud computing model into these regimes, allowing an application to dynamically span both kinds of environments.

1.1.1 Cloud Computing

Cloud computing is the use of computing resources (hardware and software) that are delivered as a service over a network (typically the Internet). The name comes from the use of a cloud-shaped symbol as an abstraction for the complex infrastructure it contains in system diagrams. Cloud computing entrusts remote services with a user's data, software, and computation. In the business model using software/platform/infrastructure as a service, users are provided access to application/system software/hardware and databases.

In the most basic cloud-service model, providers of IaaS offer physical computers or (more often) virtual machines and other resources, such as virtual networking and virtual storage. A *hypervisor* runs the virtual machines as guests. Pools of hypervisors within the cloud operational support system can support large numbers of virtual machines and the ability to scale services up and down according to customers' varying requirements.

1.1.2 High Performance Computing

High performance computing (HPC) uses supercomputers and computer clusters to solve advanced computation problems. The term is most commonly associated with computing used for scientific research or computational science. A related term, high-performance technical computing (HPTC), generally refers to the engineering applications of cluster-based computing (such as computational fluid dynamics and the building and testing of virtual prototypes). Recently, HPC has been applied to business uses of cluster-based supercomputers, such as data warehouses, line-of-business (LOB) applications, and transaction processing.

HPC includes data intensive computing and all forms of computational science.

HPC workloads are incredibly important today and the market segment is growing very quickly driven by the plunging cost of computing and the business value of understanding large data sets deeply.

High performance computing allows scientists and engineers to solve complex science, engineering, and business problems using applications that require high bandwidth, low latency networking, and very high compute capabilities. Since HPC is focused on performance, the computational network is usually a high-speed network such as InfiniBand or Myrinet 10G.

1.1.3 HPC in Clouds

In many respects, HPC workloads are natural for the cloud in that they are large scale and consume vast machine resources. Some HPC workloads are very bursty with large clusters being needed for only short periods of time. For example, semiconductor design simulation workloads are incredibly computationally intensive and need to be run at large scale, but only during some phases of the design cycle. Having more resources can get a design completed more quickly, and possibly allow additional verification runs that save millions of dollars by avoiding a design flaw. Using cloud resources, resource usage can change size over the course of the project or be freed up when no longer needed.

Many people view HPC as non-cloud hostable because these workloads need high performance, direct access to underlying server hardware. The virtualization overhead common in most cloud computing systems, however, can reduce system performance. This is particularly true of virtual networking technologies. To achieve the HPC application performance that the tightly-coupled resources are capable of, the virtual network cannot introduce significant overhead relative to the native hardware.

1.2 Virtual Networking

Considerable effort has also gone into achieving low-overhead network virtualization and traffic segregation within an individual data center through extensions or changes to the network hardware layer [59, 28, 45]. While these tools strive to provide uniform performance across a cloud data center (a critical feature for many HPC applications), they do not provide the same features once an application has migrated outside the local data center, spans multiple data centers, or involves HPC resources. Furthermore, they lack compatibility with the more specialized interconnects present on most HPC systems.

Beyond the need to support the envisioned computing model across today's and tomorrow's tightly-coupled HPC environments, data center network design and cluster/supercomputer network design seem to be converging [12, 29]. This suggests that future data centers deployed for general purpose cloud computing will become an increasingly better fit for tightly-coupled parallel applications.

Current VMs (Xen Para-virtualized IB and VMWare vSphere) do not abstract high-speed networks to a more general abstraction like Ethernet. While they can provide excellent performance, these approaches have disadvantages in terms of portability, location-independence, and host device reallocation.

1.3 Virtual Overlay Networking

Virtual overlay networks provide a uniform communication environment. They let the user treat his VM as if it was on a local LAN, while allowing VMs to be migrated transparently across different networks and data centers by managing flow routing through the overlay. However, virtual overlay networks suffer latency and bandwidth limitations on extremely low-latency and high-bandwidth networks.

1.3.1 Overview

Current adaptive cloud computing systems use software-based overlay networks to carry inter-VM traffic. For example, the VNET/U system, which is described in more detail in Section 2.3.3, combines a simple networking abstraction within the 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 [72, 64] and others that expose different abstractions to the VMs [77] have been under continuous research and development for several years. Current virtual networking systems have sufficiently low overhead to effectively host loosely-coupled scalable applications [21], but their performance is insufficient for tightly-coupled applications [61].

1.3.2 Overlay Performance Challenges

Although overlay networks have advantages, they still suffer from performance degradation. These performance challenges include latency and bandwidth overheads, and semantic gap between virtual overlay features and underlying physical network features.

Latency and Bandwidth Overheads

On high-end networks, virtual overlays can have several times higher latency and much lower throughput than native configurations. Additionally, latency usually exhibit a significant amount of variance. My analysis in Chapter 3 shows that these

performance limitations are primarily due to two issues: *delayed* and *excessive* virtual interrupts to guest virtual machines (VMs), and copy operations between host and guest buffers which reduce the number of delivered packets per interrupt. These are general issues that are likely to occur in any virtual overlay network.

Semantic Gap on Heterogeneous Interconnects

In addition, current virtual overlay networks are unable to deliver good 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 [27] is inevitable in virtual overlays whenever the semantics of the underlying physical network is 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 [57, 67, 39, 41, 40, 42], comparatively little work has been done on bridging this gap for virtual overlay networks.

1.4 Contributions

To address these challenges, I have designed, implemented, and evaluated VNET/P, which shares its model and vision with VNET/U, but is designed to achieve near-native performance in the 1 Gbps and 10 Gbps switched networks common in clusters today, as well as to operate effectively on top of even faster networks, such as Infiniband and Cray Gemini. VNET/U and this model are presented in more detail in

Section 2.3.3.

In this dissertation, I also describe additional techniques, specifically optimistic interrupts, cut-through forwarding, and noise-isolation, that bring bandwidth to near-native levels for 10 Gbps homogeneous Ethernet, cut VNET/P latency in half, reduce virtual overlay network variability, and frequently deliver native MPI application performance on 10 Gbps Ethernet.

I also describe a virtual TCP offload optimization for improving HPC application performance on heterogeneous high-end interconnects, such as InfiniBand, Cray SeaStar, and Gemini interconnects, to reduce duplicated Reliable-Connected (RC) protocol processing overhead.

The contributions are as follows:

- The design and implementation of a virtual networking system, VNET/P, that extends virtual networking for VMs to clusters and supercomputers with high performance networks. The design could be applied to other VMMs and virtual network systems.
- A quantitative study of the VMM-level virtual overlay network that identifies the system challenges in reducing latency and variability, improving throughput, and reducing semantic gap between virtual networks and underlying physical interconnects.
- The design and implementation of two novel optimizations, *Optimistic Interrupts* and *Cut-through Forwarding*, that address the reduced bandwidth, increased latency, and network performance variability issues in virtual overlay network systems.
- The design and implementation of *virtual TCP offload* support to bridge the semantic gap between the guest application, the overlay network, and the un-

derlying high-end interconnect, in enhancing a virtual Ethernet overlay network.

- An extensive evaluation of the VMM-level virtual Ethernet overlay with the proposed optimizations on 1 and 10 Gbps Ethernet networks, and InfiniBand interconnects. This evaluation shows that this approach provides performance with negligible overheads on 1 Gbps Ethernet, and manageable overheads on 10 Gbps Ethernet and InfiniBand. The optimistic interrupts and cut-through forwarding optimizations reduce latency by 50%, and increases throughput by more than 30%, and provide native MPI application benchmark performance on 10 Gbps Ethernet networks. Adding offload capabilities to the virtual Ethernet overlay reduces TCP latency, and substantially increases TCP throughput and application performance on sophisticated InfiniBand interconnects.

Through the use of low-overhead overlay-based virtual networking in high bandwidth, low-latency environments such as current clusters and supercomputers, and future data centers, this work seeks to make it practical to use virtual networking at all times, even when running tightly-coupled applications on such high-end environments. This would allow people to seamlessly and practically extend the already highly effective adaptive virtualization-based IaaS cloud computing model to such environments.

1.5 Dissertation Outline

The rest of the dissertation is organized as follows: Chapter 2 presents background on the Palacios VMM, NIC virtualization, virtual networking, overlay networking, the VNET model, virtual network optimization, and high-end interconnects. Chapter 3 then analyzes the performance of overlay networks, providing insight into the

fundamental challenges of overlay support for high-speed network devices. Chapter 4 follows with a description of my new optimizations for virtual overlay networks on high-speed interconnects. Chapter 5 describes an implementation of the proposed optimizations, and Chapter 6 follows with an extensive evaluation of the impact of these optimizations using microbenchmarks and more complex application benchmarks. Finally, Chapter 7 concludes.

Chapter 2

Related Work

This chapter describes related work to virtual overlay networking optimizations. It starts with a brief introduction of virtualization and Palacios VMM, then provides more details of network virtualization and commodity virtual networking models. It ends with a detailed description of virtual overlay networking and the VNET/U model together with high-end fast interconnects discussion.

2.1 Virtualization for Scientific Computing

2.1.1 Overview

Virtualization is a combination of software and hardware engineering that creates *Virtual Machines* (VMs) – an abstraction of the computer hardware that allows a single machine to act as if it where many machines. The *hypervisor* or *Virtual Machine Monitor* (VMM) [11, 9, 4, 2] is the control system at the core of virtualization. It acts as the control and translation system between the VMs and the hardware. A *host* operating system (OS) is the original OS installed on a computer.

A *guest* operating system is the OS installed in a virual machine. *Full system virtualization* [9, 4, 2] provides full compatibility at the hardware level, allowing existing unmodified applications and OSes to run. *Paravirtualization* [11] is a virtualization technique that presents a software interface to virtual machines that is similar but not identical to that of the underlying hardware. Paravirtualization requires the guest operating system to be explicitly ported for the para-API, a conventional OS distribution that is not paravirtualization-aware cannot be run on top of a paravirtualizing VMM. When talking about virtualization, a *domain* is one of the virtual machines that run on the system.

2.1.2 HPC with Virtualization

HPC communities are increasingly turning to virtualization as a means of deploying and managing large scale of computing systems with cloud computing. For loosely-coupled applications, current virtual machine monitors (VMMs) and other virtualization mechanisms present negligible overhead for CPU and memory intensive workloads [35, 58].

However, it is very challenging to run tightly-coupled applications in clouds. Tightly-coupled applications remain the purview of clusters and supercomputers, they are communication-intensive, and they are very sensitive to performance overheads, particularly unpredictable overheads. It is very important for the virtual networking system to introduce minimal overhead.

2.1.3 Palacios VMM

Palacios is an OS-independent, open source, BSD-licensed, publicly available, embeddable VMM designed as part of the V3VEE project (http://v3vee.org). The

V3VEE project is a collaborative community resource development project involving Northwestern University, the University of New Mexico, the University of Pittsburgh, Sandia National Labs, and Oak Ridge National Lab. Detailed information about Palacios can be found elsewhere [49]. Palacios is capable of virtualizing large scale (4096+ nodes) supercomputers with only minimal performance overheads [48]. Palacios's OS-agnostic design allows it to be embedded into a wide range of different OS architectures. Four embeddings currently exist. In this dissertation, I employ the Linux and Kitten embeddings.

2.2 Network Virtualization

2.2.1 Overview

In network virtulization [10, 6, 5, 17, 18], a single system is configured with containers, such as the Xen domain, combined with hypervisor control programs or pseudo-interfaces such as the VNIC, to create a "network in a box." This solution improves overall efficiency of a single system by isolating applications to separate containers and/or pseudo interfaces.

Virtual networking systems provide a service model that is compatible with an existing layer 2 or 3 networking standard. Examples include Xen networkng, VMware vSphere, VIOLIN [38], ViNe [75], VINI [14], SoftUDC VNET [44], OCALA [43], WoW [26], and the emerging VXLAN standard [56]. Like VNET, VIOLIN, SoftUDC, WoW, and VXLAN are specifically designed for use with virtual machines. Of these, VIOLIN allows for the dynamic setup of an arbitrary private layer 2 and layer 3 virtual network among VMs. The key contribution of the overlay described in this dissertation is to show that this model can be made to work with minimal overhead even in extremely low-latency, high-bandwidth environments.

2.2.2 Xen Networking

A Xen guest typically has access to one or more paravirtualised (PV) network interfaces. A paravirtualised network device consists of a pair of network devices. The first of these (the frontend) resides in the guest domain while the second (the backend) resides in the backend domain. A similar pair of devices is created for each virtual network interface. The front and backend devices are linked by a virtual communication channel. Guest networking is achieved by arranging for traffic to pass from the backend device onto the wider network, e.g. using bridging, routing or Network Address Translation (NAT).

These PV interfaces enable fast and efficient network communications for domains without the overhead of emulating a real network device. However, modified and dedicated paravirtualized device drivers for PV network devices need be developed specifically by Xen providers.

2.2.3 VMware ESX Networking

In an ESX system [9], a virtual machine can be configured with one or more virtual Ethernet adapters, each of which each has its own IP address and MAC address. As a result, virtual machines have the same properties as physical machines from a networking standpoint.

The key virtual networking components in ESX are virtual Ethernet adapters, used by individual virtual machines, and virtual switches, which connect virtual machines to each other and connect virtual machines to external networks.

2.2.4 Hyper-V Networking

On a Hyper-V host, virtual networks are created to define various networking topologies for virtual machines and the host. In the VMM, there are three different types of virtual networks on a Hyper-V host.

Private virtual network: this type of network allows communication between virtual machines on the same host but not with the host or with external networks. A private virtual network does not have a virtual network adapter in the host operating system, nor is it bound to a physical network adapter. Private virtual networks are typically used to isolate virtual machines from network traffic in the host operating system and in the external networks.

Internal virtual network: this type allows communication between virtual machines on the same host and between the virtual machines and the host. An internal virtual network is not bound to a physical network adapter. It is typically used to build a test environment to connect to the virtual machines from the host operating system.

External virtual network (Physical network adapter): this type allows virtual machines to communicate with each other and with externally located servers, and optionally with the host operating system. An external virtual network is bound to a physical network adapter and, optionally, it can have a virtual network adapter in the host operating system. An external virtual network can be used to allow virtual machines to access a perimeter network (also known as a screened subnet, or DMZ) and not expose the host operating system.

2.2.5 Summary

The most common virtual networks do not provide the guest VMs with a consistent network abstraction which is independent of the location and the underlying devices. In addition, specific device drivers may be required from VMM providers to have virtual NICs functional in guest OSes. In addition, as long as a physical device has been allocated or bound to some virtual device, it is very expensive to deallocate and reallocate this physical device to another VM.

In contrast with such virtual networks, the overlay network described in this dissertation provides fast access to an overlay network, which includes encapsulation and routing. It makes a set of VMs appear to be on the same local Ethernet regardless of their location anywhere in the world and their underlying hardware. This dissertation shows that this capability can be achieved without significantly compromising performance when the VMs happen to be very close together.

2.3 Overlay Networks

2.3.1 Overview

An overlay network is a computer network which is built on the top of another network. Nodes in the overlay can be thought of as being connected by virtual or logical links, each of which corresponds to a path, perhaps through many physical links, in the underlying network. For example, distributed systems such as cloud computing, peer-to-peer networks, and client-server applications are overlay networks because their nodes run on top of the Internet. The Internet was originally built as an overlay upon the telephone network, while today (through the advent of VoIP), the telephone network is increasingly turning into an overlay network built on top of

the Internet.

With virtualization, overlay networks can implement extended network functionality on top of physical infrastructure, for example to provide resilient routing (e.g, [13]), multicast (e.g. [34]), and distributed data structures (e.g., [69]) without any cooperation from the network core. Overlay networks use end-systems to provide their functionality in this setting. VNET is an example of a specific class of overlay networks, namely virtual networks.

2.3.2 Connections with Virtual Networking

The overlay described in this dissertation could itself leverage some of the related work described above. For example, effective NIC virtualization might allow us to push encapsulation directly into the guest, or to accelerate encapsulation via a split scatter/gather map. Mapping unencapsulated links to VLANs would enhance performance on environments that support them. There are many options for implementing virtual networking and the appropriate choice depends on the hardware and network policies of the target environment. In the proposed overlay, this dissertation makes the choice of minimizing these dependencies.

2.3.3 VNET Model and VNET/U

VNET Model

The VNET model was originally designed to support adaptive computing on distributed virtualized computing resources within the Virtuoso system [19], and in particular to support the adaptive execution of a distributed or parallel computation executing in a collection of VMs potentially spread across multiple providers or supercomputing sites. The VNET model has the following requirements:

- VNET makes within-VM network configuration the sole responsibility of the VM owner.
- VNET provides location independence to VMs, allowing them to be migrated between networks and from site to site, while maintaining their connectivity, without requiring any within-VM configuration changes.
- VNET provides hardware independence to VMs, allowing them to use diverse networking hardware without requiring the installation of specialized software.
- VNET provides minimal overhead, compared to native networking, in the contexts in which it is used.

The VNET model meets these requirements by carrying the user's VMs' traffic via a configurable overlay network. The overlay presents a simple layer 2 networking abstraction: a user's VMs appear to be attached to the user's local area Ethernet network, regardless of their actual locations or the complexity of the VNET topology/properties. Further information about the model can be found elsewhere [72].

The VNET overlay is dynamically reconfigurable, and can act as a locus of activity for an adaptive system such as Virtuoso. Focusing on parallel and distributed applications running in loosely-coupled virtualized distributed environments (e.g., IaaS Clouds), VNET can be effectively used to:

1. monitor application communication and computation behavior [31, 30]),
2. monitor underlying network behavior [32],
3. formulate performance optimization problems [74, 71], and
4. address such problems through VM migration and overlay network control [73], scheduling [52, 53], network reservations [50], and network service interposition [47].

These and other features that can be implemented within the VNET model have only marginal utility if carrying traffic via the VNET overlay has significant overhead compared to the underlying native network.

VNET/U

VNET/U is a user-level implementation of the VNET model. As a user-level system, it readily interfaces with VMMs such as VMware Server and Xen, and requires no host changes to be used, making it very easy for a provider to bring it up on a new machine. Further, it is easy to bring up VNET daemons when and where needed to act as proxies or waypoints. A VNET daemon has a control port which speaks a control language for dynamic configuration. A collection of tools allows for the wholesale construction and teardown of VNET topologies, as well as dynamic adaptation of the topology and forwarding rules to the observed traffic and conditions on the underlying network.

VNET/U supports a dynamically configurable general overlay topology with dynamically configurable routing on a per MAC address basis. The topology and routing configuration is subject to global or distributed control (for example, by the VADAPT [73]) part of Virtuoso. The overlay carries Ethernet packets encapsulated in UDP packets, TCP streams with and without SSL encryption, TOR privacy-preserving streams, and others. Because Ethernet packets are used, the VNET abstraction can also easily interface directly with most commodity network devices, including virtual NICs exposed by VMMs in the host, and with fast virtual devices (e.g., Linux virtio network devices) in guests.

The last reported measurement of VNET/U showed it achieving 21.5 MB/s (172 Mbps) with a 1 ms latency overhead communicating between Linux 2.6 VMs running in VMware Server GSX 2.5 on machines with dual 2.0 GHz Xeon processors [47]. A

current measurement, described in Chapter 6, shows 71 MB/s with a 0.88 ms latency. VNET/U's speeds are sufficient for its purpose in providing virtual networking for wide-area and/or loosely-coupled distributed computing. They are not, however, sufficient for use within a cluster at gigabit or greater speeds. Making this basic VM-to-VM path competitive with hardware is the focus of this dissertation.

VNET Model in This Dissertation

The VNET/P system described in this dissertation is compatible with, and compared to, the previous VNET implementation, VNET/U. In VNET/P, VNET is moved directly into the VMM to avoid context-switch overheads.

2.4 Virtual Networking Optimization

2.4.1 Overview

The high processing overheads incurred in virtualized networks limit overall I/O performance. For example, the number of processing cycles consumed for receiving network packets is 4.7 times higher in Xen than in native Linux [1]. This high cost limits network throughput for Xen guests to only 2.9 Gb/s using a 10Gbps NIC on a modern server which is able to achieve 9.3 Gb/s when running native Linux.

Several efforts have been made to improve virtual networking performances on fast interconnects. The most common and efficient optimizations fall into two categories: virtual NIC optimization and virtual interrupt optimization.

