Optimizing Overlay-based Virtual Networking Through Optimistic Interrupts and Cut-through Forwarding

Zheng Cui† Lei Xia‡ Patrick G. Bridges† Peter A. Dinda‡ John R. Lange∗

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

‡Department of EECS
Northwestern University
Evanston, IL 60208 USA
{lxia,pdinda}@northwestern.edu

∗Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260 USA
jacklange@cs.pitt.edu

***Abstract*—Overlay-based virtual networking 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 if we want to apply the model in HPC environments. Through careful study of an existing very high performance overlay-based virtual network system, we have identified two core issues limiting performance: delayed and/or excessive virtual interrupt delivery into guests, and copies between host and guest data buffers done during encapsulation. We respond with two novel optimizations: optimistic, timer-free virtual interrupt injection, and zero-copy cut-through data forwarding. These optimizations improve the latency and bandwidth of the overlay network on 10 Gbps interconnects, resulting in near-native performance for a wide range of microbenchmarks and MPI application benchmarks.**

I. 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. Increasingly these environments are turning to virtualization as a means of deploying and managing large-scale computing systems with the "Infrastructure as a Service" (IaaS) cloud computing model. These environments, when combined with virtual machines and virtual overlay networking, provide a powerful model to realize virtual distributed and parallel computing with strong isolation, portability, and recoverability properties. While giving IaaS cloud service providers full control over physical network configurations, it can provide the users of such services with location and networking hardware independence.

In this paper we focus on optimizing the performance of software-based virtual overlay network systems. We begin by analyzing the performance challenges of a virtual overlay network designed for HPC and Cloud systems, VNET/P [1]. Despite dramatically improved performance compared to other virtual overlay networks, including native performance on 1 Gbps Ethernet networks, VNET/P is still limited to near-native performance on faster networks. Specifically, in 10 Gbps Ethernet networks, VNET/P has 3 times higher latency and 60–70% throughput of native configurations. Additionally, latency exhibits a significant amount of variance. Our analysis 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.

We present two new optimizations that address the above mentioned issues in virtual overlay network systems. Furthermore, we demonstrate that these optimizations dramatically improve performance on high-end interconnects. Our optimizations include:

- *Optimistic Interrupts:* An optimistic, timer-free interrupt injection mechanism that improves both latency and throughput in the overlay network; and
- *Cut-through Forwarding:* A zero-copy cut-through data forwarding mechanism that increases the number of packets delivered per interrupt and improves the performance of optimistic interrupt injection.

These optimizations are currently implemented in *VNET/P+*, an optimized version of our VNET/P virtual overlay network implementation.[1] Compared to the VNET/P overlay implementation, VNET/P+ reduces latency by 50%, and increases throughput by more than 30%. As a consequence, it is able to provide native MPI application benchmark performance on 10 Gbps Ethernet networks.

VNET/P+ is implemented in the context of a lightweight host kernel, while the original VNET/P is implemented in a full Linux host kernel. This difference permitted a preliminary study of the effects of noise isolation on overlay performance. The results suggest that noise isolation can reduce the variability in performance.

In the work described here we 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. Our results demonstrate that this model is just as useful for optimizing virtual overlay networks as it is for virtual NICs. Furthermore, some aspects of cut-through forwarding (and noise isolation), can be applied even without

[1] VNET/P and VNET/P+ are publicly available as part of the Palacios VMM and can be downloaded from v3vee.org.

SC12, November 10-16, 2012, Salt Lake City, Utah, USA
978-1-4673-0806-9/12/$31.00

the device binding constraint. VNET/P (sans optimizations) can be run where the constraint is not possible to achieve.

The rest of the paper is organized as follows: Section II presents background on the Palacios VMM, overlay networking, and the VNET/P architecture. Section III then analyzes the performance of VNET/P, providing insight into the fundamental challenges of overlay support for high-speed network devices. Section IV follows with a description of our new optimizations for virtual overlay networks on high-speed interconnects. Sections V briefly describes an implementation of the proposed optimizations and microbenchmark results, and Section VI follows with an extensive evaluation of the impact of these optimizations using more complex benchmarks. Finally, Section VII concludes.

