Technical Report
NWU-EECS-11-07
July 14, 2011

VNET/P: Bridging the Cloud and High Performance Computing Through Fast Overlay Networking

Lei Xia Zheng Cui John Lange
Yuan Tang Peter Dinda Patrick Bridges

Abstract

Overlay networking with a layer 2 abstraction provides a powerful model for virtualized wide-area distributed computing resources, including for high performance computing (HPC) on collections of virtual machines (VMs). With the emergence of cloud computing, it is now possible to allow VMs hosting tightly-coupled HPC applications to seamlessly migrate between distributed cloud resources and tightly-coupled supercomputing and cluster resources. However, to achieve the application performance that the tightly-coupled resources are capable of, it is important that the overlay network not introduce significant overhead relative to the native hardware. To support such a model, we describe the design, implementation, and evaluation of a virtual networking system that has negligible latency and bandwidth overheads in 1-10 Gbps networks. Our system, VNET/P, is embedded into our publicly available Palacios virtual machine monitor (VMM). VNET/P achieves native performance on 1 Gbps Ethernet networks and very high performance on 10 Gbps Ethernet networks and InfiniBand. On the latter networks, performance is converging to native speeds as we continue to enhance VNET/P. In comparison to VNET/U, its previously demonstrated, and highly optimized, user-level counterpart, VNET/P can achieve bandwidths over 33 times as high. These results suggest that it is feasible to extend a software-based overlay network designed to facilitate computing at wide-area scales into tightly-coupled environments.

This project is made possible by support from the National Science Foundation (NSF) via grants CNS-0709168 and CNS-0707365, and the Department of Energy (DOE) via grant DE-SC000534.

Keywords: virtualization, cloud computing, high performance computing, overlay networking

VNET/P: Bridging the Cloud and High Performance Computing Through Fast Overlay Networking

Lei Xia* Zheng Cui§ John Lange‡
Yuan Tang† Peter Dinda* Patrick Bridges§

* Department of EECS
Northwestern University
{lxia,pdinda}@northwestern.edu

§ Department of CS
University of New Mexico
{cuizheng,bridges}@cs.unm.edu

‡ Department of CS
University of Pittsburgh
jacklange@cs.pitt.edu

† School of CSE
UESTC, China
ytang@uestc.edu.cn

ABSTRACT

Overlay networking with a layer 2 abstraction provides a powerful model for virtualized wide-area distributed computing resources, including for high performance computing (HPC) on collections of virtual machines (VMs). With the emergence of cloud computing, it is now possible to allow VMs hosting tightly-coupled HPC applications to seamlessly migrate between distributed cloud resources and tightly-coupled supercomputing and cluster resources. However, to achieve the application performance that the tightly-coupled resources are capable of, it is important that the overlay network not introduce significant overhead relative to the native hardware. To support such a model, we describe the design, implementation, and evaluation of a virtual networking system that has negligible latency and bandwidth overheads in 1–10 Gbps networks. Our system, VNET/P, is embedded into our publicly available Palacios virtual machine monitor (VMM). VNET/P achieves native performance on 1 Gbps Ethernet networks and very high performance on 10 Gbps Ethernet networks and InfiniBand. On the latter networks, performance is converging to native speeds as we continue to enhance VNET/P. In comparison to VNET/U, its previously demonstrated, and highly optimized, user-level counterpart, VNET/P can achieve bandwidths over 33 times as high. These results suggest that it is feasible to extend a software-based overlay network designed to facilitate computing at wide-area scales into tightly-coupled environments.

1. INTRODUCTION

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. Indeed, for loosely coupled applications, this concept has readily moved from research [6, 37, 17] to practice [44, 31]. However, *Tightly-coupled* scalable high performance computing (HPC) applications currently remain the purview of resources such as clusters and supercomputers.