2.4.2 Virtual NIC Optimization

There has been a wide range of work on optimizing high-speed network interface performance in virtual machines [70, 54, 62, 79] focused on paravirtualizing the NIC, or bypassing the host OS, virtual machine monitor, and sometimes the guest OS. While a purely software-based virtualized network interface has high overhead, many techniques have been proposed to support simultaneous, direct-access network I/O. For example, some work [54, 62] has demonstrated the use of self-virtualized network hardware that allows direct guest access, thus providing high performance to untrusted guests. Willmann et al have developed a software approach that also supports concurrent, direct network access by untrusted guest operating systems [68]. In addition, VPIO [79] can be applied on network virtualization to allow virtual passthrough I/O on non-self-virtualized hardware. Virtual WiFi [78] is an approach to provide the guest with access to wireless networks, including functionality specific to wireless NICs.

The work described in this dissertation leverages paravirtualized NICs to improve overlay performance, and extends them with additional optimizations appropriate for overlay networks. Approaches that completely bypass the host and virtual machine monitor, on the other hand, cannot be used in virtual network overlays because they make it impossible for the VMM to route and manage an overlay network.

2.4.3 Virtual Interrupt Optimization

Most work on optimizing software network virtualization has focused on interrupt processing; this focus is well-founded, as my analysis in Chapter 3 demonstrates. In particular, research has examined various interrupt handling schemes for virtual networking systems such as polling, regular interrupts, interrupt coalescing, and disabling and enabling interrupts [66]. Studies that specifically examined virtual

interrupt coalescing techniques attempt to avoid excessive virtual interrupts and improve throughput by coalescing interrupts in virtual NICs similar to how host NICs coalesce interrupts [20, 15]. Unfortunately, these techniques control virtual interrupt frequency using a high-frequency periodic timer that has high overheads and generate substantial OS noise.

2.4.4 InfiniBand Virtualization

Currently two approaches are very popular in virtualizing InfiniBand with high performance: VMM-bypass [55] and Passthrough [49, 18, 7]. 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.

2.5 Summary

In this chapter, I described related work in HPC with virtualization, network virtualization, overlay networks, and virtual network optimizations. To support tightly-coupled applications with minimal overhead, the virtualization layer needs to provide a fast network system. The virtual networks and optimizations described in this chapter illustrate the many different approaches for realizing and optimizing traffic transmission and reception. Non-overlay based virtual networks in general deliver good performance by either using passthrough network devices or VMM-bypass approaches. However, these approaches require the guest OS control or partially control physical devices, and thus virtual network systems lose the control of host devices, VM portability, and location independence. Current overlay-based virtual networks provide maximal flexibility and portability, however, they suffer performance degradation due to context switch overheads.

In contrast, this dissertation describes several approaches to optimize virtual overlay network performance in terms of latency, bandwidth, and CPU utilizations. These approaches reduce context switch overheads by moving overlay networks into VMMs, reduce latency and improve throughput by managing virtual interrupts and data transfer buffers, and reduce semantic differences by virtual TCP offload. With these optimizations, overlay networks deliver high-performance services while maintaining the portability, location-independence, and hardware-independence properties.

Chapter 3

Analysis

In this chapter, I analyze the performance overheads and difficulites that a virtual overlay may introduce. I first discuss the known performance problems of user-level overlays, particularly, context switch overheads. I then measure and present the overheads in VMM-level overlay, VNET/P, described in more detail in Chapter 4. I also discuss the challenge that high-end interconnects present to virtual overlay networks.

3.1 User-level Overlay Context Switch Overhead

VNET/U is implemented as a user-level system. As a user-level system, it readily interfaces with VMMs such as VMware Server and Xen, and requires no host changes to be used, making it very easy for a provider to bring it up on a new machine.

However, the last reported measurement of VNET/U showed it achieving 21.5 MB/s (172 Mbps) with a 1 ms latency overhead communicating between Linux 2.6 VMs running in VMware Server GSX 2.5 on machines with dual 2.0 GHz Xeon processors [47]. A current measurement, described in Chapter 6, shows 71 MB/s with

a 0.88 ms latency. VNET/U's speeds are sufficient for its purpose in providing virtual networking for wide-area and/or loosely-coupled distributed computing. They are not, however, sufficient for use within a cluster at gigabit or greater speeds. Making this basic VM-to-VM path competitive with hardware is the focus of this dissertation.

VNET/U is fundamentally limited by the kernel/user space transitions needed to handle a guest's packet send or receive. Context switch causes unavoidable system overhead. The cost of a context switch may come from several aspects. The processor registers need to be saved and restored, the OS kernel code (scheduler) must execute, the TLB entries need to be reloaded, and processor pipeline must be flushed [24, 51]. These direct costs are associated with almost every context switch in a multitasking system. In addition, context switch leads to cache sharing between multiple processes, which may result in performance degradation. This indrect cost varies for different workloads with different memory access behaviors and for different architectures. Work [24, 51] did a quantitive analysis and shows context switches are expensive.

In a VMM-level overlay, the virtual overlay network implementation is moved directly into the VMM to avoid such transitions. This potentially improves overlay performance.

3.2 VMM-level Homogeneous and Heterogeneous Overlay

In addition to context switch overheads, virtualization itself also impacts overlay performance. To more fully understand these overheads, I instrumented and traced the performance of packet reception and transmission in a VMM-level overlay running in a Linux host OS on AMD Opteron systems with 10Gbps Ethernet adapters (more

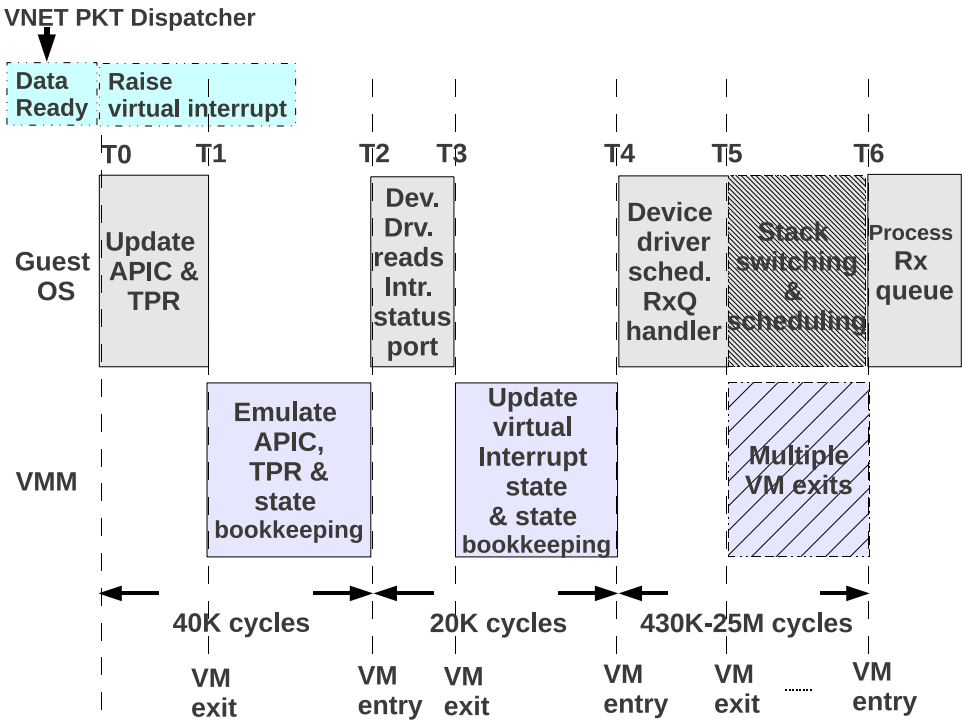


Figure 3.1: Virtual interrupt time line

details on the test systems are provided in Chapter 6). The analysis highlights three major challenges to overlay networks in high-speed networks: *delayed virtual interrupts*, *excessive virtual interrupts*, and *high-resolution timer noise*.

3.2.1 Delayed Virtual Interrupts

A VMM-level overlay packet dispatcher raises a virtual interrupt to the guest OS when a packet arrives in the virtual NIC's receive buffer, however, the time at which the guest starts to process the receive queue is dramatically delayed compared to the native case. This is because virtual interrupt handling touches virtual device registers (e.g. both APIC and device registers), incurring multiple rounds of trap-and-emulation.

Figure 3.1 illustrates the time line of a virtual interrupt. The cost of a typical trap-and-emulation of an interrupt controller register operation is around 5000 cycles, based on the experiments on an AMD cluster. As a result, each virtual interrupt must introduce at least 10K cycles of latency. In addition, the VMM must perform bookkeeping on both the guest and host states for each trap, increasing the effective length of each trap-and-emulate cycle. For example, the average processing from a VM exit to the next VM entry in Palacios is between 10K and 200K cycles. As a result, when the virtual device driver's interrupt handler is invoked, around 40K cycles have elapsed since the virtual interrupt was delivered (T0 – T2). The virtual device driver's interrupt handler performs additional register accesses that must also be trap-and-emulated (T2 – T4), and additional exits result when the guest OS switches stacks and schedules tasks. As a result, when the guest OS finally starts to process the virtual NIC's receive queue, between 430K to 25M cycles have passed (T4 – T6). The results of this profiling are shown with arrows indicating time intervals in Figure 3.1.

3.2.2 Excessive Virtual Interrupts

After processing an inbound packet, a VMM-level overlay packet dispatcher may interrupt a guest OS immediately, indicating the packet's readiness to the guest OS. Although this scheme provides correctness and low per-packet latencies, it causes excessive virtual interrupts that reduce the amount of guest CPU time actually available for packet processing. Physical network interfaces typically use interrupt coalescing to avoid this problem, where interrupts are delayed a bounded amount of time to balance interrupt delivery latency while reducing CPU interrupt processing overheads. Unfortunately, such schemes are challenging in virtual NICs, as described in the following subsection.

3.2.3 High Resolusion Timer Noise

In hardware controllers, fine-grained timers are used in conjunction with interrupt coalescing to bound the latency of I/O completion notifications. Such timers are hard and inefficient to use in a hypervisor. High-performance host NICs, for example, typically bound interrupt coalescing delays in the range of tens or hundreds of microseconds because delays longer than this can significantly impact the performance of latency-sensitive applications. Operating systems, however, typically only provide timers with granularities in the millisecond range, and even timers of this resolution are known to cause performance problems in high-performance environments. These high resolution timers also impact application performance through what is termed *OS noise* [33, 22].

3.3 Semantic Gap on High-end Interconnects

VMM-level virtual Ethernet overlays present an additional challenges on more advanced interconnect: the semantic gap between overlay features and physical interconnect features. In particular, guest OSes see only a relatively simple Ethernet interface and so do not provide the overlay higher-level semantic information about desired network semantics. This lack of knowledge about the guest-level communications can lead to I/O performance degradation.

When deploying a virtual Ethernet overlay on top of advanced interconnects, there are two straightforward approaches to address 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 such features without guest knowledge. Each has significant performance problems, described below.

3.3.1 Use minimal interconnect features

The first alternative is to use minimal interconnect features to transport guest traffic, learning higher-level functionality such as reliability to the guest. 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 dissertation, for example, impose a 2KB MTU. Small MTUs such as this 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 [16]. 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 host, as described in Section 3.2.1.

3.3.2 Translate to advanced interconnect 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 the 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 reliable Infiniband connections in the overlay causes the guest to unnecessarily send TCP connection establishment requests and acknowledgments over the reliable Infiniband connection. The guest also unnec-

essarily checksums the incoming packets and performs congestion and flow control activities. All of this increases packet latency and guest CPU processing requirements, reducing application performance.

3.4 Summary

In this chapter, I analyzed the challenges in deploying virtual overlay networks over high-speed networks. These challenges include user/kernel context switch overheads, delayed virtual interrupts injection, excessive virtual interrupt deliveries, high-resolution timer noise, and semantic gap between virtual overlay features and underlying physical network features.

Chapter 4

Optimizations

4.1 Overview

The next focus of this dissertation is on optimizing virtual overlay performance. To reduce guest user/kernel context-switch overheads, I move the virtual overlay network into VMM from the user-space. I then propose several optimizations with two levels of goals: 1) to increase overlay throughput, reduce latency and performance variation in general for both homogeneous and heterogeneous interconnects; and 2) to increase overlay throughput, reduce latency and CPU overheads for heterogeneous interconnects, such as InfiniBand.

To address the challenges described in the previous chapter, I propose three optimizations for virtual overlay network implementations: optimistic interrupts, cut-through forwarding, and virtual TCP offload. These optimizations act together to reduce per-packet latencies and improve throughput by overlapping virtual overlay's packet handling with guest interrupt processing. These optimizations also leverage the predictable environment of a low-noise host kernel which I also use to reduce virtual network performance variability.

For the first goal, I concentrate on configurations with dedicated device assignment. In these scenarios the receive ring and interrupt channel of a virtual NIC is explicitly bound to a single physical NIC. This is an important use-case in scientific clouds, high-end data centers, and virtual supercomputer environments which seek the management advantages of overlays without sacrificing optimal communication performance. In fact, hardware techniques used in high-performance networking, such as hardware passthrough and device assignment (e.g. packet hashing, message-signaled interrupts, per-flow and per-core receive rings, and single-root I/O virtualization), can all also potentially be used to support assigning portions of host NICs to virtual NICs in virtual network overlay systems.

The optimizations described in this chapter are also potentially useful in cases without a one-to-one correspondence between host and guest NICs. However, these optimizations rely on careful prediction and control of the timing between events among the host, VMM, and guest. Such timing is more difficult to predict if incoming packets could be delivered to a wide range of guests.

4.2 VMM-level Virtual Overlay

The first optimization presented is to reduce guest user space and kernel space context switch overhead during packet relay.

To reduce context switches, performance-critical components of the overlay are moved into the VMM. A general VMM-level overlay network typically has several performance critical components:

- Virtual interfaces: this component conveys guest packets between the application VM and the VMM. One or more packets can be conveyed from an application VM to VMM with a single VM exit, and from VMM to the application

VM with a single VM exit+entry.

- Routing control: this component is primarily responsible for routing and dispatching raw guest packets. It intercepts all guest packets from virtual interfaces that are associated with the overlay, and forwards them either to VMs on the same host machine or to the outside network through the interfaces to external networks. Each packet is routed based on its source and destination interface addresses.

- Interfaces to external networks: these components direct packets between the overlay and the physical network.

4.3 Optimistic Interrupts

The first optimization I propose to reduce per-packet latencies and interrupt processing overheads is *optimistic interrupts*. Normally, the overlay system will inject a single interrupt when it finishes copying (and de-encapsulating) data from the host NIC to the virtual NIC. With optimistic interrupts, I instead define two specific windows during which a virtual interrupt *may* be injected. First, optimistic interrupts can inject an Early Virtual Interrupt (EVI) *before* it begins moving data to the virtual NIC, thus allowing the guest to begin interrupt handling *while the overlay system is moving data from the host NIC*. Second, optimistic interrupts *may* inject an interrupt when the host device driver has finished processing all of the packets in the device queue from the arrival of a coalesced interrupt. The decision on whether to inject an interrupt at this point is made when an End-of-Coalesce (EoC) notification arrives from the host device driver. The decision made depends on whether a previous EVI was successfully handled by the guest and on how quickly the host processes incoming packets.

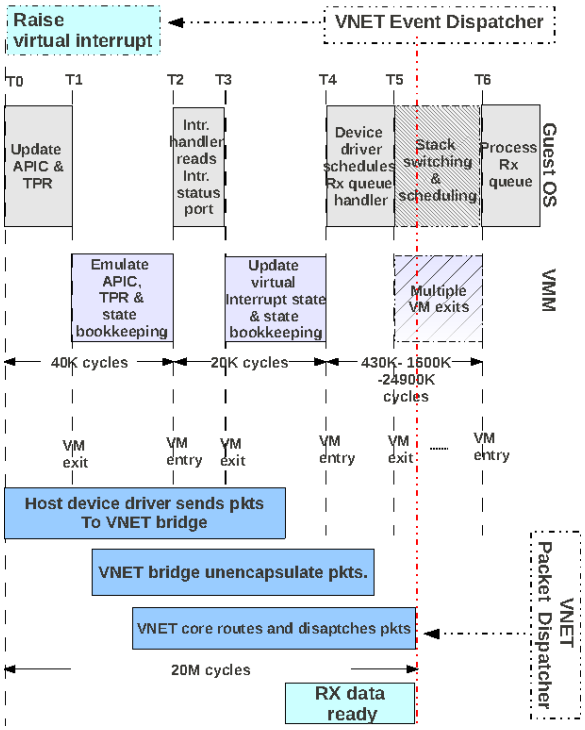


Figure 4.1: Early virtual interrupt optimization to reduce latency

4.3.1 Early Virtual Interrupt (EVI) delivery

Figure 4.1 illustrates the early virtual interrupt optimization for reducing latency. Instead of waiting for the packet to be copied into the virtual NIC receive buffer before raising a virtual interrupt, EVI interrupts the virtual NIC immediately when the host device driver identifies the data arrival event. This allows overlay de-encapsulation and packet data movement to occur concurrently with the VMM's emulation of virtual interrupts and VM exits triggered by guest OS stack switches and context switches.

In essence, virtual interrupts with EVI synchronize packet arrival events with interrupt processing in virtual machines, reducing the interrupt delay described in Section 3.2.1. This optimization is possible because device assignment allows the

overlay to know which virtual NIC a host NIC interrupt is associated with.

EVI's overall goal is to raise the virtual interrupt so the guest begins processing the packet queue immediately after the first packet has been marked in the virtual NIC receive buffer. This, however, is challenging. On the testbed, experiments show that the time at which the first packet of a train has been de-encapsulated and routed to the virtual NIC's receive queue occurs ∼20M cycles after the host NIC's device driver identifies the underlying packet arrival event. This time can vary, however, as illustrated in Figure 3.1, from 430K to 25M cycles.

There are three different cases to consider for EVI delivery:

1. **Virtual interrupts disabled:** If the virtual device driver has interrupts disabled when an early virtual interrupt is about to be raised, the EVI interrupt will not be delivered immediately. In this case, I *discard* the EVI interrupt as opposed to deferring its delivery, implicitly coalescing it with a later interrupt.

2. **Handler runs prior to packet availability:** If the guest packet handler runs prior to the packet being marked in the receive queue, the guest views the interrupt as invalid and ignores it, wasting guest OS time.

3. **Handler runs after packet availability:** If the guest handler runs significantly after the packet is available at the guest NIC (EVI was not performed early enough), latency increases compared to the native mode. Unoptimized VMM-level virtual overlay is the extreme version of this case, since the interrupt is not sent until the packet is in the virtual NIC.

Note that case (1) results in interrupts associated with packets not being delivered to the guests; optimistic interrupts handle this using the EoC notification mechanism described next.

4.3.2 End of Coalescing (EoC) notification

In physical hardware systems, masked and dropped interrupts are not generally significant problems because the host NIC will deliver another interrupt later upon the expiration of its interrupt coalescing timer. In addition, real hardware can set the length of this timer based on fine-grained information on the shape of the underlying traffic. To achieve the same effect with optimistic interrupts, I introduce an end-of-coalescing notification that the host NIC delivers to the overlay system when the host NIC has emptied its packet queue. This notification provides the virtual NIC an opportunity to make decisions about the potential termination of online traffic, as well as to recover from previous failed EVI injection attempts.

The virtual NIC handles EoC notifications based on the success or failure of the last EVI attempt and the shape of the traffic since the last EVI attempt. Specifically, if the last EVI attempt failed due to a masked interrupt, an EoC notification always results in the injection of a virtual interrupt, even if this interrupt delivery may be delayed until the guest unmasks interrupts.

If the previous EVI was successfully delivered, the virtual NIC must determine whether or not to inject a virtual interrupt. The specific case which EoC notification must guard against is when the guest has already stopped processing its receive queue, there will soon be additional packets in the receive queue to handle, and a new host interrupt (which would trigger an EVI) is unlikely to arrive soon. It does this by examining the host *receive density (RD)* (bytes received per second since the last EVI injection).

1. "Too cold": If $RD < \alpha$ the overlay system assumes that because the traffic has been sparse since the last EVI, the data that was received has probably already been retrieved by the guest device driver. Therefore, it does not inject a virtual interrupt. In other words, if the traffic has been light, then EoC

handling assumes delivery into the guest has already been done using the EVIs previously sent.