II. Background

We now describe Palacios and VNET/P, the software platforms in which the present work occurs, as well as the broader context of work in virtual overlay networks and virtual network optimization.

A. 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 [2]. Palacios is capable of virtualizing large scale (4096+ nodes) supercomputers with only minimal performance overheads [3]. Palacios's OS-agnostic design allows it to be embedded into a wide range of different OS architectures. Four embeddings currently exist. In this paper we employ the Linux and Kitten embeddings.

B. Virtual Overlay Networks

Current adaptive cloud computing systems use software-based overlay networks to carry inter-VM traffic. For example, the user-level VNET/U system [4]–[6] upon which VNET/P is based 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 [4], [7] and others that expose different abstractions to the VMs [8] 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 [9], but their performance has been insufficient for tightly-coupled applications [10]. Recent work on VNET/P, described in more detail in the following section, has enhanced the performance of virtual overlay networks for more tightly-coupled systems [1].

C. VNET/P Implementation

VNET/P is an in-VMM, overlay-based layer-2 virtual networking system for the Palacios VMM. As illustrated in Figure 1, VNET/P consists of a virtual NIC in each guest OS, an extension to the VMM (the VNET/P Core) that handles packet routing and interfacing to virtual NICs, and a Linux kernel module (the VNET/P Bridge) for interacting with the host's network interfaces and remote systems. For high performance applications, as in this paper, the virtual NIC conforms to the virtio interface, but several virtual NICs with hardware interfaces are also available in Palacios.

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

One key optimization employed by VNET/P is the use of an adaptive variant of sidecore processing [11] in which otherwise available processor cores are recruited to perform packet routing, encapsulation, and copying. This allows packet forwarding to be done in parallel with guest interrupt and packet processing, improving its performance in high-throughput cases.

Compared to VNET/U and other user-level software-based systems, VNET/P can effectively support communication-intensive applications in overlay networks. For example, compared to VNET/U, VNET/P reduces latency on 1 Gbps networks by more than an order of magnitude and achieves native throughput on these networks. This enables MPI application benchmarks to run on the overlay at native speeds on 1 Gbps networks. Compared to native or passthrough networking, however, VNET/P still has performance limitations, namely:

- **High latency.** VNET/P's latencies are 3 times higher than native latencies on 1 Gbps and 10 Gbps networks, which is particularly problematic for tightly-coupled HPC applications as well as recent DHT-based Cloud systems [12].
- **Variablility.** Network virtualization causes significant throughput and latency variation [13], [14]. Consistent, predictable network performance is critical to data-intensive computing, and performance variability also makes it hard to infer network congestion and bandwidth properties from end-to-end probes (e.g. TCP Vegas [15], PCP [16]).
- **Reduced throughput.** VNET/P delivers 60–70% of native throughput on 10 Gbps NICs. This impacts applica-

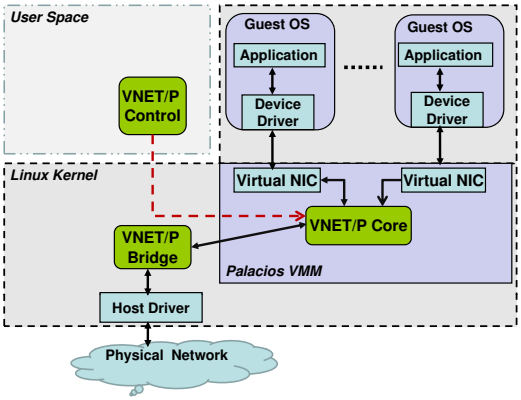


Fig. 1. VNET/P architecture.