The current limitation of cloud computing systems to *loosely-coupled* applications is not due to machine virtualization limitations. Current virtual machine monitors (VMMs) and other virtualization mechanisms present negligible overhead for CPU and memory intensive workloads [15, 28]. With VMM-bypass [27] or self-virtualizing devices [33] communication overheads can also be negligible.

While cloud computing does provide minimal intra-node overhead, it is well known that the network infrastructure imposes significant and frequently unpredictable performance penalties. This is a result of both the large scale of cloud data centers as well as the common practice of using commodity network architectures such as 1 Gbps Ethernet. This is especially an acute issue for tightly-coupled parallel applications specifically designed to target large scale machines with specialized high-end interconnects such as InfiniBand. Considerable work has gone into addressing these issues at the network hardware layer with designs such as Portland, VL2, and others [30, 10, 18]. 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 and lack compatibility with the more specialized interconnects present on HPC systems. The work we describe here is intended to support a model in which tightly-coupled applications can seamlessly migrate to and from heterogeneous data center networks, specialized HPC machines, and even the wide-area.

Beyond the need to support such a model across today's high throughput/low-latency HPC environments, we note that data center network design and cluster/supercomputer network design seems to be converging [2, 11]. This suggests that future data centers deployed for general purpose cloud computing will become an increasingly better fit for tightly-coupled parallel applications, and therefore such environments could potentially also benefit from our work.

The current limiting factor in employing cloud computing for tightly-coupled applications is the performance of the virtual networking system. This system combines a simple networking ab-

.

straction within the VMs with location-independence, hardware-independence, and traffic control. For example, an overlay networking system that exposes a layer 2 abstraction lets the user treat his VMs as being on a simple LAN, while allowing the VMs to be migrated seamlessly across resources. By controlling the overlay, the cloud provider can control the bandwidth and the paths between VMs over which traffic flows. Such systems [40, 36] and others that expose different abstractions to the VMs [8, 45] 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, 5], but their performance is insufficient for tightly-coupled applications [32].

In response to this limitation, we have designed, implemented, and evaluated VNET/P, an overlay network that provides a simple, persistent layer 2 abstraction to a collection of VMs regardless of their current locations. VNET/P shares its model and vision with our previously reported VNET/U system, but unlike VNET/U, it is designed to achieve near-native performance in the 1 Gbps and 10 Gbps switched networks common in clusters today, and pave the way for even faster networks, such as InfiniBand, in the future. The model and vision of VNET in general, and the limitations of VNET/U specifically, are described in more detail in Section 2.

VNET/P is implemented in the context of our publicly available, open source Palacios VMM, which is described in Section 3. Palacios [23] is in part designed to support virtualized supercomputing, with scaling studies of tightly-coupled parallel applications of up to 4096 nodes on Sandia National Labs' Cray XT-class Red Storm machine showing $< 5\%$ application performance degradation compared to native performance [22]. A detailed description of VNET/P's design and implementation is given in Section 4. As a part of Palacios, VNET/P is publicly available. VNET/P could be implemented in other VMMs, and as such provides a proof-of-concept that virtual networking for VMs, with performance overheads low enough to be inconsequential even in a cluster environment, is clearly possible.

The performance evaluation of VNET/P (Section 5) shows that it is able to achieve native bandwidth on 1 Gbps Ethernet with a small increase in latency, and very high bandwidth on 10 Gbps Ethernet with a similar, small latency increase. Latency increases are predominantly due to the limited support for selective interrupt exiting in the current AMD and Intel hardware virtualization extensions, which we expect to be change soon. We also demonstrate in Section 6 that VNET/P can effectively support running Ethernet-based networked programs on non-Ethernet HPC communication device, specifically InfiniBand NICs.