2. "Too hot": If $RD > \beta$ the overlay system assumes that because the traffic has been dense since the last EVI, it is probably in the middle of a stream of heavy traffic. In that case, EVIs are already being generated and driving the data transfer. Therefore, the EoC is discarded to avoid burdening the guest with an unnecessary interrupt.

3. "Just right:" If $\alpha \leq RD \leq \beta$, the system assumes that traffic density is high enough that the guest may not have processed all of it, but not high enough that a new EVI is likely to happen soon. Consequently, it raises a virtual interrupt so that this traffic is handled in a timely fashion.

The parameters α and β are experimentally determined.

4.3.3 EVI/EoC Interaction

Together, the EVI and EoC techniques that comprise the optimistic interrupt mechanism interact to overlap overlay packet processing with guest interrupt processing, and coalesce interrupts without the need for high-resolution timers. EVI's primary goal is to minimize the processing latency of packets received by the host NIC, particularly if the guest is not already processing packets. If the guest is already processing packets and interrupts are masked, however, the EVI is suppressed in favor of late interrupt injection at the EoC notification. The resulting implicit interrupt coalescing, driven by packet processing in the host OS and interrupt coalescing in the host NIC, reduces interrupt processing overheads in the guest.

Consider, for example, a virtual server on an overlay with a 9000 byte MTU that is being sent packets by a client. EVI will allow the server to immediately begin

processing of the first packet, even for a tiny packet, minimizing the first packet latency. When a train of large packets are sent to the host, however, EVI injection attempts that occur after the first packet will be deferred in favor of later delivery at EoC notifications due to masked interrupts, which are in turn driven by the rate at which the host can process packets, and the rate at which the host NIC coalesces interrupts and delivers bytes to the host. Finally, if guest packet processing after EVI injection outpaces overlay packet processing, the EoC-injected interrupt will assure that the guest processes the packets moved to the virtual NIC in a timely fashion.

4.4 Zero-copy Cut-through Data Forwarding

To increase the number of packets handled per interrupt and reduce the likelihood of guest packet processing outpacing overlay packet processing, I introduce a zero-copy cut-through data forwarding optimization. Building on the capabilities of modern NICs and the ability of the host OS to directly access guest memory, this optimization directly forwards incoming and outgoing packets between the the guest virtual NIC and the host NIC. This reduces overlay per-packet processing costs by avoiding data copies between the guest and host NIC, and page flipping costs associated with other zero-copy techniques.

Zero-copy cut-through transmission. On the transmit side, the overlay system delivers a virtual NIC's outgoing packet as a scatter/gather abstraction. This allows the overlay system to encapsulate guest packets simply by adding the appropriate UDP, IP, and Ethernet headers to the scatter-gather list without copying the guest packet. This expanded scatter/gather list can then be handed directly to the host NIC for packet transmission. All copies between the guest's buffer and a host buffer are avoided.

Implicit zero-copy reception and cut-through forwarding. On the receive side, the host NIC receives incoming packets, including the encapsulating headers, directly into buffers provided *by the guest*, without the need for page flipping or data copies. Note that this makes the overlay's encapsulation visible to the guest's virtio device driver. The guest is responsible for stripping encapsulation headers from incoming packets. This is enabled by the virtio NIC implementation exporting the length of the encapsulation header to the guest driver as a new port in to the PCI configuration space. If the encapsulation header length changes, the VMM simply raises an interrupt that notifies the NIC of configuration space changes.

4.5 Noise Isolation

To reduce variation in throughput and latency, I also target OS noise. Specifically, I adopt a *lightweight kernel* as the host OS into which the VMM is embedded. In addition to directly reducing network performance variability, this optimization also increases the effectiveness of optimistic interrupt by providing more predictable system timing and scheduling behavior. This latter benefit could also be provided in heavyweight OSes like Linux, however, by using well-known techniques for isolating virtual machines and processes on individual cores.

Even in a mainstream cloud environment, the use of a lightweight kernel is not as radical as it may seem. In essence, in the system, the combination of the VMM and a lightweight kernel provides the model of a traditional "Type I" VMM. High performance VMMs, for example VMware ESXi, adopt the same model.

4.6 Virtual TCP Offload

To reduce the semantic gap between virtual Ethernet overlay and underlying advanced heterogeneous interconnects, I supplement the virtual Ethernet NIC exported to the guest by the overlay with Virtual TCP Offload Engine (VTOE) capabilities.

4.6.1 Overview

TCP Offload Engines (TOEs) Ethernet devices offload the processing of the entire kernel TCP/IP stack to the network controller. It is primarily used with high-speed network interfaces, such as 10 Gbps Ethernet, where processing overhead of the network stack becomes significant [8]. Most modern operating systems support TCP offload engines.

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 lowers virtualization overhead and achieves better network performance. For connections that cross networks that do not guarantee reliable transport, the overlay itself provides reliability using TCP; existing overlays may 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.

4.6.2 Virtual TCP Offload in VNET/P

Figure 4.2 shows the overall architecture of VNET/P supplemented with a Virtual TCP Offload Engine, which I denote as VNET+VTOE. In this system, guests run in

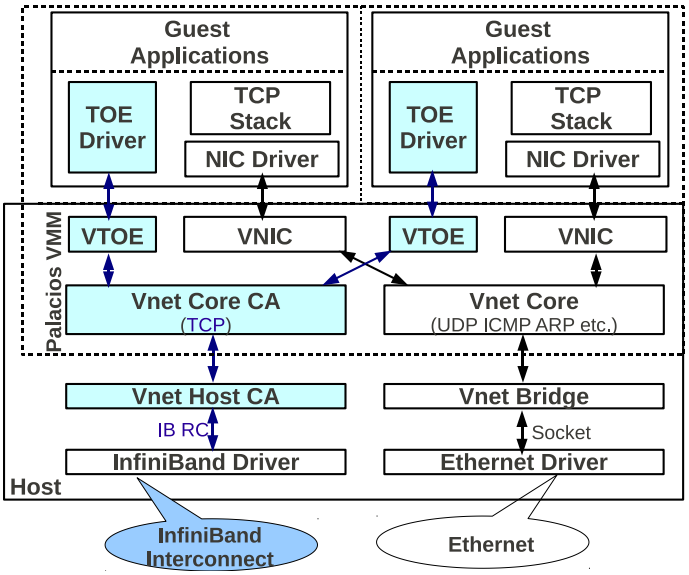


Figure 4.2: VNET+VTOE architecture with Linux VM and Palacios VMM.

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 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.

4.6.3 VTOE NIC Architecture

A VTOE NIC must support two main classes of operations between the guest and the overlay in the virtual machine monitor. First, the virtual NIC and overlay must manage stateful connection establishment and teardown, mapping guest TCP connection management requests to underlying network requests. Second, the VTOE system must map guest TCP data transmission and reception actions to underlying overlay actions.

Like most TOE NICS, the VTOE NIC architecture includes four key structures, a *send work queue*, a *receive work queue*, an *event queue*, and a set of *control registers*. and an VTOE has a *send work queue (SWQ)* to accept outgoing application data to hand to the overlay, a *receive work queue (RWQ)* to buffer data from the overlay before the data is dispatched to individual applications, an *event queue (EQ)* to be notified of asynchronous events from the overlay, and a set of *control registers (CRs)* to issue commands to the overlay.

In order for the guest to execute an operation, it must place a virtual work queue element (VWQE) in the virtual work queue. From there the operation is picked up for execution by the VTOE. Therefore, the virtual work queue forms the communications medium between the guest and the VMM.

The event queue in the VTOE is used for the overlay to deliver shadow connection status and overlay status to the guests. The shadow connection status is driven by the transport services of the underlying advanced interconnects. The set of events include CONNECT_REQUEST, CONNECT_RESPONSE, ADDRESS_RESOLVED, ROUTE_RESOLVED, ADDRESS_ERROR, ROUTE_ERROR, CONNECT_ERROR, UNREACHABLE, CONNECT_REJECTED, CONNECT_ESTABLISHED, DISCONNECTED, and DEVICE_REMOVAL.

4.7 Summary

In this chapter, I answer the performance challenges in deploying virtual overlay networks, as presented in Chapter 3, by moving virtual overlay networks into VMMs, by overlapping virtual interrupt emulation and overlay packet processing, by increasing the number of packets delivered per virtual interrupt, by coalescing virtual interrupts without any problematic timers, and by reducing the semantic gap between virtual networks features and underlying physical network features.

Chapter 5

Implementation

I now describe the implementation of the proposed optimizations in the VNET/P overlay implementation, in the context of Palacios as embedded in Linux and Kitten hosts.

5.1 VNET/P

5.1.1 Architecture

Figure 5.1 shows the overall architecture of VNET/P, and illustrates the operation of VNET/P in the context of the Palacios VMM embedded in a Linux host. In this architecture, *guests* run in *application VMs*. Off-the-shelf guests are fully supported. Each application VM provides a virtual (Ethernet) NIC to its guest. 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. The virtual NIC conveys Ethernet packets between the application VM and the Palacios VMM. Using the virtio virtual NIC, one or more packets can be conveyed from an ap-

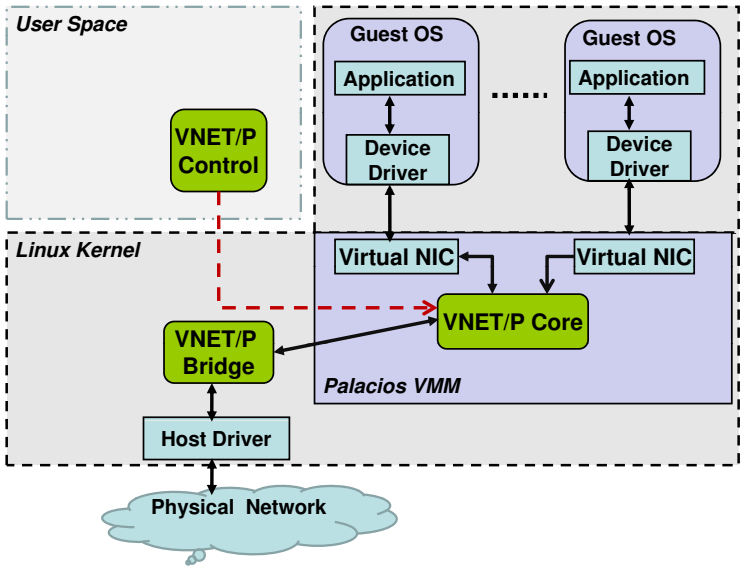


Figure 5.1: VNET/P architecture

plication VM to Palacios with a single VM exit, and from Palacios to the application VM with a single VM exit+entry.

The *VNET/P core* is the component of VNET/P that is directly embedded into the Palacios VMM. It is responsible for routing Ethernet packets between virtual NICs on the machine and between the local machine and remote VNET instances on other machines. The VNET/P core's routing rules are dynamically configurable, through the control interface run in user space.

The VNET/P core also provides an expanded interface that the control utilities can use to configure and manage VNET/P. The *VNET/P control* component uses this interface to do so. It in turn acts as a daemon that exposes a TCP control port that uses the same configuration language as VNET/U. Between compatible encapsulation and compatible control, the intent is that VNET/P and VNET/U be

interoperable, with VNET/P providing the "fast path".

To exchange packets with a remote machine, the VNET/P core uses a *VNET/P bridge* to communicate with the physical network. The VNET/P bridge runs as a kernel module in the host kernel and uses the host's networking facilities to interact with physical network devices and with the host's networking stack. An additional responsibility of the bridge is to provide encapsulation. For performance reasons, we use UDP encapsulation in a form compatible with that used in VNET/U. TCP encapsulation is also supported. The bridge selectively performs UDP or TCP encapsulation for packets destined for remote machines, but can also deliver an Ethernet packet without encapsulation. In our performance evaluation, we consider only encapsulated traffic.

The VNET/P core consists of approximately 2500 lines of C in Palacios, while the VNET/P bridge consists of about 2000 lines of C comprising a Linux kernel module. VNET/P is available via the V3VEE project's public git repository, as part of the "devel" branch of the Palacios VMM.

5.1.2 VNET/P core

The VNET/P core is primarily responsible for routing and dispatching raw Ethernet packets. It intercepts all Ethernet packets from virtual NICs that are associated with VNET/P, and forwards them either to VMs on the same host machine or to the outside network through the VNET/P bridge. Each packet is routed based on its source and destination MAC addresses. The internal processing logic of the VNET/P core is illustrated in Figure 5.2.

Routing: To route Ethernet packets, VNET/P maintains routing tables indexed by source and destination MAC addresses. Although this table structure only provides linear time lookups, a hash table-based routing cache is layered on top of the

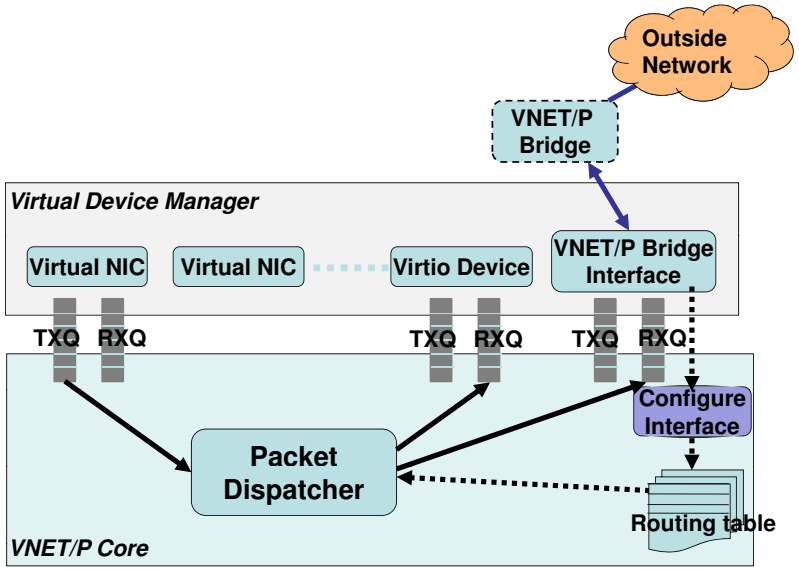


Figure 5.2: VNET/P core's internal logic.

table, and the common case is for lookups to hit in the cache and thus be serviced in constant time.

A routing table entry maps to a destination, which is either a *link* or an *interface*. A link is an overlay destination—it is the next UDP/IP-level (i.e., IP address and port) destination of the packet, on some other machine. A special link corresponds to the local network. The local network destination is usually used at the "exit/entry point" where the VNET overlay is attached to the user's physical LAN. A packet routed via a link is delivered to another VNET/P core, a VNET/U daemon, or the local network. An interface is a local destination for the packet, corresponding to some virtual NIC.

For an interface destination, the VNET/P core directly delivers the packet to the relevant virtual NIC. For a link destination, it injects the packet into the VNET/P

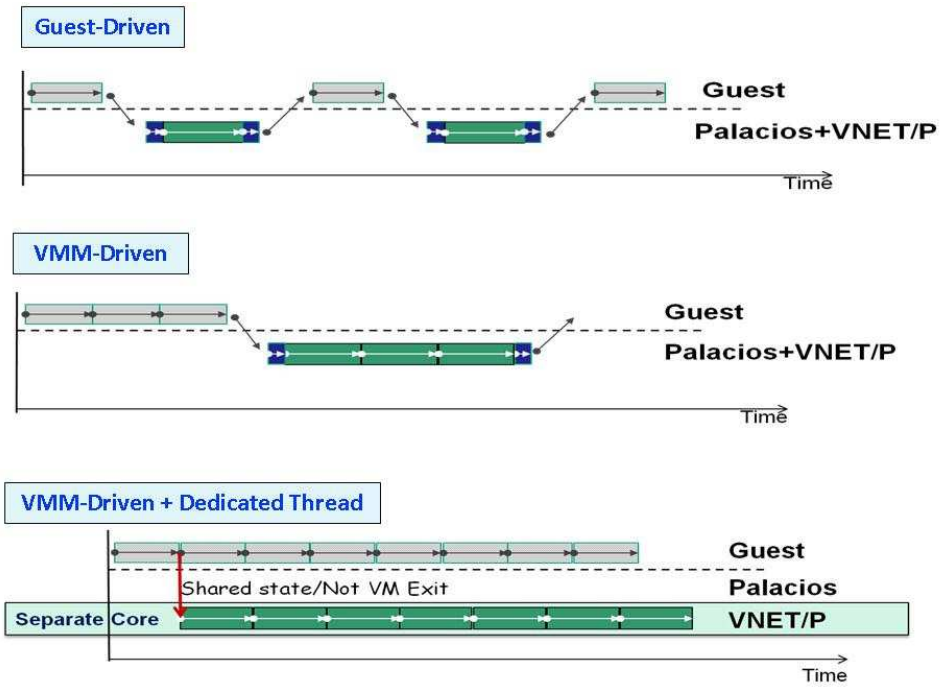


Figure 5.3: The VMM-driven and guest-driven modes in the virtual NIC. Guest-driven mode can decrease latency for small messages, while VMM-driven mode can increase throughput for large messages. Combining VMM-driven mode with a dedicated packet dispatcher thread results in most send-related exits caused by the virtual NIC being eliminated, thus further enhancing throughput.

bridge along with the destination link identifier. The VNET/P bridge demultiplexes based on the link and either encapsulates the packet and sends it via the corresponding UDP or TCP socket, or sends it directly as a raw packet to the local network.

Packet processing: Packet forwarding in the VNET/P core is conducted by *packet dispatchers*. A packet dispatcher interacts with each virtual NIC to forward packets in one of two modes: *guest-driven mode* or *VMM-driven mode*. The operation of these modes is illustrated in the top two timelines in Figure 5.3.

The purpose of guest-driven mode is to minimize latency for small messages in a parallel application. For example, a barrier operation would be best served with

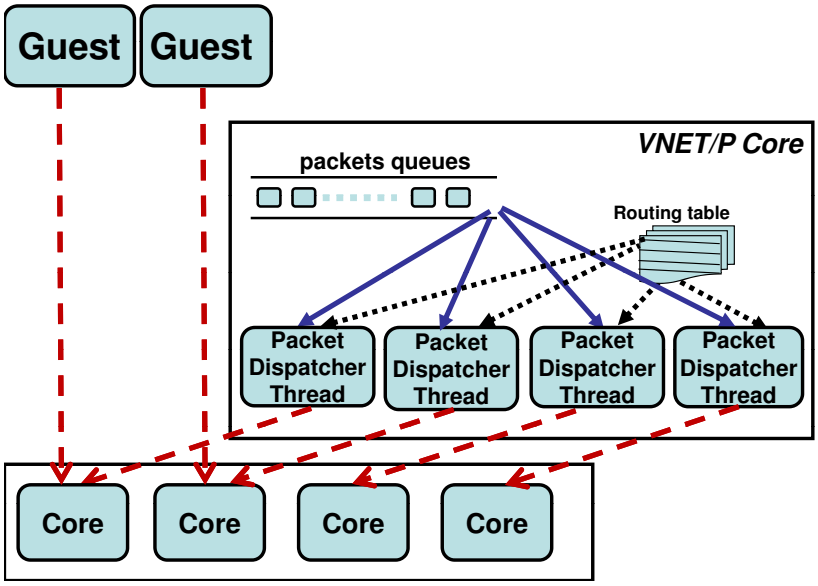


Figure 5.4: VNET/P running on a multicore system. The selection of how many, and which cores to use for packet dispatcher threads is made dynamically.

guest-driven mode. In the guest-driven mode, the packet dispatcher is invoked when the guest's interaction with the NIC explicitly causes an exit. For example, the guest might queue a packet on its virtual NIC and then cause an exit to notify the VMM that a packet is ready. In guest-driven mode, a packet dispatcher runs at this point. Similarly, on receive, a packet dispatcher queues the packet to the device and then immediately notifies the device.

The purpose of VMM-driven mode is to maximize throughput for bulk data transfer in a parallel application. Unlike guest-driven mode, VMM-driven mode tries to handle multiple packets per VM exit. It does this by having the VMM poll the virtual NIC. The NIC is polled in two ways. First, it is polled, and a packet dispatcher is run, if needed, in the context of the current VM exit (which is

unrelated to the NIC). Even if exits are infrequent, the polling and dispatch will still make progress during the handling of timer interrupt exits.