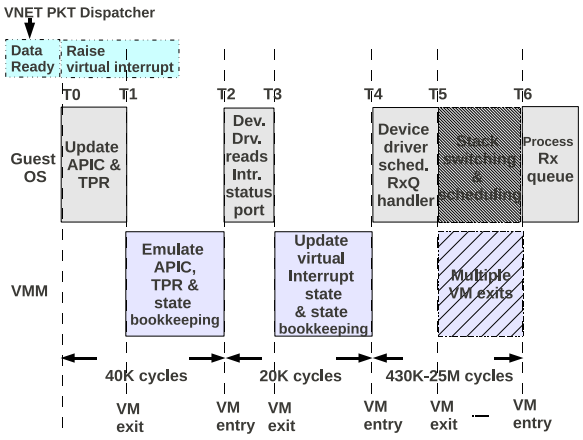


Fig. 2. Virtual interrupt time line

tion performance, as demonstrated in HPCC and some NAS application benchmarks in our previous work [1].

D. Virtual Networking Optimization

There has been a wide range of work on optimizing high-speed network interface performance in virtual machines [17]–[20], much of it focused on paravirtualizing the NIC, or by-passing the host OS, virtual machine monitor, and sometimes the guest OS. The work described in this paper 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.

Most work on optimizing software network virtualization has focused on interrupt processing; this focus is well-founded, as our analysis in Section III 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 [21]. 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 [22], [23]. Unfortunately, these techniques control virtual interrupt frequency using a high-frequency periodic timer that has high overheads and generate substantial OS noise.

III. Analysis

To more fully understand the performance challenges that high-speed networks present to virtual overlay networks, we instrumented and traced the performance of packet reception and transmission in VNET/P running in a Linux host OS on AMD Opteron systems with 10Gbps Ethernet adapters (more details on the test systems are provided in Section VI). Our analysis highlights three major challenges to overlay networks in high-speed networks: *delayed virtual interrupts*, *excessive virtual interrupts*, and *high-resolution timer noise*.

A. Delayed virtual interrupts

Although the VNET/P packet dispatcher raises a virtual interrupt to the guest OS when a packet arrives in the virtual NIC's receive buffer, 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 2 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 our experiments on our 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–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).

B. Excessive virtual interrupts

After processing an inbound packet, the VNET/P packet dispatcher interrupts 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.

C. High-resolution 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.

IV. Optimizations

To address the challenges described in the previous section, we propose a set of two main receive-side optimizations for virtual overlay network implementations: optimistic interrupts and cut-through forwarding. These optimizations act together to reduce per-packet latencies and improve throughput by overlapping VNET's packet handling with guest interrupt processing, by coalescing interrupts *without* the need for problematic high-resolution timers, and by avoiding buffering of encapsulated data when possible. These optimizations also leverage the predictable environment of a low-noise host kernel which we also use to reduce virtual network performance variability. In the remainder of this section, we describe the general approach of these optimizations and provide details on their behavior and how they are tuned.

Our optimizations are focused on scenarios in which host NIC receive rings and interrupt messages can be assigned to individual virtual NICs. Such scenarios are increasingly common in data centers, with processor and NIC vendors introducing specific hardware support for such usage. 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 such as VNET/P.

The optimizations described below 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.