Through the use of low overhead virtual networking in high-bandwidth, low-latency environments such as current clusters and supercomputers, and future data centers, we seek 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 us to seamlessly and *practically* extend the already highly effective virtualization-based IaaS cloud computing model to such environments. A parallel computation running on a collection of VMs could seamlessly migrate from the wide-area to a cluster if it proved to require a more tightly-coupled network, and conversely, from the cluster to the wide-area if its communication requirements were relaxed. Finally, the computation could migrate from one cluster to another, for example, to take advantage of a better price.

Our contributions are as follows:

- We articulate the benefits of extending virtual networking for VMs down to clusters and supercomputers with high performance networks. These benefits are also applicable to current and future data centers that support IaaS cloud computing.
- We describe the design and implementation of a virtual networking system, VNET/P, that does so. The design could be applied to other virtual machine monitors and virtual network systems.
- We evaluate VNET/P, finding that it provides performance with negligible overheads on 1 Gbps Ethernet networks, and manageable overheads on 10 Gbps Ethernet networks.
- We describe how VNET/P also provides its abstraction on top of InfiniBand hardware, allowing guests to exploit such hardware without any special drivers or an InfiniBand stack.
- We describe how a system like VNET/P could be made even faster.

It is important to point out that VNET/P is an *overlay* network that is implemented entirely in *software*. It is not a data center network design, facilitated by hardware, nor a system to provide Internet connectivity to customer VMs in a data center. The point of a VNET overlay is to allow a collection of VMs to communicate as if they are on a simple physical Ethernet network regardless of their location. As with any overlay, VNET has numerous mechanisms to establish, monitor, and control the overlay topology and routing on it, which we summarize from prior work in Section 2. The focus of this paper, and of VNET/P, is on how to extend VNET to tightly-coupled environments. In this, the key challenge is how to make it possible for two VMs, both mapped to such an environment, to communicate with the bandwidth and latency the environment is capable of, all while maintaining the VNET abstraction. The paper focuses on this challenge.

2. VNET MODEL AND CURRENT IMPLEMENTATION

We now describe the VNET model and how it supports adaptive computing, as well as the most recent implementation of VNET prior to this work and its limitations.

2.1 Model

The VNET model was originally designed to support adaptive computing on distributed virtualized computing resources within the Virtuoso system [38], 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 key requirements, which also hold for the present paper, were as follows.

1. VNET would make within-VM network configuration the sole responsibility of the VM owner.
2. VNET would provide 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.
3. VNET would provide hardware independence to VMs, allowing them to use diverse networking hardware without requiring the installation of specialized software.
4. VNET would provide 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. The VNET model and our initial implementation are described in detail elsewhere [40].

The VNET overlay is dynamically reconfigurable, and can act as a locus of activity for an an adaptive system such as Virtuoso. Focusing on parallel and distributed applications running in loosely-coupled virtualized distributed environments e.g., "IaaS Clouds", we we demonstrated that the VNET "layer" can be effectively used to: (1) monitor application communication and computation behavior [13, 12]), (2) monitor underlying network behavior [14]), (3) formulate performance optimization problems [42, 39], (4) address such problems through VM migration and overlay network control [41], scheduling [25, 26], network reservations [24]), and network service interposition [20].

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.

2.2 VNET/U implementation

The third generation implementation of VNET, which we now refer to as 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 [41]) 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.

VNET/U is implemented as a user-level system on top of our VTL traffic library [20]. 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 allow 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.

2.3 VNET/U performance

VNET/U is among the fastest virtual networks implemented using user-level software, achieving 21.5 MB/s (172 Mbps) [20] 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. These 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 paper. VNET/U is fundamentally limited by the kernel/user space transitions needed to handle a guest's packet send or receive. In VNET/P, we move VNET directly into the VMM to avoid such transitions.

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 and the University of New Mexico, with close collaboration with Sandia National Labs for our efforts in the virtualization of supercomputers. Detailed information about Palacios can be found elsewhere [23, 21, 46]. Palacios is capable of virtualizing large scale (4096+ nodes) supercomputers with only minimal performance overheads [22].