The second manner in which the NIC can be polled is in the context of a packet dispatcher running in a kernel thread inside the VMM context, as shown in Figure 5.4. The packet dispatcher thread can be instantiated multiple times, with these threads running on different cores in the machine. If a packet dispatcher thread decides that a virtual NIC queue is full, it forces the NIC's VM to handle it by doing a cross-core IPI to force the core on which the VM is running to exit. The exit handler then does the needed event injection. Using this approach, it is possible to dynamically employ idle processor cores to increase packet forwarding bandwidth. Combined with VMM-driven mode, VNET/P packet processing can then proceed in parallel with guest packet sends, as shown in the bottommost timeline of Figure 5.3.

Influenced by Sidecore [46], an additional optimization we developed was to offload in-VMM VNET/P processing, beyond packet dispatch, to an unused core or cores. This makes it possible for the guest VM to have full use of its cores (minus the exit/entry costs when packets are actually handed to/from it).

VNET/P can be configured to statically use either VMM-driven or guest-driven mode, or *adaptive operation* can be selected. In adaptive operation, which is illustrated in Figure 5.5, VNET/P switches between these two modes dynamically depending on the arrival rate of packets destined to or from the virtual NIC. For a low rate, it enables guest-driven mode to reduce the single packet latency. On the other hand, with a high arrival rate it switches to VMM-driven mode to increase throughput. Specifically, the VMM detects whether the system is experiencing a high exit rate due to virtual NIC accesses. It recalculates the rate periodically. The algorithm employs simple hysteresis, with the rate bound for switching from guest-driven to VMM-driven mode being larger than the rate bound for switching back. This avoids rapid switching back and forth when the rate falls between these bounds.

```
num_packets = {number of exits caused by virtual NIC
                  from last period of time window}

rate = num_packets/window_time

if (rate > upper_rate_limit && current_mode == Guest-Driven)

    then:
                switch_mode (VMM-Driven)
                current-mode = VMM-Driven

else if (rate < lower_rate_limit && current-mode == VMM-Driven)

    then:
                switch_mode (Guest-Driven)
                current-mode = Gust-Driven

else

    do-nothing

endif
```

Figure 5.5: Adaptive mode dynamically selects between VMM-driven and guest-driven modes of operation to optimize for both throughput and latency.

5.1.3 Virtual NICs

VNET/P is designed to be able to support any virtual Ethernet NIC device. A virtual NIC must, however, register itself with VNET/P before it can be used. This is done during the initialization of the virtual NIC at VM configuration time. The registration provides additional callback functions for packet transmission, transmit queue polling, and packet reception. These functions essentially allow the NIC to use VNET/P as its backend, instead of using an actual hardware device driver backend.

Linux virtio virtual NIC: Virtio [63], which was recently developed for the Linux kernel, provides an efficient abstraction for VMMs. A common set of virtio device drivers are now included as standard in the Linux kernel. To maximize per-

formance, our performance evaluation configured the application VM with Palacios's virtio-compatible virtual NIC, using the default Linux virtio network driver.

MTU: The maximum transmission unit (MTU) of a networking layer is the size of the largest protocol data unit that the layer can pass onwards. A larger MTU improves throughput because each packet carries more user data while protocol headers have a fixed size. A larger MTU also means that fewer packets need to be processed to transfer a given amount of data. Where per-packet processing costs are significant, larger MTUs are preferable. Because VNET/P adds to the per-packet processing cost, supporting large MTUs is helpful.

VNET/P presents an Ethernet abstraction to the application VM. The most common Ethernet MTU is 1500 bytes. However, 1 Gbit and 10 Gbit Ethernet can also use "jumbo frames", with an MTU of 9000 bytes. Other networking technologies support even larger MTUs. To leverage the large MTUs of underlying physical NICs, VNET/P itself supports MTU sizes of up to 64 KB.[1] The application OS can determine the virtual NIC's MTU and then transmit/receive accordingly.

The MTU used by virtual NIC can result in encapsulated VNET/P packets that exceed the MTU of the underlying physical network. In this case, fragmentation has to occur, either in the VNET/P bridge or in the host NIC (via TCP Segmentation Offloading (TSO)). Fragmentation and reassembly is handled by VNET/P and is totally transparent to the application VM. However, performance will suffer when significant fragmentation occurs. Thus it is important that the application VM's device driver select an MTU carefully, and recognize that the desirable MTU may change over time, for example after a migration to a different host. In Chapter 6, we analyze throughput using different MTUs.

[1]This may be expanded in the future. Currently, it has been sized to support the largest possible IPv4 packet size.

5.1.4 VNET/P Bridge

The VNET/P bridge functions as a network bridge to direct packets between the VNET/P core and the physical network through the host NIC. It operates based on the routing decisions made by the VNET/P core which are passed along with the packets to be forwarded. It is implemented as a kernel module running in the host.

When the VNET/P core hands a packet and routing directive up to the bridge, one of two transmission modes will occur, depending on the destination. In a *direct send*, the Ethernet packet is directly sent. This is common for when a packet is exiting a VNET overlay and entering the physical network, as typically happens on the user's network. It may also be useful when all VMs will remain on a common layer 2 network for their lifetime. In an *encapsulated send* the packet is encapsulated in a UDP packet and the UDP packet is sent to the directed destination IP address and port. This is the common case for traversing a VNET overlay link.

Similarly, for packet reception, the bridge uses two modes, simultaneously. In a *direct receive* the host NIC is run in promiscuous mode, and packets with destination MAC addresses corresponding to those requested by the VNET/P core are handed over to it. This is used in conjunction with direct send. In an *encapsulated receive* UDP packets bound for the common VNET link port are disassembled and their encapsulated Ethernet packets are delivered to the VNET/P core. This is used in conjunction with encapsulated send. Our performance evaluation focuses solely on encapsulated send and receive.

5.1.5 Control

The VNET/P control component allows for remote and local configuration of links, interfaces, and routing rules so that an overlay can be constructed and changed over

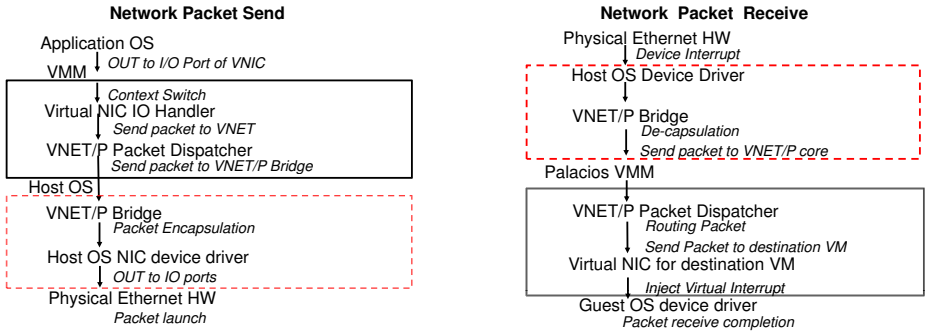


Figure 5.6: Performance-critical data paths and flows for packet transmission and reception. Solid boxed steps and components occur within the VMM itself, while dashed boxed steps and components occur in the host OS.

time. VNET/U already has user-level tools to support VNET, and, as we described in Section 2.3.3, a range of work already exists on the configuration, monitoring, and control of a VNET overlay. In VNET/P, we reuse these tools as much as possible by having the user-space view of VNET/P conform closely to that of VNET/U. The *VNET/P configuration console* allows for local control to be provided from a file, or remote control via TCP-connected VNET/U clients (such as tools that automatically configure a topology that is appropriate for the given communication pattern among a set of VMs [73]). In both cases, the VNET/P control component is also responsible for validity checking before it transfers the new configuration to the VNET/P core.

5.1.6 Performance-critical data paths and flows

Figure 5.6 depicts how the components previously described operate during packet transmission and reception. These are the performance critical data paths and flows within VNET/P, assuming that virtio virtual NICs (Section 5.1.3) are used. The boxed regions of the figure indicate steps introduced by virtualization, both within the VMM and within the host OS kernel. There are also additional overheads in-

volved in the VM exit handling for I/O port reads and writes and for interrupt injection.

Transmission: The guest OS in the VM includes the device driver for the virtual NIC. The driver initiates packet transmission by writing to a specific virtual I/O port after it puts the packet into the NIC's shared ring buffer (TXQ). The I/O port write causes an exit that gives control to the virtual NIC I/O handler in Palacios. The handler reads the packet from the buffer and writes it to VNET/P packet dispatcher. The dispatcher does a routing table lookup to determine the packet's destination. For a packet destined for a VM on some other host, the packet dispatcher puts the packet into the receive buffer of the VNET/P bridge and notify it. Meanwhile, VNET/P bridge fetches the packet from the receive buffer, determines its destination VNET/P bridge, encapsulates the packet, and transmits it to the physical network via the host's NIC.

Note that while the packet is handed off multiple times, it is copied only once inside the VMM, from the send buffer (TXQ) to the receive buffer of the VNET/P bridge. Also note that while the above description and the diagram suggest sequentiality, packet dispatch can occur on a separate kernel thread running on a separate core, and the VNET/P bridge itself introduces additional concurrency. From the guest's perspective, the I/O port write that initiated transmission returns essentially within a VM exit/entry time.

Reception: The path for packet reception is essentially symmetric to that of transmission. The host NIC in the host machine receives a packet using its standard driver and delivers it to the VNET/P bridge. The bridge unencapsulates the packet and sends the payload (the raw Ethernet packet) to the VNET/P core. The packet dispatcher in VNET/P core determines its destination VM and puts the packet into the receive buffer (RXQ) of its virtual NIC.

Similar to transmission, there is considerably concurrency in the reception process. In particular, packet dispatch can occur in parallel with the reception of the next packet.

5.1.7 Performance tuning parameters

VNET/P is configured with the following parameters:

- Whether guest-driven, VMM-driven, or adaptive mode is used.
- For adaptive operation, the upper and lower rate bounds (α_l, α_u) for switching between guest-driven, and VMM-driven modes, as well as the window size ω over which rates are computed.
- The number of packet dispatcher threads $n_{dispatchers}$ that are instantiated.
- The yield model and parameters for the bridge thread, the packet dispatch threads, and the VMM's halt handler.

The last of these items requires some explanation, as it presents an important tradeoff between VNET/P latency and CPU consumption. In both the case of packets arriving from the network and packets arriving from the guest's virtual NIC, there is, conceptually, a thread running a receive operation that could block or poll. First, consider packet reception from the physical network. Suppose a bridge thread is waiting for UDP packet to arrive. If no packets have arrived recently, then blocking would allow the core on which the thread is running to be yielded to another thread, or for the core to be halted. This would be the ideal for minimizing CPU consumption, but on the next arrival, there will be a delay, even if the core is idle, in handling the packet since the thread will need to be scheduled. There is a similar tradeoff in the packet dispatcher when it comes to packet transmission from the guest.

A related tradeoff, for packet reception in the guest, exists in the VMM's model for handling the halt state. If the guest is idle, it will issue an HLT or related instruction. The VMM's handler for this case consists of waiting until an interrupt needs to be injected into the guest. If it polls, it will be able to respond as quickly as possible, thus minimizing the packet reception latency, while if it blocks, it will consume less CPU.

For all three of these cases VNET/P's and the VMM's *yield strategy* comes into play. Conceptually, these cases are written as polling loops, which in their loop bodies can yield the core to another thread, and optionally put the thread to sleep pending a wake-up after a signaled event or the passage of an interval of time. Palacios currently has selectable yield strategy that is used in these loops, and the strategy has three different options, one of which is chosen when the VM is configured:

- Immediate yield. Here, if there is no work, we immediately yield the core to any competing threads. However, if there are no other active threads that the core can run, the yield immediately returns. A poll loop using the immediate yield has the lowest latency while still being fair to competing threads.

- Timed yield: Here, if there is no work, we put the thread on a wake up queue and yield CPU. Failing any other event, the thread will be awakened by the passage of time T_{sleep}. A poll loop using the timed yield strategy minimizes CPU usage, but at the cost of increased latency.

- Adaptive yield: Here, the poll loop reports how much time has passed since it last did any work. Until that time exceeds a threshold T_{nowork}, the immediate yield strategy is used, and afterwards the timed yield strategy is used. By setting the threshold, different tradeoffs between latency and CPU usage are made.

In our performance evaluations, we use the following parameters to focus on the performance limits of VNET/P:

Parameter	Value
mode	adaptive
α_l	10^3 packets/second
α_u	10^4 packets/second
ω	5ms
$n_{dispatchers}$	1
yield strategy	immediate yield
T_{sleep}	not used
T_{nowork}	not used

5.2 VNET/P+

To understand the impact of the optimizations described in Sections 4.3, 4.4 and 4.5, we implemented them in the VNET/P overlay network previously described in Section 5.1. I denote VNET/P enhanced with the optimizations as VNET/P+.

VNET/P+ includes a new implementation of the VNET/P bridge for Kitten that includes custom UDP encapsulation (Kitten does not currently include general TCP/IP networking support), and extends VNET/P with three more components which are used to implement the optimistic interrupt and cut-through forwarding optimizations:

1. The **device allocator** maps host NICs to virtual NICs and maintains device allocation tables to support EVI and EoC notification routing.

2. The **memory allocator** controls direct memory access (DMA) from the host NIC to the virtual NICs' memory to support zero-copy cut-through forwarding.

3. The **event dispatcher** handles virtual interrupt and event delivery to virtual NICs for both EVI injection and EoC notification.

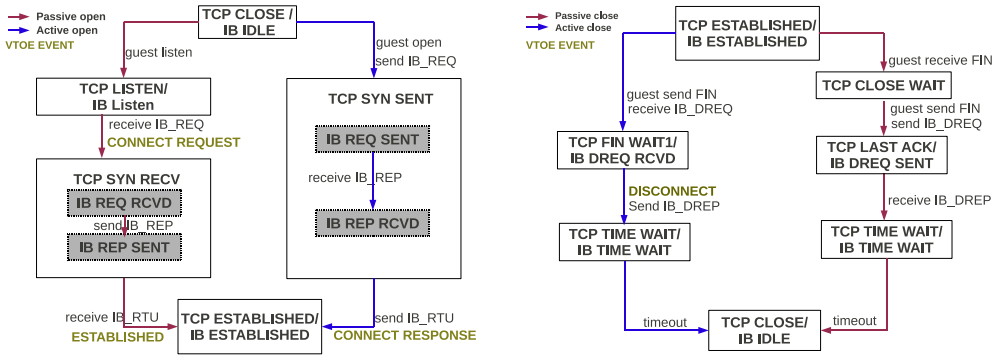
The complete implementation of VNET/P+ in Kitten and Palacios, including the the reimplementation of the VNET/P bridge for Kitten comprises approximately 10,000 source lines of code, of which approximately 2,000 are changes to support the optimizations described above.

5.3 VNET/P+VTOE

Our initial implementation of VTOE support in VNET/P has focused on Infiniband networks, though 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.

5.3.1 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, 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.



(a) Connection establishment (b) Connection termination

Figure 5.7: State machines for both frontend VNET sockets and shadow CIDs, during connection establishment and termination.

Connection Management

For connection management, the VNET Core Connection Agent (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 Connection Agent (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 5.7(a) shows the TCP state machine of all VNET socket and the IB state of the shadow connection during connection establishment. The right half of figure Figure 5.7(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 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 then transitions to IB_ESTABLISHED state. It then notifies the guest of the response to its connection request. The guest then transitions the guest socket to the TCP_ESTABLISHED state.

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 5.7(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 Connection ID (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 it receives the disconnection request from the passive-closing host, the active-closing shadow CID 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.

Data transfer

Transmission with zero overlay copy: 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.

Again, 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 copy: 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 copy, 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.

5.3.2 Interfacing virtual TCP offload 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 implementation. In particular, we use the Infiniband SDP approach to dynamically change the application address family into AF_INET_VNET by using a preloaded 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 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, then all calls on the socket are redirected to the Linux socket chain. If a VNET socket is required, then 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.

5.4 Summary

In this chapter, I described the implemention of the proposed optimizations, as presented in Chapter 4. The VNET/P overlay system is embedded in the Palacios VMM, which comprises three components – a virtual NIC for each guest, a VMM extension handling packets routing and interacting with virtual NICs, and a host bridge interfacing physical NIC and remote systems. Optimistic interrupts and cut-through forwarding are implemented in the VNET/P system. The virtual TCP offload implementation is focused on InfiniBand interconnect, and includes a VTOE NIC driver and VNET socket module in the guest. Inside the VMM and host, the core connection agent and host connection agent are responsible for mapping between guest TCP connections and host IB connections.

Chapter 6

Evaluation

The purposes of the performance evaluation are: 1) to determine how close VNET/P comes to native throughput and latency in the most demanding (lowest latency, highest throughput) hardware environments; and 2) to determine how efficient VNET/P+ and VTOE improve VNET/P performance. This dissertation considers communication between machines whose NICs are directly connected in most of the detailed benchmarks. In the virtualized configuration the guests and performance testing tools run on top of Palacios with VNET/P carrying all traffic between them using encapsulation. In the native configuration, the same guest environments run directly on the hardware.

The evaluation of communication performance in this environment occurs at three levels. First, I benchmark TCP and UDP bandwidth and latency. Second, I benchmark MPI using a widely used benchmark. Finally, I evaluated the performance of the HPCC and NAS application benchmarks in a cluster to see the impact of VNET/P and the proposed optimizations on the performance and scalability of parallel applications.

6.1 Experimental Setup

6.1.1 Testbed

The testbed, consists of 6 physical machines each of which has dual quad core 2.3 GHz 2376 AMD Opteron "Shanghai" processors (8 cores total), 32 GB RAM, an nVidia MCP55 Forthdeth 1 Gbps Ethernet NIC, and a NetEffect NE020 10 Gbps Ethernet fiber optic NIC (10GBASE-SR) in a PCI-e slot. The 10 Gbps Ethernet NICs are connected through a Fujitsu XG2000 series 10GigE switch.

Besides the Ethernet fabric, each node also has a Mellanox MT26428 InfiniBand NIC in a PCI-e slot. The Infiniband NICs were connected via a Mellanox MTS 3600 36-port 20/40Gbps InfiniBand switch.

6.1.2 Configurations

A range of measurements are made using the cycle counter. DVFS control is disabled on the machine's BIOS and in the host Linux kernel. I also sanity-checked the measurement of larger spans of time by comparing cycle counter-based timing with a separate wall clock.

All microbenchmarks included in the performance section are run in the testbed described above. The HPCC and NAS application benchmarks are run on a 6-node test cluster described in Section 6.2.6.

I considered the following several software configurations:

- *Native:* In the native configuration, neither Palacios nor VNET/P is used. A minimal BusyBox-based Linux environment based on an unmodified 2.6.30 kernel runs directly on the host machines. There are several sub-configurations:

  - **Native-1G/10G:** Native performance on 1 and 10 Gbps Ethernet, respectively.
  - **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.

- *Passthrough:* I use the passthrough configuration to measure the performance overheads of the virtualized environment without VNET/P. Palacios runs on each host machine. On each host machine, a single VM is launched with with the same Linux environment as in Native, with the exception that the kernel has a ∼20 line modification to support passthrough device address mapping, as explained in Section 5.1. The VM, which is configured with a single core and 1 GB of RAM, has direct access to the Ethernet devices via Palacios's PCI passthrough mechanism. I refer to the 1 and 10 Gbps results in this configuration as *Passthrough-1G* and *Passthrough-10G*, respectively.

- *VNET/P:* The VNET/P configuration corresponds to the architectural diagram given in Figure 5.1, with a single guest VM running on Palacios. The guest VM is configured with one virtio network device, 4 cores, and 1 GB of RAM. The guest VM runs a minimal BusyBox-based Linux environment, based on the 2.6.30 kernel. The kernel used in the VM is identical to that in the Native configuration, with the exception that the virtio NIC drivers are loaded. The virtio MTU is configured as 9000 Bytes.

  There are several sub-configurations:

  - **VNET-1G/10G:** VMM-level VNET performance on 1 and 10 Gbps Ethernet, respectively.

  - **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.

- *VNET/P+:* The VNET/P+ configuration corresponds to the optimizations described in Sections 4.3, 4.4, and 4.5. This configuration is similar to the VNET/P configuration, except that VNET/P+ is hooked with the Kitten lightweight kernel instead of Linux as the host OS. Unless otherwise specified, the virtio NIC provided to the guest was configured to use 9000 byte MTUs.

To assure accurate time measurements both natively and in the virtualized case, the guest is configured to use the CPU's cycle counter, and Palacios is configured to allow the guest direct access to the underlying hardware cycle counter. The 1 Gbps NIC only supports MTUs up to 1500 bytes, while the 10 Gbps NIC can support MTUs of up to 9000 bytes. I use these maximum sizes unless otherwise specified. For Native+IPoIB and VNET+IPoIB configurations, MTUs are set to 65520.