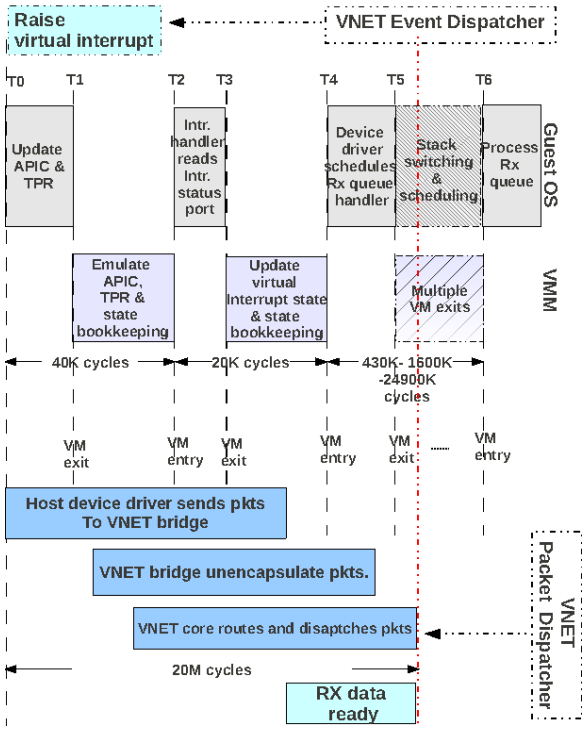


Fig. 3. Early virtual interrupt optimization to reduce latency

A. Optimistic Interrupts

The primary optimization we 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 deencapsulating) data from the host NIC to the virtual NIC. With optimistic interrupts, we 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.

1) Early Virtual Interrupt (EVI) delivery: Figure 3 illustrates our 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 deencapsulation 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 III-A. 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 our testbed, experiments show that the time at which the first packet of a train has been deencapsulated 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 2, 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, we *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 (i.e. the EVI was not performed early enough), latency increases compared to the native mode. Unoptimized VNET/P is the extreme scenario 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.

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, we 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 we assume delivery into the guest has already been done using the EVIs we 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.

3) EVI/EoC Interaction: Together, the EVI and EoC techniques that comprise our 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.

B. 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, we 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 the interrupt that notifies the NIC of configuration space changes.

C. Noise isolation to reduce performance variation

To reduce variation in throughput and latency, we target OS noise. Specifically, we 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 our 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.

V. Implementation and Microbenchmarks

To understand the impact of the optimizations described in Section IV, we implemented them in the VNET/P overlay network previously described in Section II. We then studied the effects of the optimizations on a set of simple UDP, TCP, and MPI throughput and latency microbenchmarks. Macro- and application benchmarks are described in Section VI. We refer to VNET/P enhanced with the optimizations as VNET/P+.

A. Implementation

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. This source code and changes will be made available in Palacios and Kitten in the future.

B. Testbed

Our testbed, which is used both here and in the next section, 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, and a NetEffect NE020 10 Gbps Ethernet fiber optic NIC (10GBASE-SR) in a PCI-e slot. For VNET/P and native measurements, we ran a simple Linux 2.6.32 host with a minimal BusyBox configuration, and the Palacios VMM. Passthrough and VNET/P+ measurements were made with Kitten as the host operating system and the Palacios VMM, as described in Section IV-C, unless otherwise specified. The guest used was a Linux 2.6.30 kernel also with a minimal BusyBox running on a virtual machine with a single virtio network interface, 4 cores, and 1GB of memory. Unless otherwise specified, the virtio NIC provided to the guest was configured to use 9000 byte MTUs.

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

C. Microbenchmarks

We used simple two-node ICMP, UDP, TCP, and MPI benchmarks to provide an initial characterization of the impact of our proposed optimizations. UDP throughput and goodput were measured using Iperf-2.0.4 with 8900 byte writes for 150 seconds, while TCP throughput was measured using *ttcp-1.10*. For simple MPI tests, we used the Intel MPI Benchmark Suite (IMB 3.2.2) [24] running on OpenMPI 1.3 [25], focusing on the point-to-point messaging performance.

1) Ping Latency: Figure 4 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 the passthrough latency. The passthrough itself is limited due to the need for interrupt exiting and reinjection.

2) UDP and TCP Performance: Figure 5 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).

3) Network Performance Variability: In addition to ICMP, UDP, and TCP performance, we also examined ICMP and TCP performance variability. To test latency variation, we use *ping* with 64-byte messages for 5000 iterations. To test throughput variation, we examine variation in Iperf performance with 8900 byte sends over the course of an hour.

Figure 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.

4) MPI: As shown in Figure 7, 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.

D. Understanding Low-level Behavior

We used the previously discussed benchmarks to better understand the fine-grained behavior and performance impacts of optimistic interrupts and zero-copy cut through forwarding. We 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.

Our 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. Our 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.