At a high level Palacios is designed to be an OS-independent, embeddable VMM that is widely compatible with existing OS architectures. In other words, Palacios is not an operating system, nor does it depend on any one specific OS. This OS-agnostic approach allows Palacios to be embedded into a wide range of different OS architectures, each of which can target their own specific environment (for instance 32 or 64 bit operating modes). Palacios is intentionally designed to maintain the separation between the VMM and OS. In accordance with this, Palacios relies on the hosting OS for such things as scheduling and process/thread management, memory management, and physical device drivers. This allows OS designers to control and use Palacios in whatever ways are most suitable to their architecture.

The Palacios implementation is built on the virtualization extensions deployed in current generation x86 processors, specifically AMD's SVM [3] and Intel's VT [16, 43]. A result of this is that Palacios only supports both host and guest environments that target the x86 hardware platform. However, while the low level implementation is constrained, the high level architecture is not, and can be easily adapted to other architectures with or with out hardware virtualization support. Specifically Palacios supports both 32 and 64 bit host and guest environments, both shadow and nested paging models, and a significant set of devices that comprise the PC platform. Work is also underway to support future I/O architectures such as IOMMUs. In addition to supporting full-system virtualized environments, Palacios provides support for the implementation of paravirtual interfaces. Due to the ubiquity of the x86 architecture Palacios is capable of operating across many classes of machines. To date, Palacios has successfully virtualized commodity desktops and servers, high end InfiniBand clusters, and supercomputers such as a Cray XT.

As stated earlier, Palacios is an OS-independent VMM that is designed to be embeddable into any host OS. Four embeddings currently exist: Linux, Sandia National Labs' publicly available Kitten lightweight kernel [34], Minix, and GeekOS. In this paper, we mostly employ the Linux embedding, although we use the Kitten embedding for Infiniband.

4. VNET/P DESIGN AND IMPLEMENTATION

We now describe how VNET/P has been architected and implemented in the context of Palacios as embedded in a Linux host. Section 6 describes how VNET/P is implemented in the context of a Kitten embedding. The nature of the embedding affects VNET/P primarily in how it interfaces to the underlying networking hardware and networking stack. In the Linux embedding, this interface is accomplished directly in the Linux kernel. In the Kitten embedding, the interface is done via a service VM.

4.1 Architecture

Figure 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 applica-

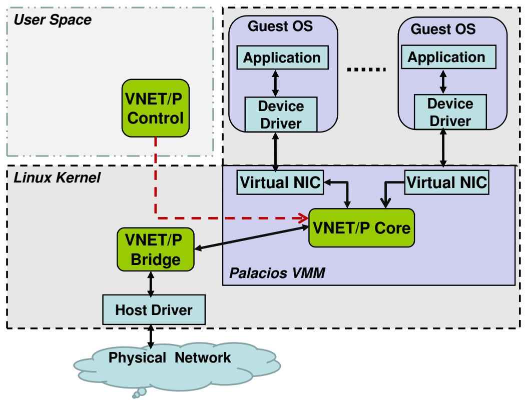


Figure 1: VNET/P architecture

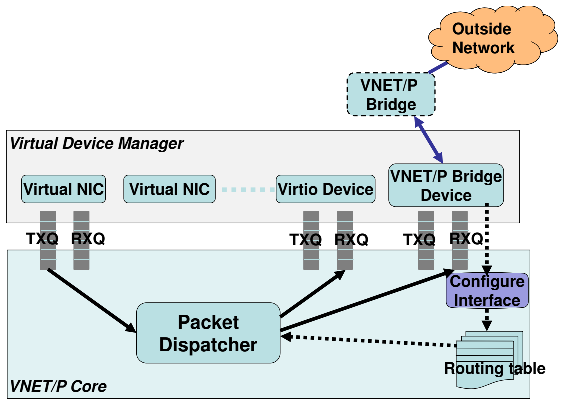


Figure 2: VNET/P core's internal logic.