6.2 Micro-benchmarks

I used simple two-node ICMP, UDP, TCP, and MPI benchmarks to provide an initial characterization of the impact of the proposed optimizations. UDP throughput and goodput were measured using Iperf-2.0.4 [3] with 8900 byte writes for 150 seconds, while TCP throughput was measured using *ttcp-1.10*. For simple MPI tests, I used the Intel MPI Benchmark Suite (IMB 3.2.2) [37] running on OpenMPI 1.3 [25], focusing on the point-to-point messaging performance.

6.2.1 TCP and UDP microbenchmarks

Latency and throughput are the fundamental measurements I use to evaluate VNET/P system performance. First, I consider these at the IP level, measuring the round-trip latency, the UDP goodput, and the TCP throughput between two nodes. I measure round-trip latency using *ping* by sending ICMP packets of different sizes. UDP and TCP throughput are measured using *ttcp-1.10*.

VMM-level Overlay

UDP and TCP with a standard MTU: Figure 6.1 shows the TCP throughput and UDP goodput achieved in each of the configurations on each NIC. For the 1 Gbps network, host MTU is set to 1500 bytes, and for the 10 Gbps network, host MTUs of 1500 bytes and 9000 bytes are both tested. For 1 Gbps, VNET/U is also compared running on the same hardware with Palacios. Compared to previously reported results (21.5 MB/s, 1 ms), the combination of the faster hardware used here, and Palacios, leads to VNET/U increasing its bandwidth by 330%, to 71 MB/s, with a 12% reduction in latency, to 0.88 ms.

To measure VNET/P, I begin by considering UDP goodput when a standard host MTU size is used. For UDP measurements, ttcp was configured to use 64000 byte writes sent as fast as possible over 60 seconds. For the 1 Gbps network, VNET/P easily matches the native goodput. For the 10 Gbps network, VNET/P achieves 74% of the native UDP goodput. For TCP throughput, ttcp was configured to use a 256 KB socket buffer, and to communicate 40 MB writes were made. Similar to the UDP results, VNET/P has no difficulty achieving native throughput on the 1 Gbps network. On the 10 Gbps network, using a standard Ethernet MTU, it achieves 78% of the native throughput. The UDP goodput and TCP throughput that VNET/P is capable of, using a standard Ethernet MTU, are approximately 8 times those from

VNET/U given the 1 Gbps results.

UDP and TCP with a large MTU: I next consider TCP and UDP performance with 9000 byte jumbo frames the 10 Gbps NICs support. The VNET/P MTU was adjusted so that the final encapsulated packets will fit into these frames without fragmentation. For TCP, ttcp is configured to use writes of corresponding size, maximize the socket buffer size, and do 4 million writes. For UDP, ttcp is configured to use commensurately large packets sent as fast as possible for 60 seconds. The results are also shown in the Figure 6.1. Notice that performance increases across the board compared to the 1500 byte MTU results. Compared to the VNET/U performance in this configuration, the UDP goodput and TCP throughput of VNET/P are over 10 times higher.

VNET/P+

Figure 6.3 shows that VNET/P+ achieves 90% of the native UDP goodput (1.3 times higher than VNET/P), and 94% of the native TCP throughput (1.5 times higher than VNET/P).

InfiniBand

I used simple two-node TCP and MPI benchmarks to provide an initial characterization of the impact of the proposed VTOE infrastructure. TCP throughput was measured using *ttcp-1.10*. For simple MPI tests, I used the Intel MPI Benchmark Suite (IMB 3.2.2) [37] running on OpenMPI 1.3 [25], focusing on the point-to-point messaging performance. For each test case, I ran 10 times and report the average as the result.

TCP Uni-stream Bandwidth:

Figure 6.4(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 9.4Gbps, 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.

In terms of CPU usage, VNET+VTOE has 76% receive-side usage compared to 99% for VNET+IPoIB. This reduced CPU usage comes from eliminating the virtual interrupts overhead caused by ACKs in VNET+IPoIB. On the transmit side, VNET+IPoIB has less CPU usage than VNET+VTOE because the higher VNET+IPoIB receiving CPU utilization slows down the sender by having the sender waiting for ACKs.

TCP Bi-Stream Bandwidth:

In addition to unidirectional TCP performance, I also examined bi-stream bandwidth performance to measure the duplex capability of VNET+VTOE. In this test, I use two machines and 2 threads on each machine. Each thread on one machine has a connection to exactly one thread on the other machine. Thus 2 connections are established between these two machines. On each connection, the basic TTCP bandwidth test is performed. The throughput and CPU usage are shown in Figure 6.4(b).

Native+SDP shows good duplex performance, delivering 10 Gbps bandwidth for each stream. In contrast, Native+IPoIB bottlenecks on bi-directional data transfer, with each stream dropping to half of the InfiniBand 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.

CPU Utilization Discussion:

The CPU utilization is also presented for each test configuration. 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.

The benchmark reveals that Native+SDP and VNET+VTOE can not only achieve high aggregated bandwidth, but 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.

There are two noticeable facts from the benchmark results:

- Receive-side CPU utilization in virtualization mode drops compared to native, while transmit-side CPU utilization increases; and
- In both native cases, receive-side CPU utilization is more than transmit CPU utilization, while in the virtualization cases, transmit-side CPU usage is higher than receive side.

In the native cases, the real physical link is fast enough to keep the receive-side CPU busy with incoming packets and the flow control and the slow rate of generating ACKs have the sender slow down. So in the native cases, the network performance is bound to the receiving side CPU utilization, and how fast the receiver generates ACKs. While in the virtualized networks, the virtual link provided by the overlay is slower due to virtulization overhead, reducing load on the receiver. As a result,

the receiver is not too busy to handle incoming packets and most of the time has available buffers for more incoming packets. Because of this, the flow control keeps the sender sending data as fast as it can and so the network performance is bound to the overlay virtual link data-transfer rate.

6.2.2 ICMP Latency

VMM-level VNET

Figure 6.2 shows the ICMP round-trip latency for different packet sizes, as measured by ping. The latencies are the average of 100 measurements. While the increase in latency of VNET/P over Native is significant in relative terms (2x for 1 Gbps, 3x for 10 Gbps), it is important to keep in mind the absolute performance. On a 10 Gbps network, VNET/P achieves a 130 μs round-trip, end-to-end latency. The latency of VNET/P is almost seven times lower than that of VNET/U.

VNET/P+

Figure 6.5 shows the round-trip latency for different packet sizes as measured by ping. The latencies are the average of 100 measurements. The latency of VNET/P+ is less than half that of VNET/P, and approaches passthrough latency. Passthrough itself is limited due to the need for interrupt exiting and reinjection.

6.2.3 Network Performance Variability

In addition to ICMP, UDP, and TCP performance, I also examined ICMP and TCP performance variability. To test latency variation, I use *ping* with 64-byte messages for 5000 iterations. To test throughput variation, I examine variation in Iperf [3]

performance with 8900 byte sends over the course of an hour.

Figure 6.6 shows the results of VNET/P and VNET/P+ latency and bandwidth variability experiments. VNET/P shows large latency bursts every few hundred of iterations, while VNET/P+ shows substantially less latency variation. Likewise, VNET/P+ demonstrates lower throughput variation than VNET/P.

6.2.4 MPI microbenchmarks

Parallel programs for distributed memory computers are typically written to the MPI interface standard. I used the OpenMPI 1.3 [25] implementation in the evaluations. I measured the performance of MPI over VNET/P by employing the widely-used Intel MPI Benchmark Suite (IMB 3.2.2) [37], focusing on the point-to-point messaging performance. I compared the basic MPI latency and bandwidth achieved by VNET/P, VNET/P+, Passthrough, VTOE, and natively.

VMM-level VNET

Figures 6.7 and 6.8(a) illustrate the latency and bandwidth reported by Intel MPI PingPong benchmark for the 10 Gbps configuration. Here the latency measured is the one-way, end-to-end, application-level latency. That is, it is the time from when an MPI send starts on one machine to when its matching MPI receive call completes on the other machine. For both Native and VNET/P, the host MTU is set to 9000 bytes.

VNET/P's small message MPI latency is about 55 μs, about 2.5 times worse than the native case. However, as the message size increases, the latency difference decreases. The measurements of end-to-end bandwidth as a function of message size show that native MPI bandwidth is slightly lower than raw UDP or TCP throughput,

and VNET/P performance tracks it similarly. The bottom line is that the VMM-level VNET/P implementation can deliver an MPI latency of 55 μs and bandwidth of 510 MB/s on 10 Gbps Ethernet hardware.

Figure 6.8(b) shows the results of the MPI SendRecv microbenchmark in which each node simultaneously sends and receives. There is no reduction in performance between the bidirectional case and the unidirectional case. For both figures, the absolute numbers are shown, but it is important to note that the overhead is quite stable in percentage terms once the message size is large: beyond 256K, the one-way bandwidth of VNET/P is around 74% of native, and the two-way bandwidth is around 62% of native.

VNET/P+

As shown in Figure 6.9, MPI point-to-point performance with VNET/P+ is equal to the passthrough performance, and approaches the native performance for both small and large messages.

InfiniBand

Figure 6.10 shows the IMB MPI point-to-point performance with VNET+VTOE. For small messages, VNET+VTOE has more than 2 times lower latency that VNET-+IPoIB, but 2 times higher latency 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.

6.2.5 Understanding Low-level Behavior

In VNET/P+, I used the previously discussed benchmarks to better understand the fine-grained behavior and performance impacts of optimistic interrupts and zero-copy cut through forwarding. I found that, during high-bandwidth packet reception, the combination of EVI and EoC notifications results in 1 to 1.5 virtual interrupts being injected into the guest for every physical interrupt raised by the (coalescing) host NIC. Only 0.5% of EVI injections are premature, limiting the impact of the guest discarding premature interrupt as invalid. In addition, around 10% of EVIs failed due to masked interrupts by guests.

The results also found that cut-though forwarding was important for improving the performance of VNET/P, but *only* when used in conjunction with optimistic interrupts. The results show that zero-copy cut-through forwarding without optimistic interrupts results in less than a 3% improvement in throughput and no improvement in latency. When optimistic interrupts are also enabled, in contrast, cut-through forwarding results in throughput improvements of 10%, although no improvement in small message latencies.

6.2.6 HPCC Latency-bandwidth benchmarks on more nodes

To test VNET/P performance on more nodes, I ran the HPCC benchmark [36] suite on a 6 node cluster with 1 Gbps and 10 Gbps Ethernet, and InfiniBand NICs. The VMs were all configured exactly as in previous tests, with 4 virtual cores, 1 GB RAM, and a virtio NIC. For the VNET/P, VNET/P+, VTOE and Passthrough test cases, each host ran one VM. I executed tests with 2, 3, 4, 5, and 6 VMs, with 4 HPCC processes per VM (one per virtual core). Thus, the performance results are based on HPCC with 8, 12, 16, 20 and 24 processes for all of VNET/P, VNET/P+, VTOE, Passthrough, and Native tests. In the native cases, no VMs were used, and

the processes ran directly on the host. For 1 Gbps testing, the host MTU was set to 1500, while for the 10 Gbps cases, the host MTU was set to 9000 for Ethernet, and 65520 for InfiniBand IPoIB configuration.

This benchmark consists of the ping-pong test and the ring-based tests. The ping-pong test measures the latency and bandwidth between all distinct pairs of processes. The ring-based tests arrange the processes into 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 message volume divided by the number of processes and the maximum time needed in all processes. I reported the ring-based bandwidths by multiplying them with the number of processes in the test.

1 Gbps Ethernet

Figure 6.11 shows the results for different numbers of test processes. The ping-pong latency and bandwidth results are consistent with what can be seen in the previous microbenchmarks: in the 1 Gbps network, bandwidth are nearly identical to those in the native cases while latencies are 1.2–2 times higher. Both latency and bandwidth under VNET/P exhibit the same good scaling behavior of the native case.

10 Gbps Ethernet

Figure 6.12 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 what can be seen in the previous microbenchmarks: in

VNET/P+, bandwidths are within 90% of native, and latencies are about 1.3 times that of native latencies. In VNET/P, bandwidths are within 60–70% of native, and latencies are about 2.5–3 times that of native latencies. The results show that the optimistic interrupts and cut-through forwarding optimizations can substantially enhance the performance of a software-based VMM-level overlay virtual network like VNET/P.

InfiniBand

Figure 6.13 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 what can be seen in 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 the VTOE can substantially enhance the performance of a software-based VMM-level overlay virtual network like VNET/P on InfiniBand.

6.3 Application Benchmarks

I evaluated the effect of a VNET/P overlay and the proposed optimizations on application performance by running at least two HPCC application benchmarks and the whole NAS benchmark suite on the cluster described in Section 6.2.6. Overall, the performance results from the HPCC and NAS benchmarks suggest that VNET/P, VNET/P+, and VTOE can achieve high performance for many parallel applications.

6.3.1 HPCC application benchmarks

I considered the three application benchmarks from the HPCC suite that exhibit the largest volume and complexity of communication: MPIRandomAcceess, PTRANS, and MPIFFT. For 1 Gbps networks, the difference in performance is negligible so I focus here on 10 Gbps Ethernet and InfiniBand interconnects.

10 Gbps Ethernet

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 6.14(a) shows the results of MPIRandomAccess, comparing the VNET/P+, VNET/P, Passthrough, and Native cases. VNET/P+ achieves 87% of native performance, while VNET/P achieves 60-65% application performance compared to the native cases.

PTRANS does a parallel matrix transpose, exercising simultaneous communications between pairs of processors. The performance is measured in the total communication capacity (GB/s) of the network. Figure 6.14(b) shows the result of PTRANS for the VNET/P+, VNET/P, Passthrough, and Native cases. VNET/P+ achieves 100% of the native performance, while VNET/P achieves 60–70% of the of native case.

MPIFFT implements a double precision complex one-dimensional Discrete Fourier Transform (DFT). Its performance is measured in Gflop/s. Figure 6.14(c) shows the result of MPIFFT for the VNET/P+, VNET/P, Passthrough, and Native cases. VNET/P+ again achieves 100% of native performance, while the VNET/P achieves only 60-70%. These results suggest that the impact of the optimizations to the VNET/P networking system are likely to strongly be felt in application codes.

InfiniBand

Figure 6.15(a) shows the results of MPIRandomAccess, 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+Uverbs 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.

Figure 6.15(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.

Figure 6.15(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.

Discussion

As shown in the evaluation results, VTOE significantly improves bandwidth and reduces CPU utilization for bandwidth-intensive codes compared with VNET+IPoIB. For large messages and throughput-sensitive applications, VTOE outperforms Native+IPoIB. However, small-message latency in VNET+VTOE is still high, about twice Native+IPoIB latency, although it has been improved compared with that in

VNET+IPoIB by more than 50%. 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, it is noticeable the high latency has negative impacts on the overall performance. I expect that the optimistic interrupt techniques will reduce this overhead, but have not yet implemented these techniques in VNET+VTOE.

Considering the tradeoff between CPU overhead and network performance, 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. This is often justified in high performance environments in which fast message completion is critical to overall application performance

6.3.2 NAS parallel benchmarks

The NAS Parallel Benchmark (NPB) suite [76] is a set of five kernels and three pseudo-applications that is widely used in parallel performance evaluation. I specifically use NPB-MPI 2.4 in the evaluation. In this description, executionsn are named with the format "name.class.procs". For example, *bt.B.16* means to run the BT benchmark on 16 processes with a class B problem size.

I run each benchmark with at least two different scales and one problem size, except FT, which is only run with 16 processes. One VM is run on each physical machine, and it is configured as described in Section 6.2.6. The test cases with 8 processes are running within 2 VMs and 4 processes started in each VM. The test cases with 9 processes are run with 4 VMs and 2 or 3 processes per VM. Test cases

Mop/s	Native-1G	VNET/P-1G	$\frac{VNET/P-1G}{Native-1G}(\%)$
ep.B.8	103.15	101.94	98.8%
ep.B.16	204.88	203.9	99.5%
ep.C.8	103.12	102.1	99.0%
ep.C.16	206.24	204.14	99.0%
mg.B.8	4400.52	3840.47	87.3%
mg.B.16	1506.77	1498.65	99.5%
cg.B.8	1542.79	1319.43	85.5%
cg.B.16	160.64	159.69	99.4%
ft.B.16	1575.83	1290.78	81.9%
is.B.8	78.88	74.61	94.6%
is.B.16	35.99	35.78	99.4%
is.C.8	89.54	82.15	91.7%
is.C.16	84.76	82.22	97.0%
lu.B.8	6818.52	5495.23	80.6%
lu.B.16	7847.99	6694.12	85.3%
sp.B.9	1361.38	1215.85	89.3%
sp.B.16	1489.32	1399.6	94.0%
bt.B.9	3423.52	3297.04	96.3%
bt.B.16	4599.38	4348.99	94.6%

Table 6.1: NAS Parallel Benchmark performance with VNET/P on 1 Gbps Ethernet. VNET/P can achieve native performance on many applications, while it can get reasonable and scalable performance when supporting highly communication-intensive parallel application workloads.

with 16 processes have 4 VMs with 4 processes per VM. I report each benchmark's *Mop/s total* result for all of native, Passthrough, VNET/P, and VNET/P+ on both 1 Gbps and 10 Gbps Ethernet.

1 Gbps Ethernet

Table 6.1 shows the NPB performance results, comparing the VNET/P and Native cases on both 1 Gbpsnetwork. The upshot of the results is that for most of the NAS

benchmarks, VNET/P is able to achieve in excess of 95% of the native performance even on 1 Gbps networks. I now describe the results for each benchmark.

EP is an "embarrassingly parallel" kernel that estimates the upper achievable limits for floating point performance. It does not require a significant interprocessor communication. VNET/P achieves native performance in all cases.

MG is a simplified multigrid kernel that requires highly structured long distance communication and tests both short and long distance data communication. With 16 processes, MG achieves native performance on the 1 Gbps network.

CG implements the conjugate gradient method to compute an approximation to the smallest eigenvalue of a large sparse symmetric positive definite matrix. It is typical of unstructured grid computations in that it tests irregular long distance communication, employing unstructured matrix vector multiplication. With 16 processes, CG achieves native performance on the 1 Gbps network.

FT implements the solution of partial differential equations using FFTs, and captures the essence of many spectral codes. It is a rigorous test of long-distance communication performance. With 16 nodes, it achieves 82% of native performance on 1 Gbps.

IS implements a large integer sort of the kind that is important in particle method codes and tests both integer computation speed and communication performance. Here VNET/P achieves native performance in all cases.

LU solves a regular-sparse, block (5×5) lower and upper triangular system, a problem associated with implicit computational fluid dynamics algorithms. VNET/P achieves 75%-85% of native performance on this benchmark.

SP and BT implement solutions of multiple, independent systems of non diagonally dominant, scalar, pentadiagonal equations, also common in computational fluid

Mop/s	Native	Passthrough	VNET/P	**VNET/P+**	$\frac{Passthrough}{Native}$	$\frac{VNET/P}{Native}$	$\frac{VNET/P+}{Native}$
ep.B.8	102.18	102.17	102.12	**102.12**	99.9%	99.9%	**99.9%**
ep.B.16	208	207.96	206.25	**207.93**	99.9%	99.3%	**99.9%**
ep.C.8	103.13	102.76	102.14	**103.08**	99.6%	99%	**99.9%**
ep.C.16	206.22	205.39	203.98	**204.98**	99.6%	98.9%	**99.4%**
mg.B.8	5110.29	4662.53	3796.03	**4643.67**	91.2%	74.3%	**90.9%**
mg.B.16	9137.26	8384.93	7405	**8262.08**	91.8%	81%	**90.4%**
cg.B.8	2096.64	1824.05	1806.57	**1811.14**	87%	86.2%	**86.4%**
cg.B.16	592.08	592.05	554.91	**592.07**	99.9%	93.7%	**99.9%**
ft.B.8	2055.435	2055.4	1562.1	**2055.3**	99.9%	76.2%	**99.9%**
ft.B.16	1432.3	1432.2	1228.39	**1432.18**	99.9%	85.7%	**99.9%**
is.B.8	59.15	59.14	59.04	**59.13**	99.9%	99.8%	**99.9%**
is.B.16	23.09	23.05	23	**23.04**	99.8%	99.6%	**99.8%**
is.C.8	132.08	132	131.87	**132.04**	99.9%	99.8%	**99.9%**
is.C.16	77.77	77.12	76.94	**77.1**	99.9%	98.9%	**99.9%**
lu.B.8	7173.65	6730.23	6021.78	**6837.06**	93.8%	83.9%	**95.3%**
lu.B.16	12981.86	11630.65	9643.21	**12198.65**	89.6%	74.3%	**94%**
sp.B.9	2634.53	2634.5	2421.98	**2634.5**	99.9%	91.9%	**99.9%**
sp.B.16	3010.71	3009.5	2916.81	**2954.16**	99.9%	96.8%	**98.1%**
bt.B.9	5229.01	4750.4	4076.52	**4798.63**	90.8%	78.0%	**91.8%**
bt.B.16	6315.11	6314.1	6105.11	**6242.83**	99.9%	96.7%	**99%**

Table 6.2: NAS performance on VNET/P, VNET/P+, Native, and Passthrough configurations. The optimizations implemented in VNET/P+ can help us achieve full native performance on almost all of the benchmarks.

dynamics. The salient difference between the two is the communication to computation ratio. For SP with 16 processes, VNET/P achieves 94% of native performance on 1 Gbps.For BT at the same scale, 95% of native at 1 Gbps.