VI. Performance evaluation

Beyond the microbenchmarks we described in the previous section, we also evaluated VNET/P+ using the HPC Challenge and NAS benchmarks, with the goal of characterizing the performance impact of fast overlay networking and our optimizations on communication-intensive applications.

A. HPC Challenge benchmarks

The HPC Challenge (HPCC) benchmarks [26] are a set of macro and application benchmarks for evaluating various aspects of the performance of high performance computing systems. We used the communication-oriented macro-benchmarks and application benchmarks to compare the performance of VNET/P+ with native, passthrough, and VNET/P approaches. For these tests, each VM was configured with 4 virtual cores, 1 GB RAM, and a virtio NIC. For passthrough and VNET/P testing, each host had one VM running on it. We ran tests with 2, 3, 4, 5, and 6 VMs with 4 HPCC processes started on each VM. Thus our performance results are based on HPCC with 8, 12, 16, 20, and 24 processes. In the native cases, no VMs are used and the processes ran directly on the host.

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

Figure 8 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 we saw 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 our optimizations can substantially enhance the performance of a software-based overlay virtual network like VNET/P.

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

In MPIRandomAccess, random numbers are generated and written to a distributed table, with local buffering. Performance is measured in billions of updates per second (GUPs) that are performed. Figure 9(a) shows the results of MPIRandomAccess, comparing the VNET/P+, VNET/P, Passthrough,

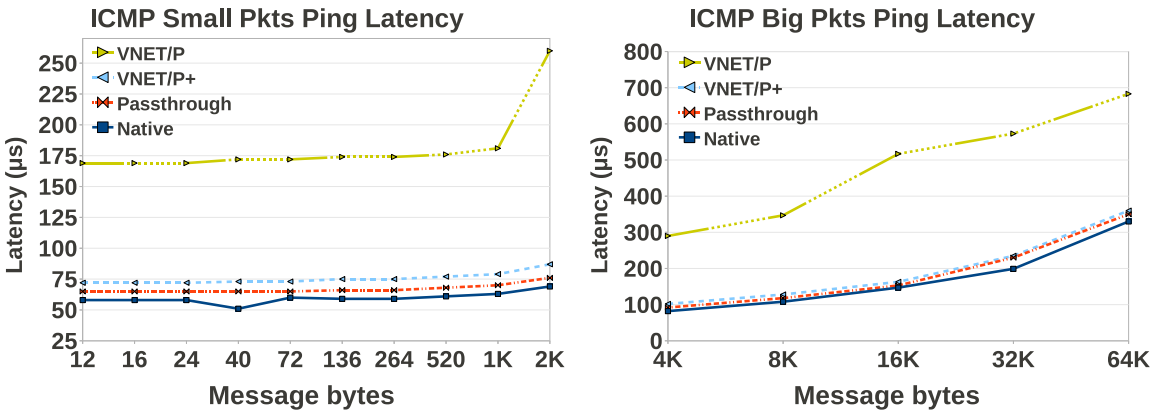


Fig. 4. End-to-end round-trip latency of VNET as a function of ICMP packet size. Small packet latencies are: VNET/P+—72μs, Passthrough—65μs, Native—58μs, VNET/P—169μs.