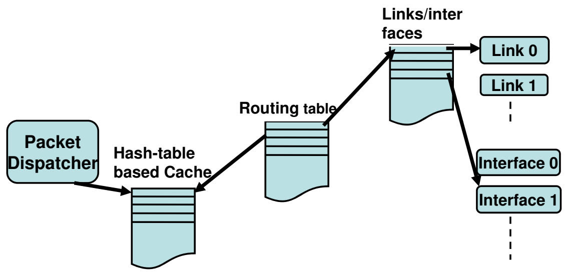


Figure 3: VNET/P routing data structures.

tion 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 application VM to Palacios with a single VM exit, and from Palacios to the application VM with a single VM exit/entry pair.

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 this machine and remote VNET on other machines. The VNET/P core's routing rules are dynamically configurable, through the control interface by the utilities that can be 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" for communication within high bandwidth/low latency environments.

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.

4.2 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 2.

Routing. To route Ethernet packets, VNET/P uses the routing logic and data structures shown in Figure 3. Routing rules are maintained in a routing table that is indexed by source and destination MAC addresses. Although this table structure only provides linear time lookups, it is important to note that a hash table-based routing cache is layered on top of the 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 net-

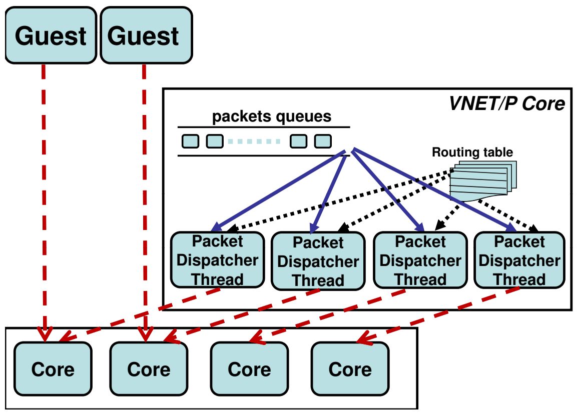


Figure 4: VNET/P running on a multicore system. The selection of how many, and which cores to use for packet dispatcher threads is made dynamically.

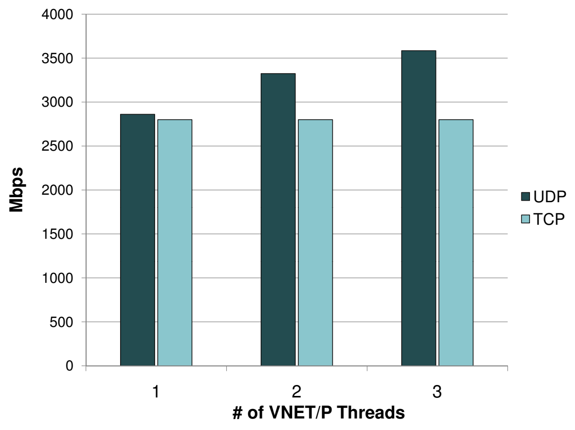


Figure 5: Early example of scaling of receive throughput by executing the VMM-based components of VNET/P on separate cores, and scaling the number of cores used. The ultimate on-wire MTU here is 1500 bytes.

work. 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 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 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 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 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 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.

Influenced by Sidecore [19], an additional optimization we developed was to offload in-VMM VNET/P processing, beyond packet dispatch, to an unused core or cores, thus making 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). Figure 5 is an example of the benefits of doing so for small MTU communication.

VNET/P switches between these two modes dynamically depending on the arrival rate of packets destined to or from the virtual NIC. For 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. If the rate is high enough when the guest transmits packets, then VNET/P switches the virtual NIC associated with that guest from guest-driven mode to VMM-driven mode. In other hand, if the rate drops low from the last recalculate period, it switches back from VMM-driven to guest-driven mode.