10 Gbps Ethernet

Table 6.2 shows the NPB performance results, comparing the VNET/P+, VNET/P, Passthrough, and Native cases. The optimizations implemented in VNET/P+ make it possible to achieve native performance in a number of cases where the unoptimized

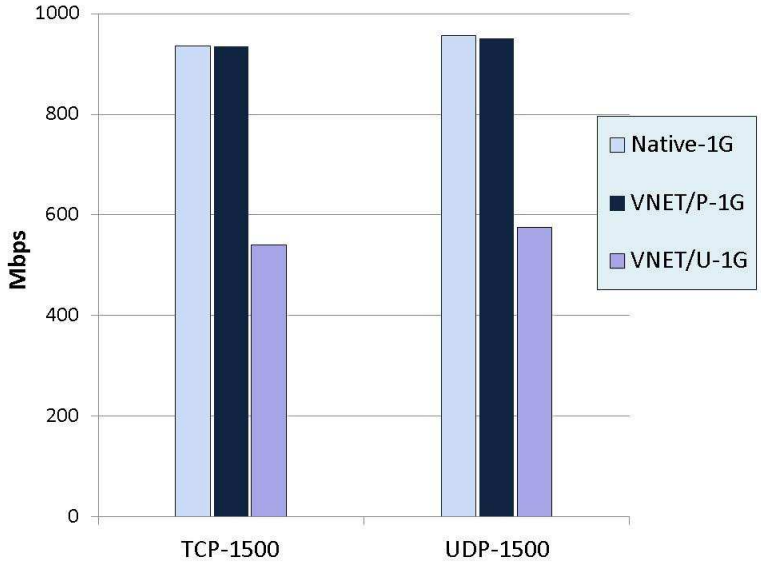
VNET/P was unable to, particularly for MG, FT, LU, cg.B.16, and bt.B.9.

In a few cases, VNET/P+ did not achieve full native performance. In particular, VNET/P+ achieves passthrough levels of performance but only 87% of native in the case of cg.B.8. Similarly, VNET/P+ achieves 91% of native performance in bt.B.9, while VNET/P only delivers 78% of native. The performance differences at smaller scales between VNET/P+ and passthrough virtualized cases compared to native are due to basic interrupt and memory virtualization overheads. These overheads are comparatively smaller at larger node counts, where a greater fraction of application time is spent on communication.

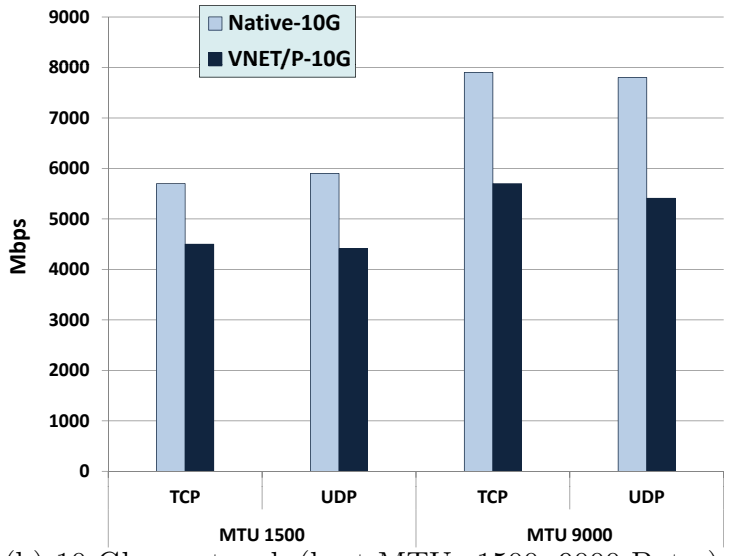
The results of the evaluations on NPB strongly suggest that the optimizations implemented in VNET/P+ make it possible for a software-based overlay virtual network to provide native performance for communication-intensive applications on 10 Gbps networks.

6.4 Summary

In this chapter, I evaluate the impacts of the proposed virtual network optimizations, as presented in Chapter 4 and Chapter 5 for a wide range of micro-benchmarks and MPI application bench-marks. the evaluation results show that the proposed optimizations can dramatically improve virtual network performance on 1 Gbps Ethernet, 10 Gbps Ethernet, and InfiniBand interconnects. Specifically, these optimization cut virtual overlay latency in half, improve throughput by more than 2.5 times, and achieve near-native MPI application performance.

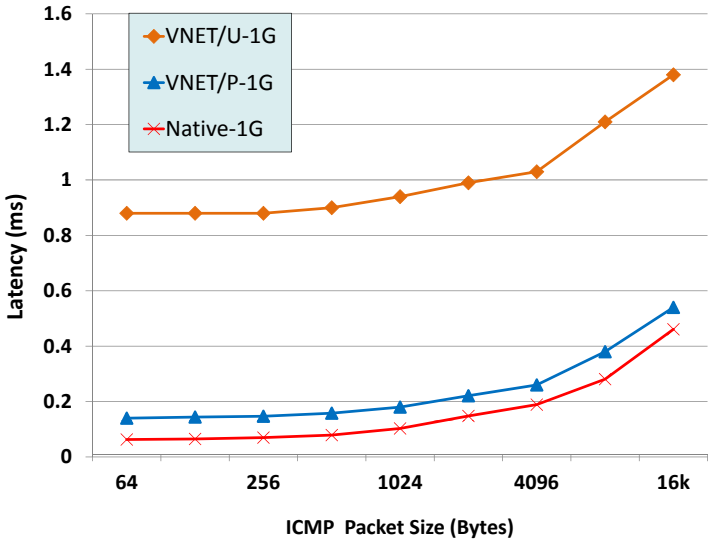


(a) 1 Gbps network (host MTU=1500 Bytes)

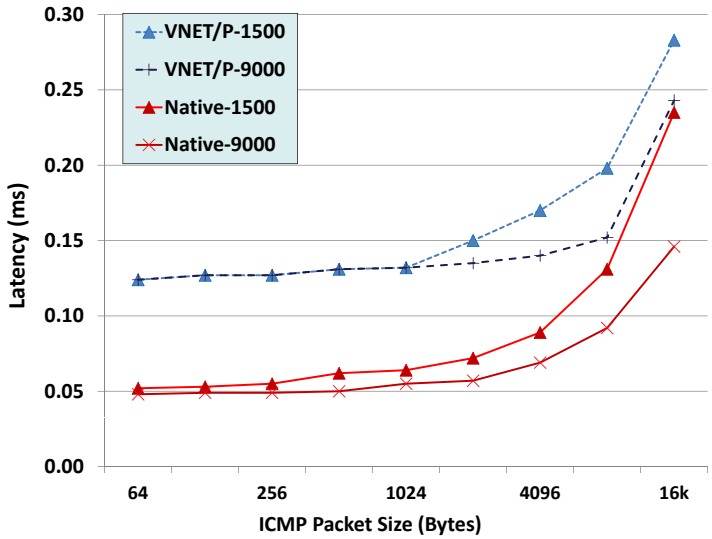


(b) 10 Gbps network (host MTU=1500, 9000 Bytes)

Figure 6.1: End-to-end TCP throughput and UDP goodput of VNET/P on 1 Gbps network. VNET/P performs identically to the native case for the 1 Gbps network and achieves 74–78% of native throughput for the 10 Gbps network.



(a) 1 Gbps network (Host MTU=1500 Bytes)



(b) 10 Gbps network (Host MTU=1500, 9000 Bytes)

Figure 6.2: End-to-end round-trip latency of VNET/P as a function of ICMP packet size. Small packet latencies on a 10 Gbps network in VNET/P are ∼130 μs.

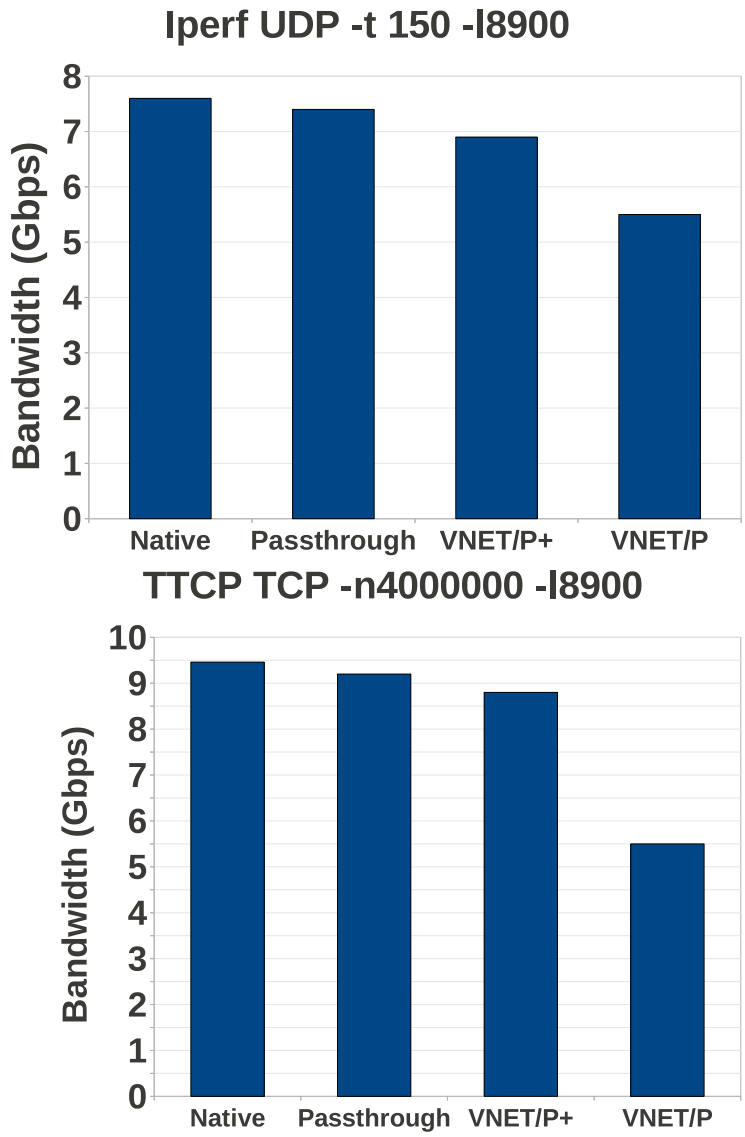
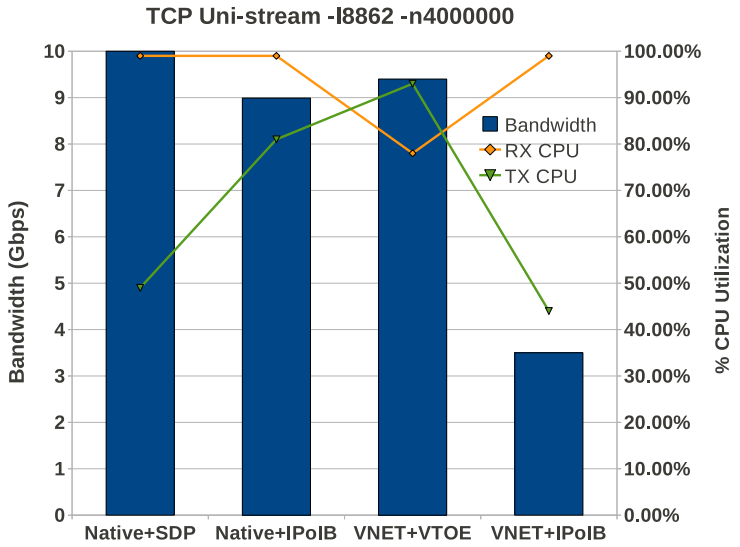
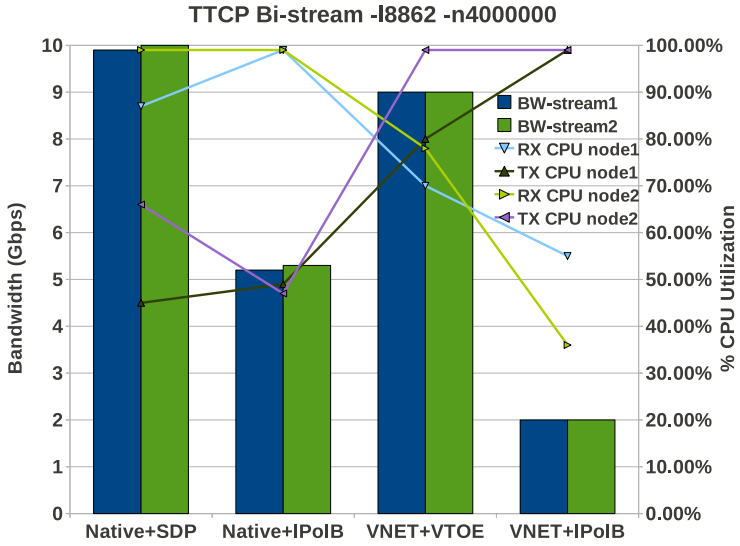


Figure 6.3: End-to-end UDP goodput and TCP throughput of VNET/P+ and VNET/P on 10 Gbps network. VNET/P+ performs better then VNET/P for the 10 Gbps network



(a) TCP Uni-stream Bandwidth



(b) TCP Bi-stream Bandwidth

Figure 6.4: 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

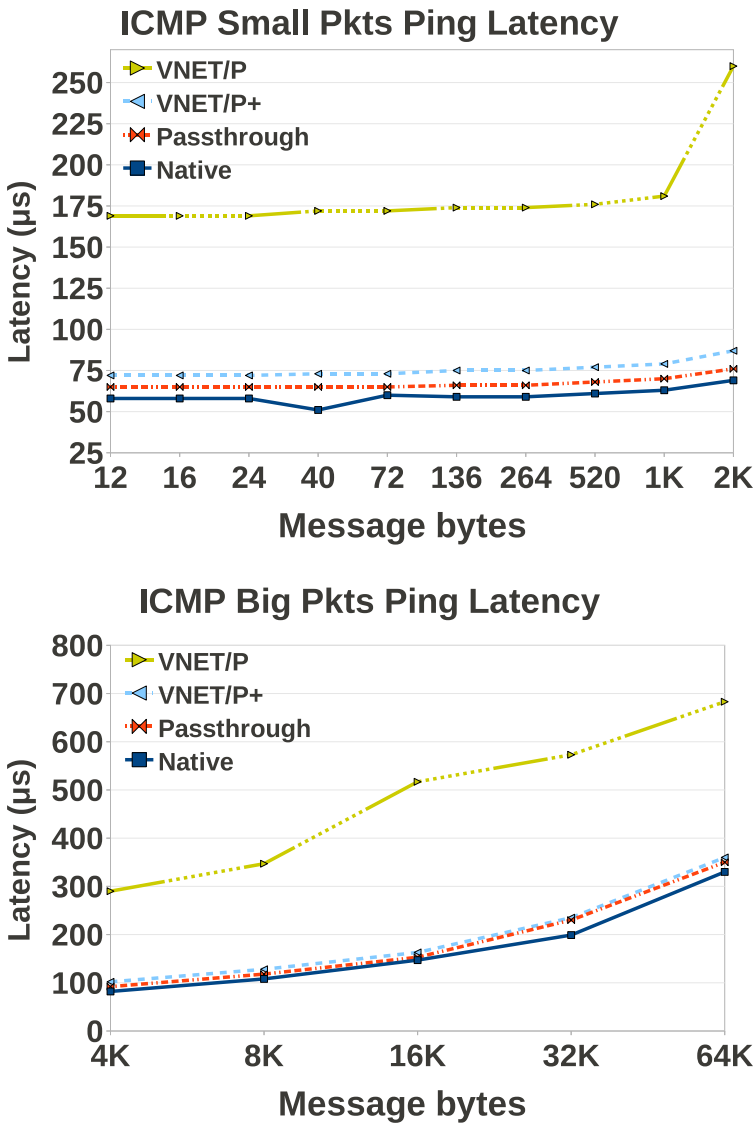
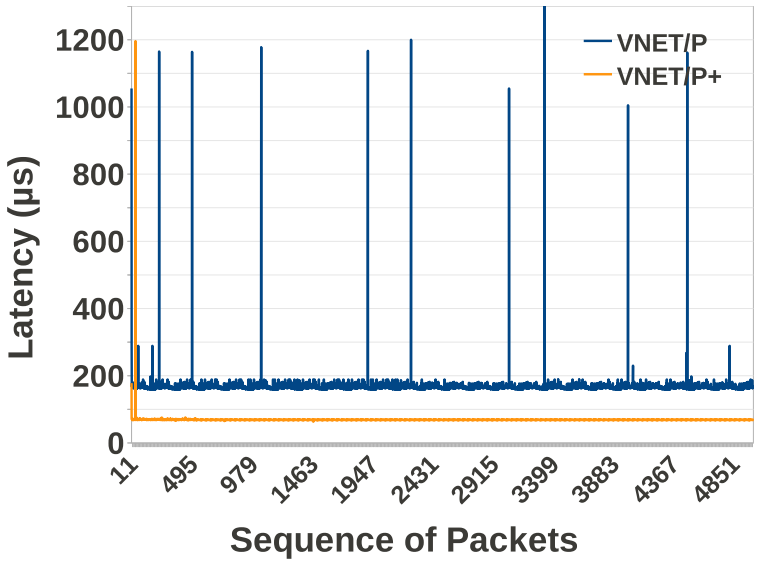
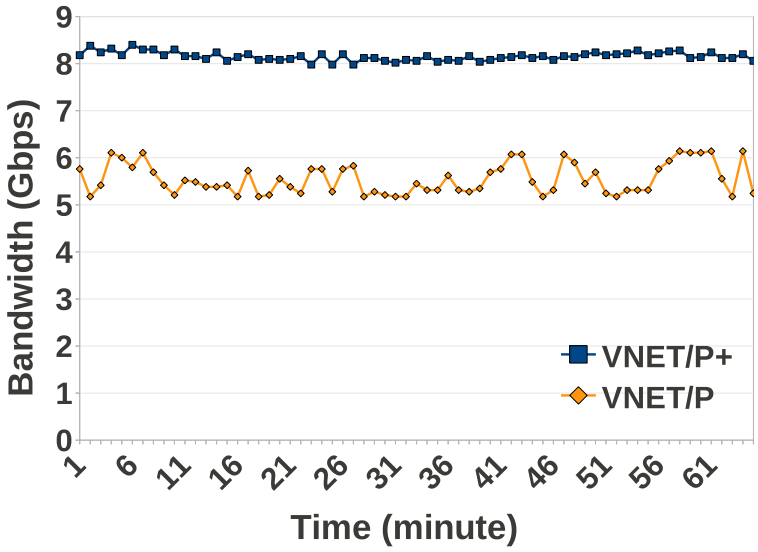


Figure 6.5: End-to-end round-trip latency of VNET as a function of ICMP packet size. Small packet latencies are: VNET/P+—72μs, Passthrough—65μs, Native—58μs, VNET/P—169μs.



(a) 5000 iterations of 64 bytes latency variation



(b) 1 hour TCP throughput variation

Figure 6.6: 64-byte packets ICMP latency and TCP throughput variation results on 10 Gbps Ethernet. VNET/P+ shows near-zero variation except the first two probing packets, while VNET/P has large latency bursts. VNET/P+ also shows less variation of TCP throughput.