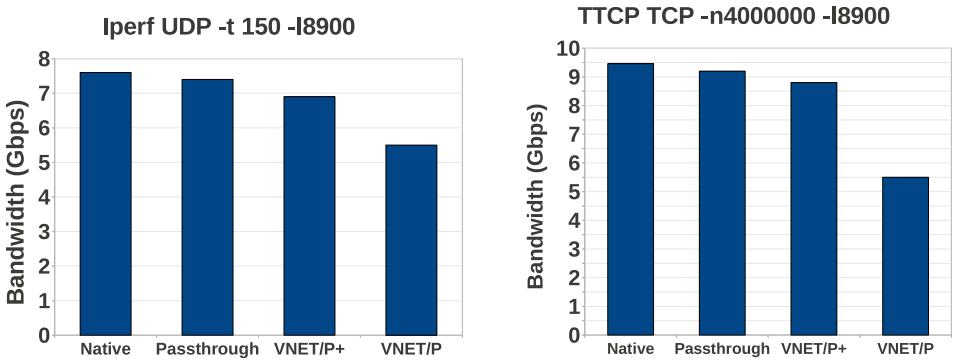
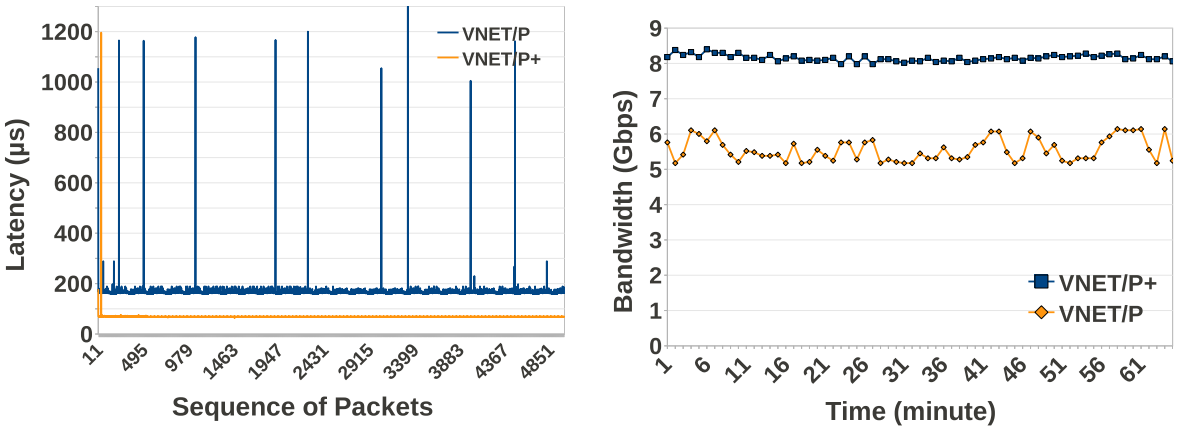


Fig. 5. End-to-end UDP goodput and TCP throughput of VNET/P+ and VNET/P on 10 Gbps network. VNET/P+ performs better then VNET/P for the 10 Gbps network



(a) 5000 iterations of 64 bytes latency variation

(b) 1 hour TCP throughput variation.

Fig. 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.

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 the simultaneous communications between pairs of processors. The performance is measured in the total communication capacity (GB/s) of the network. Figure 9(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.