For a 1 Gbps network, guest-driven mode is sufficient to allow VNET/P to achieve the full native throughput. On a 10 Gbps network, VMM-driven mode is essential to move packets through the VNET/P core with near-native throughput.

4.3 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 [35], 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 performance, 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 distinctly 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 and receive accordingly. VNET/P can advertise the appropriate MTU.

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 Section 5, we analyze throughput using different MTU sizes.

4.4 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 a Linux 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:

- *Direct send* Here, 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.

- *Encapsulated send* Here 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.

For packet reception, the bridge uses two modes, simultaneously:

- *Direct receive* Here 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.

- *Encapsulated receive* Here, 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.

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

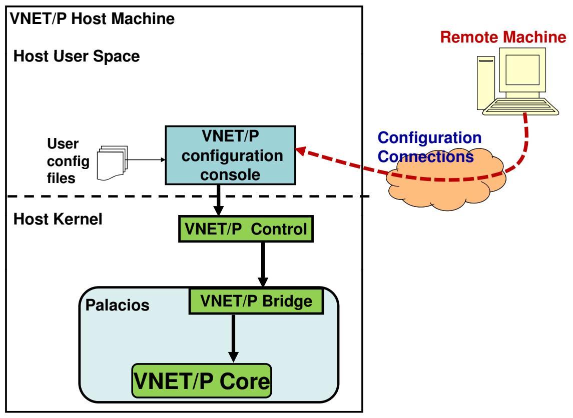


Figure 6: VNET/P overlay control interface. VNET/P is controlled via a userspace utilities, including a daemon that supports remote configuration using a protocol compatible with VNET/U. A VNET overlay is controlled via this protocol, and thus all the configuration, monitoring, and control algorithms previously described in the context of VNET/U apply to VNET/P as well.

Our performance evaluation focuses solely on encapsulated send and receive.

4.5 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 time. VNET/U already has user-level tools to support VNET, and, as we described in Section 2, 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 [41]). 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. The control interface is illustrated in Figure 6.

4.6 Performance-critical data paths and flows

Figure 7 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 4.3) 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 involved in the VM exit handling for I/O port reads and writes and for interrupt injection.

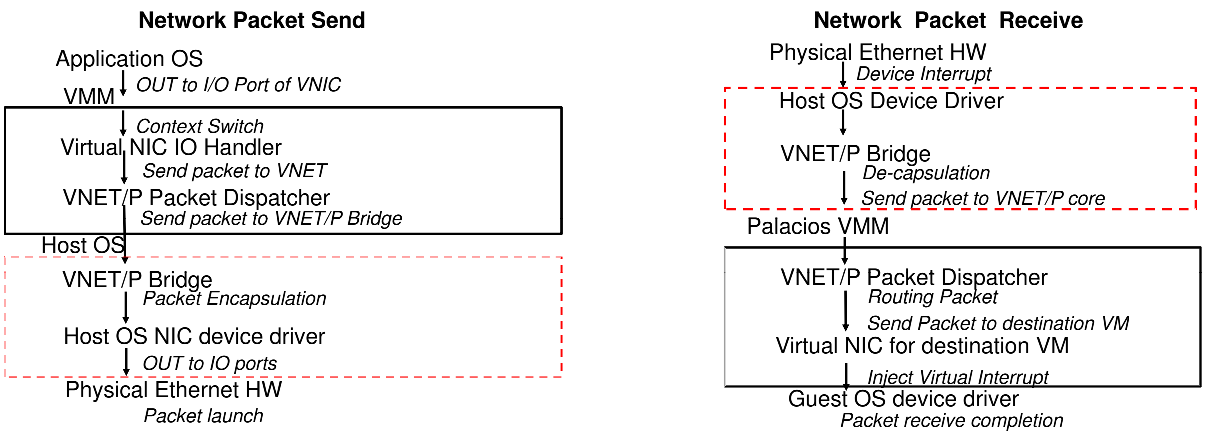


Figure 7: Performance-critical data paths and flows for packet transmission and reception. Solid boxed steps and components occur within the VMM itself, while dashed boxed steps and components occur in the host OS.