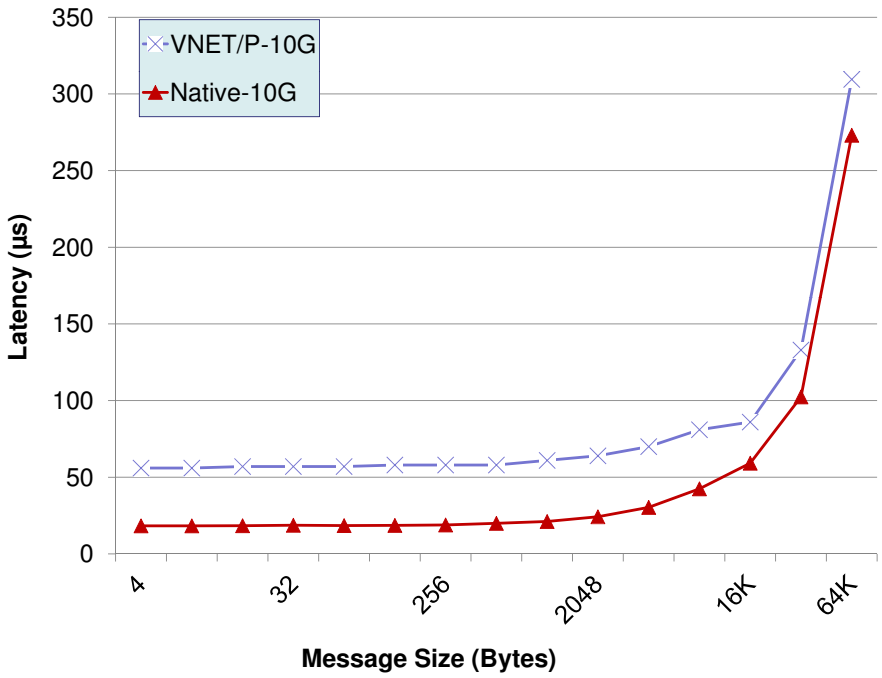
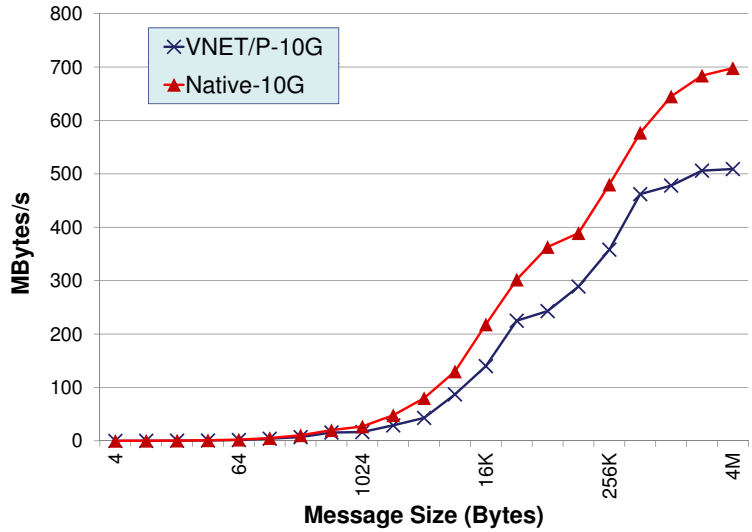
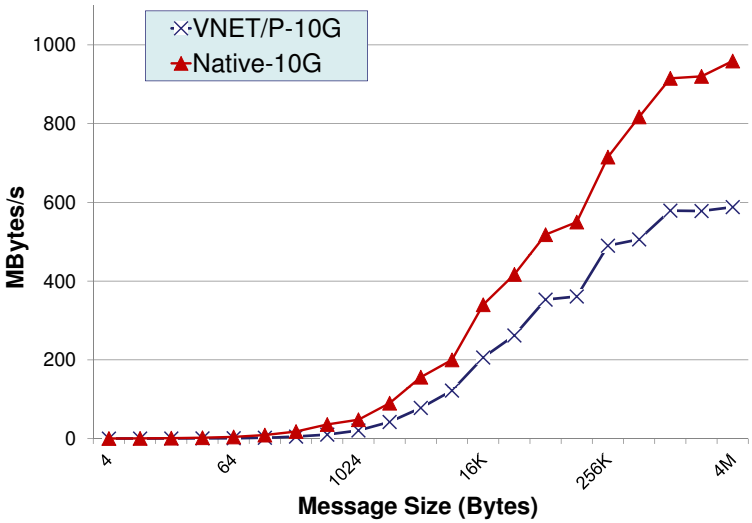


Figure 6.7: One-way latency on 10 Gbps hardware from Intel MPI PingPong microbenchmark



(a) One-way bandwidth



(b) SendRecv Bandwidth

Figure 6.8: Intel MPI PingPong microbenchmark showing (a) one-way bandwidth and (b) bidirectional bandwidth as a function of message size on the 10 Gbps hardware.

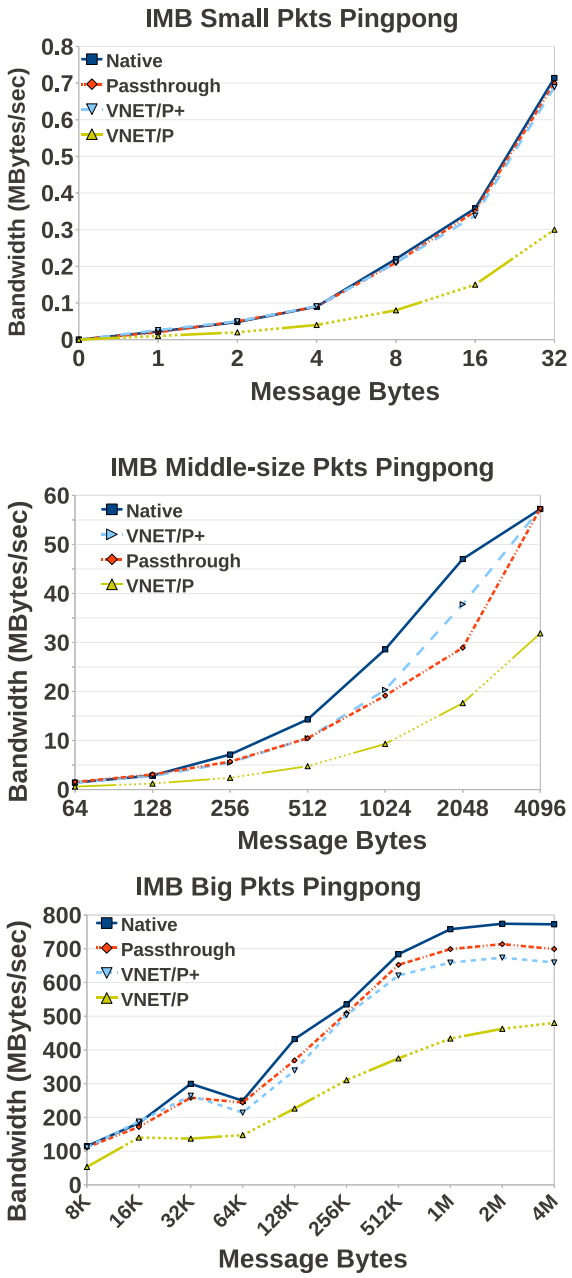


Figure 6.9: Intel MPI PingPong microbenchmark showing bidirectional bandwidth as a function of message size on the 10Gbps Ethernet.

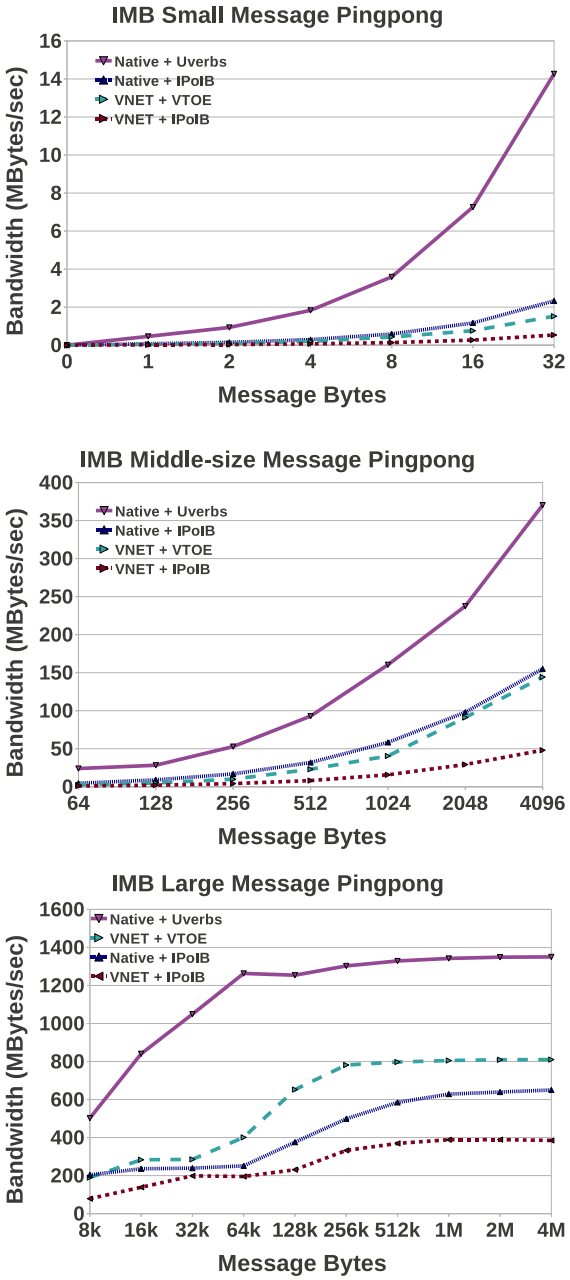
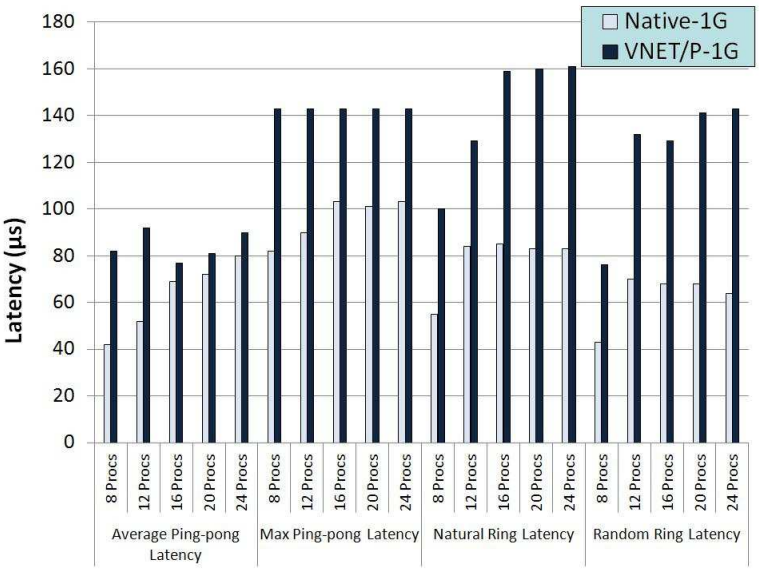
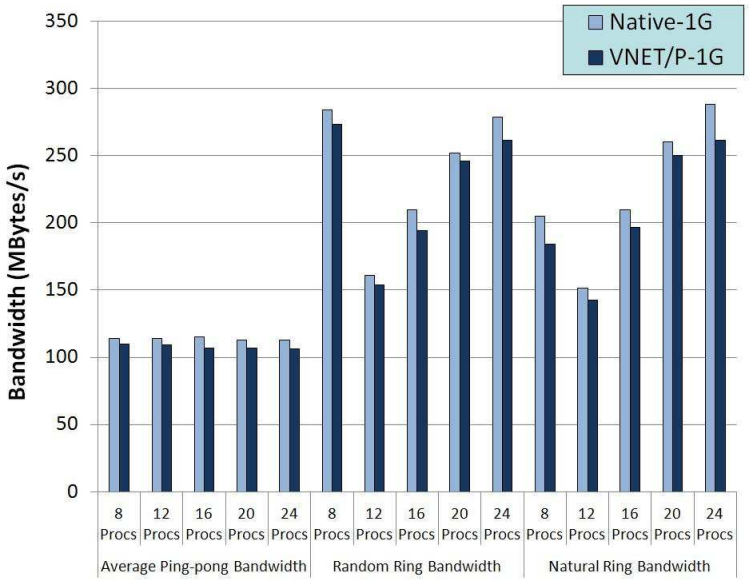


Figure 6.10: Intel MPI PingPong microbenchmark showing bidirectional bandwidth as a function of message size on InfiniBand Interconnect.

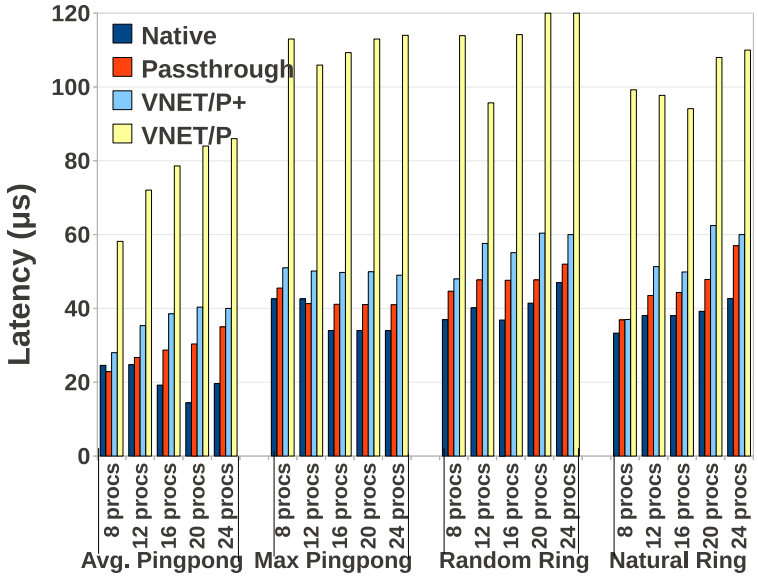


(a) Latency on 1G

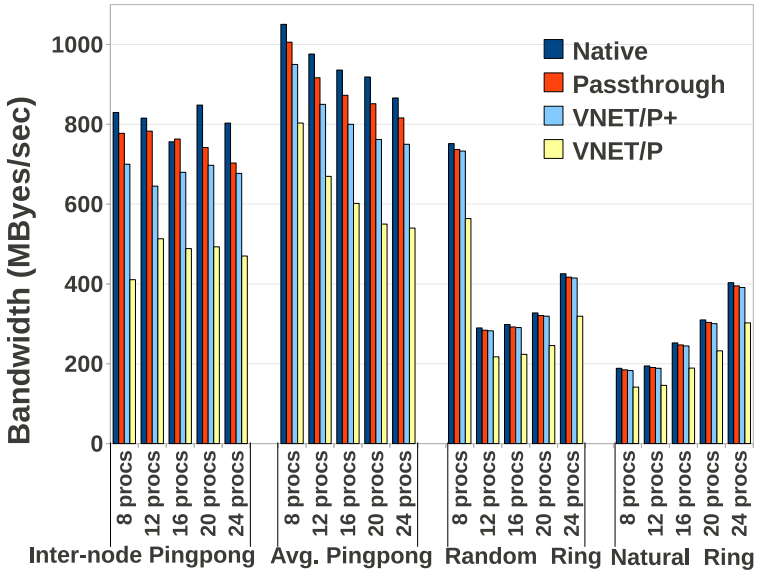


(b) Bandwidth on 1G

Figure 6.11: HPCC Latency-bandwidth benchmark for 1 Gbps Ethernet. Ring-based bandwidths are multiplied by the total number of processes in the test. The ping-pong latency and bandwidth tests show results that are consistent with the previous microbenchmarks, while the ring-based tests show that latency and bandwidth of VNET/P scale similarly to the native cases.

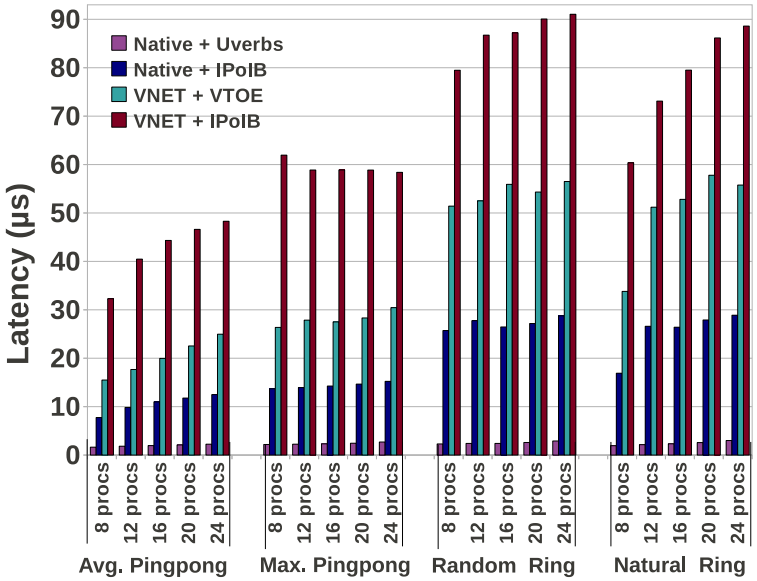


(a) HPCC Latency on 10G

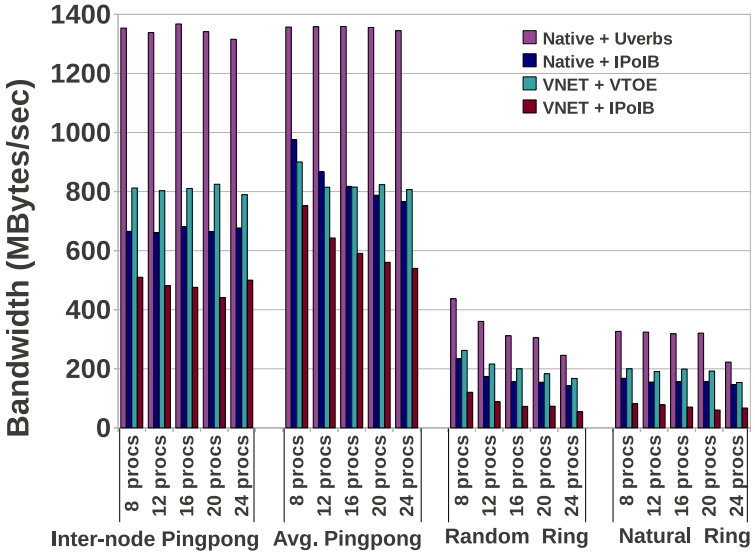


(b) HPCC Bandwidth on 10G

Figure 6.12: HPCC Latency-Bandwidth benchmark for all of Native, Passthrough VNET/P+, and VNET/P. The results are generally consistent with the previous microbenchmarks, while the ring-based tests show that latency and bandwidth of VNET/P+ scale and perform better than VNET/P.