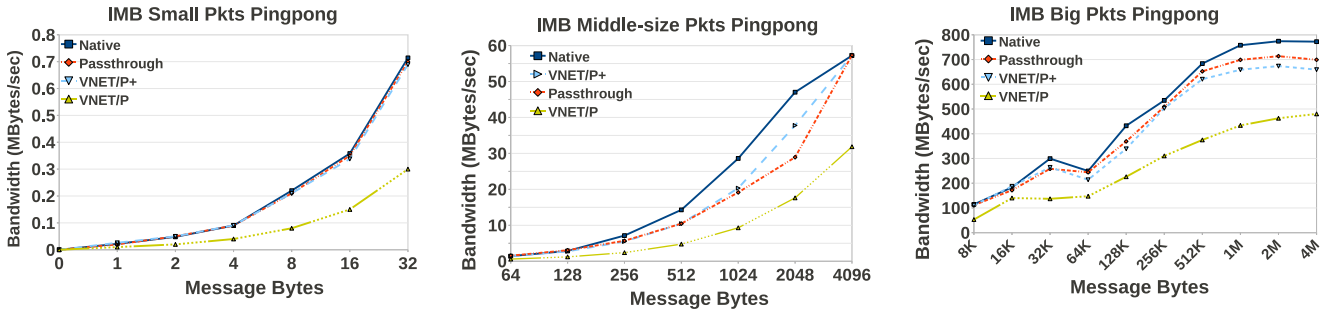
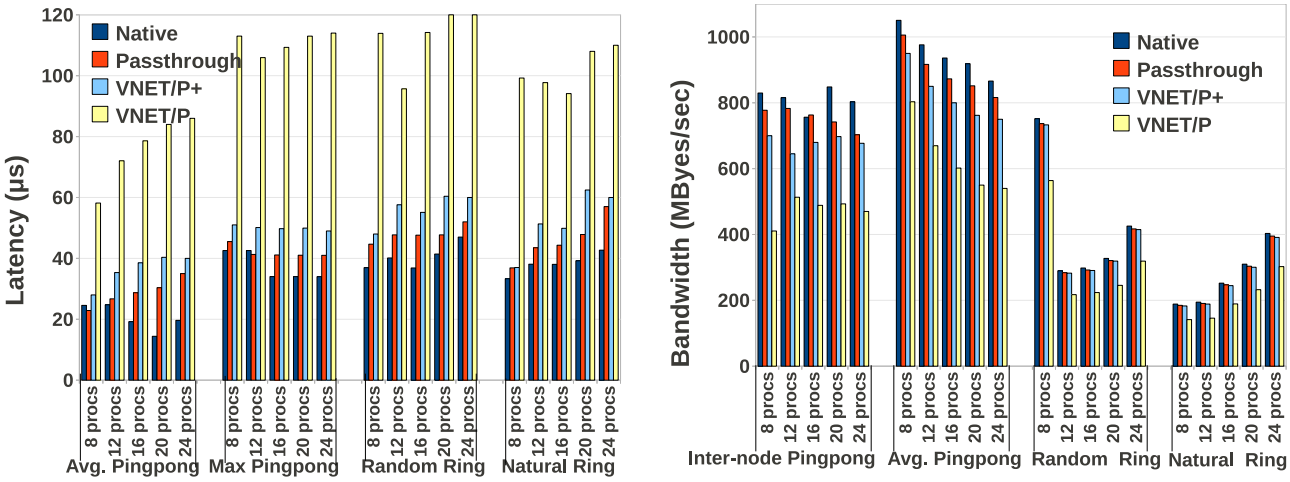


Fig. 7. Intel MPI PingPong microbenchmark showing bidirectional bandwidth as a function of message size on the 10Gbps Ethernet



(a) HPCC Latency on 10G

(b) HPCC Bandwidth on 10G

Fig. 8. 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.

MPIFFT implements a double precision complex one-dimensional Discrete Fourier Transform (DFT). Its performance is measured in Gflop/s. Figure 9(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 our optimizations to the VNET/P networking system are likely to strongly be felt in application codes.

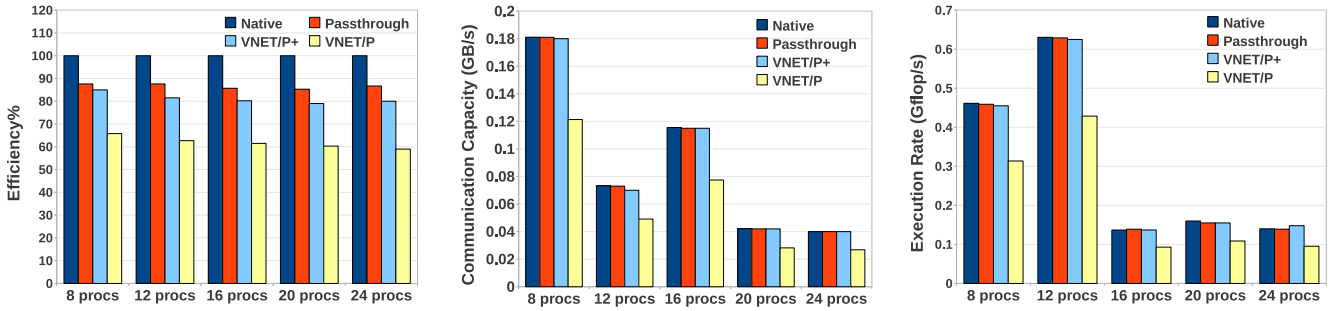
B. NAS Benchmarks

We compared the performance of VNET/P+, VNET/P, passthrough, and the native environment on the complete NAS parallel benchmark suite (NPB) [27]. NPB consists of five kernels and three pseudo-applications, and is widely used in parallel performance evaluation. We specifically use NPB-MPI 2.4 in our evaluation. In our description, we name executions 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.