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 NIC on the host machine.

Note that while the packet is handed off multiple times, it is copied only once inside the VMM, from the send buffer (TXQ) of 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. Thus, 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. PERFORMANCE EVALUATION

The purpose of our performance evaluation is to determine how close VNET/P comes to native throughput and latency in the most demanding (lowest latency, highest throughput) hardware environments. We consider communication between two machines whose NICs are directly connected. 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.

Our evaluation of communication performance in this environment occurs at three levels. First, we benchmark the TCP and UDP bandwidth and latency. Second, we benchmark MPI using a widely used benchmark. Finally, we use basic communication benchmark included with a widely used HPC benchmark suite.

5.1 Testbed and configurations

Our testbed consists of two physical machines, which we call host machines, that have the following hardware:

- Quadcore 2.4 GHz X3430 Intel Xeon(tm) processors
- 8 GB RAM
- Broadcom NetXtreme II 1 Gbps Ethernet NIC (1000BASE-T)
- NetEffect NE020 10 Gbps Ethernet fiber optic NIC (10GBASE-SR) in a PCI-e slot

The Ethernet NICs of these machines are directly connected with twisted pair and fiber patch cables.

We considered the following two 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. We refer to the 1 and 10 Gbps results in this configuration as *Native-1G* and *Native-10G*, respectively.
- *VNET/P:* The VNET/P configuration corresponds to the architectural diagram given in Figure 1, with a single guest VM running on Palacios. The guest VM is configured with one virtio network device, 1 core, 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 identical to that in the Native configuration, with the exception that the virtio NIC drivers are loaded. The virtio MTU

is configured as its largest possible size (64K Bytes).[2] We refer to the 1 and 10 Gbps results in this configuration as *VNET/P-1G* and *VNET/P-10G*, respectively.

Performance measurements are made between identically configured machines. That is, *VNET/P-1G* refers to 1 Gbps communication between two machines configured as described in *VNET/P* above, and so on.

To assure accurate time measurements in the virtualized case, our 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.

Our 1 Gbps NIC only supports MTUs up to 1500 bytes, while our 10 Gbps NIC can support MTUs of up to 9000 bytes. We use these maximum sizes unless otherwise specified.

5.2 TCP and UDP microbenchmarks

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

UDP and TCP with a standard MTU

Figure 8 show the TCP throughput and UDP goodput achieved in each of our configurations on each NIC. For the 1Gbps network, host MTU is setup to 1500 bytes, and for 10Gbps network, host MTUs of 1500 bytes and 9000 bytes are both tested.

We 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. The UDP goodput that VNET/P is capable of, using a standard Ethernet MTU, is approximately 33 times that of VNET/U.

For TCP throughput, ttcp was configured to use a 256 KB socket buffer, and to communicate 40 MB writes are 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 while operating approximately 33 times faster than VNET/U.

UDP and TCP with a large MTU

We now consider TCP and UDP performance with 9000 byte jumbo frames our 10 Gbps NICs support. We adjusted the VNET/P MTU so that the ultimate encapsulated packets will fit into these frames without fragmentation. For TCP we configure ttcp to use writes of corresponding size, maximize the socket buffer size, and do 4 million writes. For UDP, we configure ttcp to use commensurately large packets sent as fast as possible for 60 seconds. The results are also shown in the Figure 8. We can see that performance increases across the board compared to the 1500 byte MTU results.

[2] An astute reader may find it quite curious that we would choose a guest MTU (64K) that is significantly larger than the host MTUs (1500/9000 bytes) into which we will ultimately encapsulate packets. This unusual choice, which optimizes performance, is due to a tradeoff between fragmentation/reassembly costs, and VM exit/entry and per-exit VNET/P processing costs. With a smaller guest MTU, fragmentation/reassembly could, of course, be reduced or even avoided. However, as the guest MTU shrinks, for a given volume of data to be transferred there will be more guest exits, each of which contributes a fixed cost. The right guest MTU will depend on the hardware and the host OS.