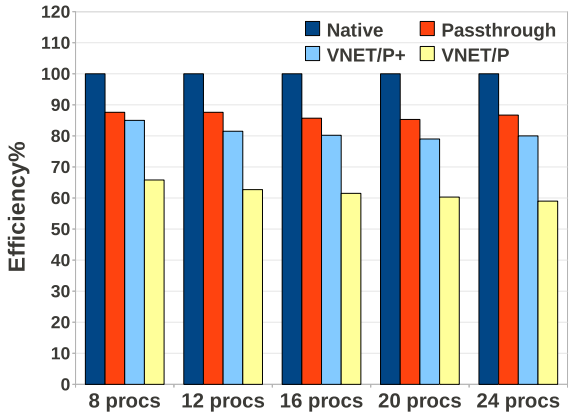


(a) HPCC Latency on InfiniBand

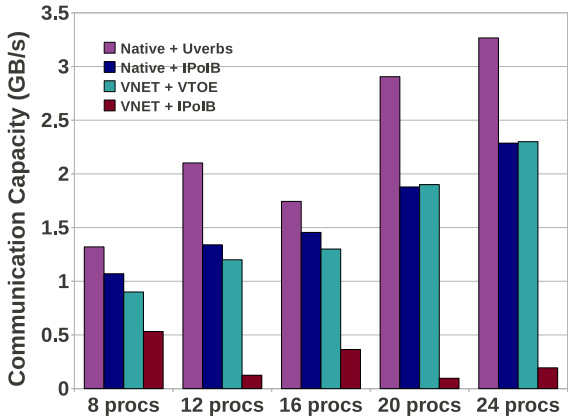


(b) HPCC Bandwidth on InfiniBand

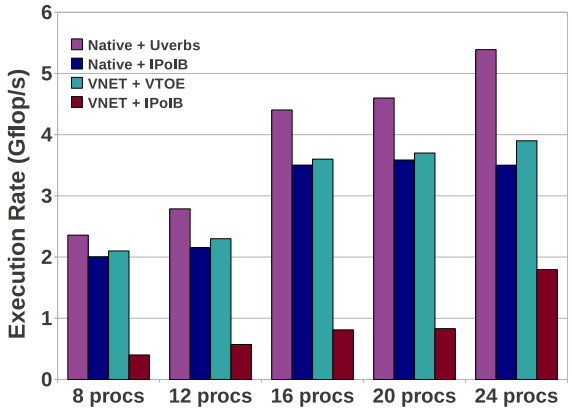
Figure 6.13: HPCC Latency-Bandwidth benchmark for all of Native+Uverb, Native+IPoIB, VNET+VTOE, and VNET+IPoIB. The results are generally consistent with the previous microbenchmarks, while the ring-based tests show that latency and bandwidth of VNET+VTOE scale and perform better than VNET+IPoIB.



(a) HPCC MPIRandomAccess

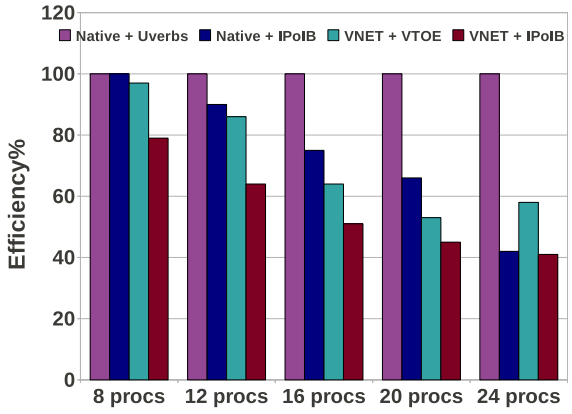


(b) HPCC PTRANS

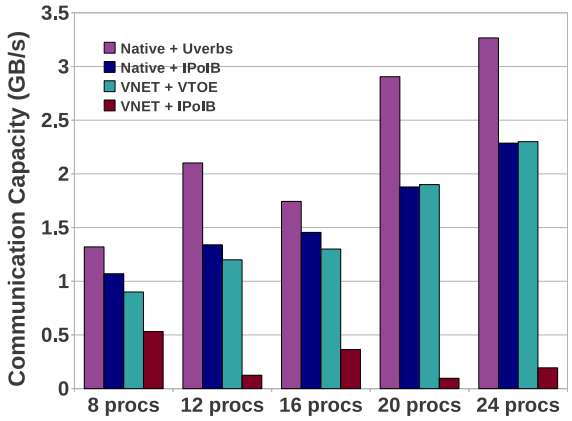


(c) HPCC MPIFFT

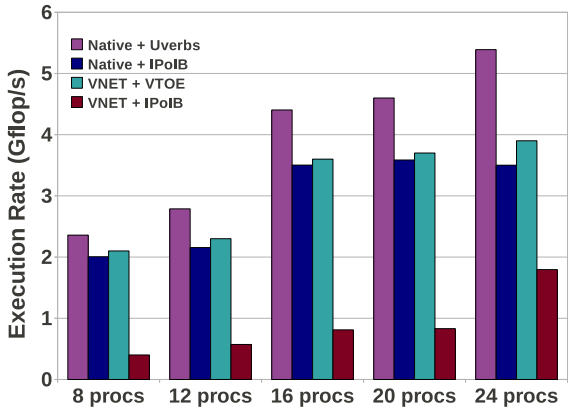
Figure 6.14: HPCC application benchmark results. VNET/P+ achieves near-native and scalable application performance when supporting parallel application workloads on 10 Gbps Ethernet with rigorous network communication.



(a) HPCC MPIRandomAccess



(b) HPCC PTRANS



(c) HPCC MPIFFT

Figure 6.15: 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.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In this dissertation, I described the VMM-level model of overlay networking in a distributed virtualized computing environment and efforts to extending this simple and flexible model to support tightly-coupled high performance computing applications running on high-performance networking hardware in current supercomputing environments, future data centers, and future clouds. VNET/P is the design and implementation of a VMM-level virtual overlay for such environments. It achieves near-native throughput and latency on 1 and 10 Gbps Ethernet, InfiniBand, Cray Gemini, and other high performance interconnects.

In addition, I presented a quantitative study of general VMM-level virtual overlay network performance on both homogeneous Ethernet and heterogeneous high-end interconnects. I observe that high latency, reduced throughput, and performance variability, are the primary problems existing in current virtual overlay networks. I also observed that delayed virtual interrupts, excessive virtual interrupts, and high-resolution timer noise are the general challenges in network overlay I/O virtualiza-

tion. To overcome these challenges, I adopted two main optimization approaches, optimistic interrupts and cut-through forwarding. Together with LWK-based noise isolation, these techniques cut overlay network latency in half, improve throughput by more than 30%, reduce network performance variability, and frequently deliver native application performance.

Besides of the analysis and optimization of the virtual overlay on both homogeneous and heterogeneous interconnects, for heterogeneous interconnects, specially, I also 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 difference, I 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 dramatically improves virtual Ethernet overlay TCP throughput by more than 2.5 times, cuts TCP latency by 50%, and greatly improves TCP application performance.

Together, these techniques enable VNET/P system to provide a simple and flexible high-performance level 2 Ethernet network abstraction in a large range of systems no matter what the actual underlying networking technology is, with minimal overhead.

I conclude that virtual networking is able to deliver near-native HPC application performance through VMM-level virtual overlay, optimistic interrupts, cut-through forwarding and virtual TCP offload. improve HPC in a cloud environment through improving the performance of virtual overlay networks.

7.2 Future Work

7.2.1 Additional VNET Optimizations

Although VTOE has reduced VNET+IPoIB latency on InfiniBand by 50%, its latency is still high. 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 overlays processing with the virtual interrupt emulation. Merging this technique into the virtual TCP offload model should further 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 to enable remote user space memory copies without the intervention of either guest kernels or VTOE modules, avoiding all data copies.

7.2.2 Level of Network Virtualization

In this dissertation, I focus primarily on layer 2 network virtualization. Layer 2 virtual overlay networks provides good portability, location independence, and hardware independence, however, the complete layer 2 abstraction can cause significant performance degradation over high-end interconnects due to semantic differences between virtual overlay features and underlying physical network features.

Another option for network virtualization is to virtualize network in different levels, for example, layer 4. Layer 4 virtual overlay networks provide guest systems with an opportunity to offload higher-level protocol characteristics into VMMs and even in hardware, which can realize software and hardware semantic conversions. however, the software-based semantic mapping exposes a significant burden to the

VMM, specially as the system scale gets larger and larger. The overheads could be potentially reduced if the optimizations are supported in hardware.

For the reasons listed above, understanding the tradeoffs of different levels of network virtualization is a potential direction for future work. This dissertation provides some initial results in this direction, as the VTOE optimization is essentially layer 4 virtualization, supported by layer 2 network virtualization for portability. Additional work is needed to fully understand the performance, virtualization overhead, and portability tradeoffs with network virtualization and different layers, particularly for different applications and protocols.

References

[1] Achieving 10 gbs using xen para-virtualized network drivers.

[2] Hyper-V. http://en.wikipedia.org/wiki/Hyper-V.

[3] iperf homepage. (web page). http://sourceforge.net/projects/iperf/.

[4] KVM. http://www.linux-kvm.org/page/Main_Page.

[5] Microsoft. http://en.wikipedia.org/wiki/Microsoft_Virtual_Server.

[6] Opensolaris network virtualization and resource control.

[7] 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.

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

[9] VMware ESX. http://en.wikipedia.org/wiki/VMware_ESX.

[10] Vyatta. http://en.wikipedia.org/wiki/Vyatta.

[11] Xen. https://en.wikipedia.org/wiki/Xen.

[12] H. Abu-Libdeh, P. Costa, A. Rowstron, G. O'Shea, and A. Donnelly. Symbiotic routing in future data centers. In *Proceedings of SIGCOMM*, August 2010.

[13] D. Andersen, H. Balakrishnan, F. Kaashoek, and R. Morris. Resilient overlay networks. In *18th ACM Symposium on Operating Systems Principles (SOSP 2001)*, 2001.

[14] A. C. Bavier, N. Feamster, M. Huang, L. L. Peterson, and J. Rexford. In vini veritas: realistic and controlled network experimentation. In *Proceedings of SIGCOMM*, September 2006.

[15] X. Chang, J. K. Muppala, Z. Han, and J. Liu. Analysis of interrupt coalescing schemes for receive-livelock problem in gigabit ethernet network hosts. In *ICC'08*, pages 1835–1839.

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

[17] Z. Cui, P. G. Bridges, J. R. Lange, and P. A. Dinda. Virtual TCP offload: optimizing ethernet overlay performance on advanced interconnects. In *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*, HPDC '13, pages 49–60, New York, NY, USA, 2013. ACM.

[18] Z. Cui, L. Xia, P. G. Bridges, P. A. Dinda, and J. R. Lange. 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*, SC '12, pages 99:1–99:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[19] P. Dinda, A. Sundararaj, J. Lange, A. Gupta, and B. Lin. Methods and Systems for Automatic Inference and Adaptation of Virtualized Computing Environments, March 2012. United States Patent Number 8,145,760.

[20] Y. Dong, D. Xu, Y. Zhang, and G. Liao. Optimizing network I/O virtualization with efficient interrupt coalescing and virtual receive side scaling. In *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, pages 26 – 34, sept. 2011.

[21] C. Evangelinos and C. Hill. Cloud Computing for parallel Scientific HPC Applications: Feasibility of running Coupled Atmosphere-Ocean Climate Models on Amazon's EC2. In *Proceedings of Cloud Computing and its Applications (CCA)*, October 2008.

[22] K. B. Ferreira, P. G. Bridges, and R. Brightwell. Characterizing application sensitivity to OS interference using kernel-level noise injection. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC'08)*, page 19. IEEE/ACM, 2008.

[23] R. Figueiredo, P. A. Dinda, and J. Fortes. A case for grid computing on virtual machines. In *23rd International Conference on Distributed Computing Systems (ICDCS 2003)*, May 2003. To Appear.

[24] R. Fromm and N. Treuhaft. Revisiting the cache interference costs of context switching, 2007.

[25] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. 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.

[26] A. Ganguly, A. Agrawal, P. O. Boykin, and R. Figueiredo. IP over P2P: Enabling self-configuring virtual ip networks for grid computing. In *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, April 2006.

[27] T. Garfinkel and M. Rosenblum. 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*, HOTOS'05, pages 20–20, Berkeley, CA, USA, 2005. USENIX Association.

[28] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: A scalable and flexible data center network. In *Proceedings of SIGCOMM*, August 2009.

[29] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu. Bcube: A high performance, server-centric network architecture for modular data centers. In *Proceedings of SIGCOMM*, August 2009.

[30] A. Gupta. *Black Box Methods for Inferring Parallel Applications' Properties in Virtual Environments*. PhD thesis, Northwestern University, May 2008. Technical Report NWU-EECS-08-04, Department of Electrical Engineering and Computer Science.

[31] A. Gupta and P. A. Dinda. Inferring the topology and traffic load of parallel programs running in a virtual machine environment. In *10th Workshop on Job Scheduling Strategies for Parallel Processing (JSPPS 2004*, June 2004.

[32] A. Gupta, M. Zangrilli, A. Sundararaj, A. Huang, P. Dinda, and B. Lowekamp. Free network measurement for virtual machine distributed computing. In *20th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2006.

[33] T. Hoefler, T. Schneider, and A. Lumsdaine. Characterizing the Influence of System Noise on Large-Scale Applications by Simulation. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC'10)*, Nov. 2010.

[34] Y. hua Chu, S. Rao, S. Sheshan, and H. Zhang. Enabling conferencing applications on the internet using an overlay multicast architecture. In *Proceedings of ACM SIGCOMM 2001*, 2001.

[35] W. Huang, J. Liu, B. Abali, and D. Panda. A case for high performance computing with virtual machines. In *Proceedings of the 20th ACM International Conference on Supercomputing (ICS)*, June–July 2006.

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

[37] Intel. Intel cluster toolkit 3.0 for Linux. http://software.intel.com/en-us/articles/intel-mpi-benchmarks/.

[38] X. Jiang and D. Xu. Violin: Virtual internetworking on overlay infrastructure. Technical Report CSD TR 03-027, Department of Computer Sciences, Purdue University, July 2003.

[39] S. T. Jones. Implicit operating system awareness in a virtual machine monitor, 2007.

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

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

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

[43] D. A. Joseph, J. Kannan, A. Kubota, K. Lakshminarayanan, I. Stoica, and K. Wehrle. Ocala: An architecture for supporting legacy applications over overlays. In *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation (NSDI)*, May 2006.

[44] M. Kallahalla, M. Uysal, R. Swaminathan, D. E. Lowell, M. Wray, T. Christian, N. Edwards, C. I. Dalton, and F. Gittler. Softudc: A software-based data center for utility computing. *IEEE Computer*, 37(11):38–46, 2004.

[45] C. Kim, M. Caesar, and J. Rexford. Floodless in seattle: a scalable Ethernet architecture for large enterprises. In *Proceedings of SIGCOMM*, August 2008.

[46] S. Kumar, H. Raj, K. Schwan, and I. Ganev. Re-architecting vmms for multicore systems: The sidecore approach. In *Proceedings of the 2007 Workshop on the Interaction between Operating Systems and Computer Architecture*, June 2007.

[47] J. Lange and P. Dinda. Transparent network services via a virtual traffic layer for virtual machines. In *Proceedings of the 16th IEEE International Symposium on High Performance Distributed Computing (HPDC)*, June 2007.

[48] J. Lange, K. Pedretti, P. Dinda, C. Bae, P. Bridges, P. Soltero, and A. Merritt. Minimal-overhead virtualization of a large scale supercomputer. In *Proceedings of the 2011 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, March 2011.

[49] J. Lange, K. Pedretti, T. Hudson, P. Dinda, Z. Cui, L. Xia, P. Bridges, A. Gocke, S. Jaconette, M. Levenhagen, and R. Brightwell. Palacios and Kitten: New High Performance Operating Systems for Scalable Virtualized and Native Supercomputing. In *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, April 2010.

[50] J. Lange, A. Sundararaj, and P. Dinda. Automatic dynamic run-time optical network reservations. In *14th IEEE International Symposium on High Performance Distributed Computing (HPDC)*, July 2005.

[51] C. Li, C. Ding, and K. Shen. Quantifying the cost of context switch. In *Proceedings of the 2007 workshop on Experimental computer science*, ExpCS '07, New York, NY, USA, 2007. ACM.

[52] B. Lin and P. Dinda. Vsched: Mixing batch and interactive virtual machines using periodic real-time scheduling. In *Proceedings of ACM/IEEE SC (Supercomputing)*, November 2005.

[53] B. Lin, A. Sundararaj, and P. Dinda. Time-sharing parallel applications with performance isolation and control. In *Proceedings of the 4th IEEE International Conference on Autonomic Computing (ICAC)*, June 2007.

[54] J. Liu, W. Huang, B. Abali, and D. Panda. High performance VMM-Bypass I/O in virtual machines. In *USENIX Annual Technical Conference*, May 2006.

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

[56] M. Mahalingam, D. Dutt, K. Duda, P. Agarwal, L. Kreeger, T. Sridhar, M. Bursell, and C. Wright. VXLAN: A framework for overlaying virtualized

layer 2 networks over layer 3 networks. IETF Network Working Group Internet Draft, Febuary 2012. Current expiration: August 2012.

[57] A. Menon, A. L. Cox, and W. Zwaenepoel. Optimizing network virtualization in Xen. In *Proceedings of USENIX '06 Annual Technical Conference*, ATEC '06, pages 2–2, Berkeley, CA, USA, 2006. USENIX Association.

[58] M. F. Mergen, V. Uhlig, O. Krieger, and J. Xenidis. Virtualization for high-performance computing. *Operating Systems Review*, 40(2):8–11, 2006.

[59] R. N. Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat. Portland: A scalable fault-tolerant layer 2 data center network fabric. In *Proceedings of SIGCOMM*, August 2009.

[60] D. Nurmi, R. Wolski, C. Grzegorzyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The Eucalyptus open-source cloud-computing system. In *Proceedings of the 9th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid)*, May 2009.

[61] S. Ostermann, A. Iosup, N. Yigitbasi, R. Prodan, T. Fahringer, and D. Epema. An early performance analysis of cloud computing services for scientific computing. Technical Report PDS2008-006, Delft University of Technology, Parallel and Distributed Systems Report Series, December 2008.

[62] H. Raj and K. Schwan. High performance and scalable I/O virtualization via self-virtualized devices. In *16th IEEE International Symposium on High Performance Distributed Computing (HPDC)*, July 2007.

[63] R. Russell. virtio: towards a de-facto standard for virtual I/O devices. *Operating Systems Review*, 42(5):95–103, 2008.

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

[65] P. Ruth, P. McGachey, X. Jiang, and D. Xu. Viocluster: Virtualization for dynamic computational domains. In *IEEE International Conference on Cluster Computing (Cluster)*, September 2005.

[66] K. Salah, K. El-Badawi, and F. Haidari. Performance analysis and comparison of interrupt-handling schemes in gigabit networks. *Comput. Commun.*, 30(17):3425–3441, Nov. 2007.

[67] J. R. Santos, Y. Turner, G. Janakiraman, and I. Pratt. Bridging the gap between software and hardware techniques for I/O virtualization. In *USENIX*

2008 Annual Technical Conference on Annual Technical Conference, ATC'08, pages 29–42, Berkeley, CA, USA, 2008. USENIX Association.

[68] J. Shafer, D. Carr, A. Menon, S. Rixner, A. L. Cox, W. Zwaenepoel, and P. Willmann. Concurrent direct network access for virtual machine monitors. In *Proceedings of the 13th International Symposium on High Performance Computer Architecture (HPCA)*, February 2007.

[69] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of ACM SIGCOMM 2001*, pages 149–160, 2001.

[70] J. Sugerman, G. Venkitachalan, and B.-H. Lim. Virtualizing I/O devices on VMware workstation's hosted virtual machine monitor. In *USENIX Annual Technical Conference*, June 2001.

[71] A. Sundararaj. *Automatic, Run-time, and Dynamic Adaptation of Distributed Applications Executing in Virtual Environments*. PhD thesis, Northwestern University, December 2006. Technical Report NWU-EECS-06-18, Department of Electrical Engineering and Computer Science.

[72] A. Sundararaj and P. Dinda. Towards virtual networks for virtual machine grid computing. In *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.

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

[74] A. Sundararaj, M. Sanghi, J. Lange, and P. Dinda. An optimization problem in adaptive virtual environmnets. In *seventh Workshop on Mathematical Performance Modeling and Analysis (MAMA)*, June 2005.

[75] M. O. Tsugawa and J. A. B. Fortes. A virtual network (vine) architecture for grid computing. In *20th International Parallel and Distributed Processing Symposium (IPDPS)*, April 2006.

[76] R. Van der Wijngaart. NAS parallel benchmarks version 2.4. Technical Report NAS-02-007, NASA Advanced Supercomputing (NAS Division), NASA Ames Research Center, October 2002.

[77] D. Wolinsky, Y. Liu, P. S. Juste, G. Venkatasubramanian, and R. Figueiredo. 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 / SC)*, November 2009.

[78] L. Xia, S. Kumar, X. Yang, P. Gopalakrishnan, Y. Liu, S. Schoenberg, and X. Guo. Virtual WiFi: Bring Virtualization from Wired to Wireless. In *Proceedings of the 7th International Conference on Virtual Execution Environments (VEE'11)*, 2011.

[79] L. Xia, J. Lange, P. Dinda, and C. Bae. Investigating virtual passthrough I/O on commodity devices. *Operating Systems Review*, 43(3), July 2009. Initial version appeared at WPIO 2008.