We run each benchmark with at least two different scales and one problem size. One VM is run on each physical machine, and it is configured as described in Section VI-A. 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 with 16 processes have 4 VMs with 4 processes per VM. We report each benchmark's *Mop/s total* result for all four cases.

Figure 10 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.



(a) HPCC MPIRandomAccess (b) HPCC PTRANS (c) HPCC MPIFFT

Fig. 9. HPCC application benchmark results. VNET/P+ achieves near-native and scalable application performance when supporting parallel application workloads on 10 Gbps networks with rigorous network communication.

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%**

Fig. 10. 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.

The results of our 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.

VII. Conclusions

In this paper, we presented a quantitative study of general virtual overlay network performance on 10 Gbps Ethernet. We observe that high latency, reduced throughput, and performance variability are the primary problems existing in current virtual overlay networks. We observed that delayed virtual interrupts, excessive virtual interrupts, and high-resolution timer noise are the challenges in network overlay I/O virtualization. To overcome these challenges, we adopt 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.

References

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

[2] 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 2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, April 2010, pp. 1 –12.

[3] J. R. Lange, K. Pedretti, P. Dinda, P. G. Bridges, C. Bae, P. Soltero, and A. Merritt, "Minimal-overhead virtualization of a large scale supercomputer," in *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '11. New York, NY, USA: ACM, 2011, pp. 169–180. [Online]. Available: http://doi.acm.org/10.1145/1952682.1952705

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

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

[6] 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.

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

[8] 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 2009)*, November 2009.

[9] 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.

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

[11] 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.

[12] A. Lakshman and P. Malik, "Cassandra: A structured storage system on a P2P network," in *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing*. New York, NY, USA: ACM, 2009.

[13] G. Wang and T. Ng, "The impact of virtualization on network performance of Amazon EC2 data center," in *Proceedings of IEEE INFOCOM 2010*, march 2010, pp. 1 –9.

[14] A. Gulati, A. Merchant, and P. Varman, "mclock: Handling throughput variability for hypervisor IO scheduling."

[15] L. S. Brakmo and L. L. Peterson, "TCP Vegas: End to end congestion avoidance on a global internet," *IEEE Journal on Selected Areas in Communications*, vol. 13, pp. 1465–1480, 1995.

[16] T. Anderson, A. Collins, A. Krishnamurthy, and J. Zahorjan, "PCP: Efficient endpoint congestion control," in *Proceedings of Third Symposium on Networked Systems Design and Implementation (NSDI'06)*, 2006.

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

[18] J. Liu, W. Huang, B. Abali, and D. Panda, "High performance VMM-bypass I/O in virtual machines," in *Proceedings of the USENIX Annual Technical Conference*, May 2006.

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

[20] L. Xia, J. Lange, P. Dinda, and C. Bae, "Investigating Virtual Passthrough I/O on Commodity Devices," *Operating Systems Review*, vol. 43, no. 3, July 2009, initial version appeared at WIOV 2008.

[21] K. Salah, K. El-Badawi, and F. Haidari, "Performance analysis and comparison of interrupt-handling schemes in gigabit networks," *Comput. Commun.*, vol. 30, no. 17, pp. 3425–3441, Nov. 2007. [Online]. Available: http://dx.doi.org/10.1016/j.comcom.2007.06.013

[22] 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*, sept. 2011, pp. 26 –34.

[23] 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*, pp. 1835–1839.

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

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

[27] R. Van der Wijngaart, "NAS parallel benchmarks version 2.4," NASA Advanced Supercomputing (NAS Division), NASA Ames Research Center, Tech. Rep. NAS-02-007, October 2002.