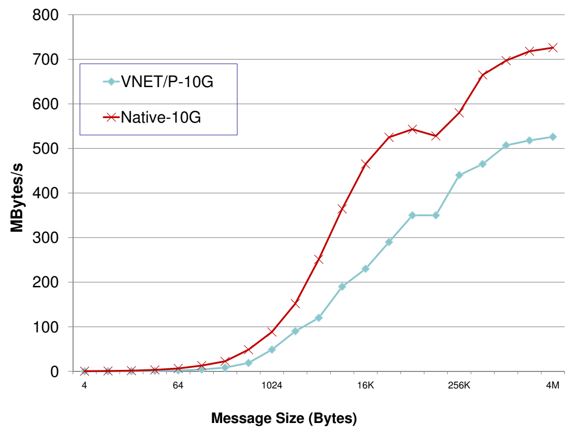


Figure 11: Intel MPI SendRecv microbenchmark showing bidirectional bandwidth as a function of message size for native and VNET/P on the 10 Gbps hardware with a host MTU of 9000. There is no reduction in performance compared to the unidirectional case.

Latency

Figure 9 shows the 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 over three times lower than that of VNET/U.

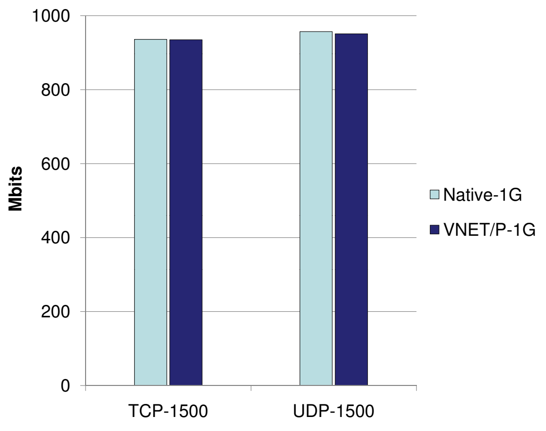
5.3 MPI microbenchmarks

Parallel programs for distributed memory computers are typically written to, or compiled to, the MPI interface standard [29]. We used the OpenMPI 1.3 [7] implementation in our evaluations. We measured the performance of MPI over VNET/P by employing the widely-used Intel MPI Benchmark Suite (IMB 3.2.2) [4], focusing on the point-to-point messaging performance. We compared the basic MPI latency and bandwidth achieved by VNET/P and native.

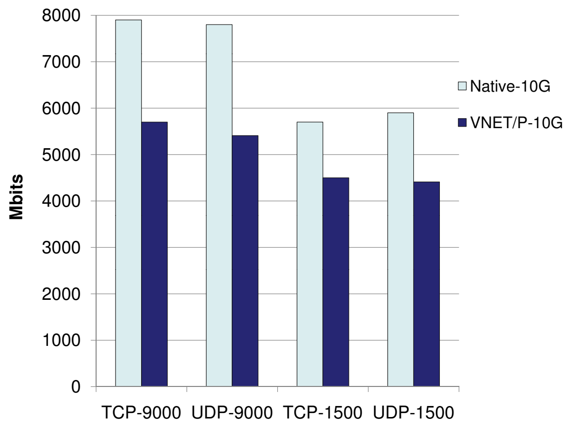
Figure 10 illustrates the latency and bandwidth reported by Intel MPI PingPong benchmark for our 10 Gbps configuration. Here, the latency measured is the one-way, end-to-end, application-level latency. That is, it is the time from when an an MPI send starts on one machine to when its matching MPI receive 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 shows 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 current VNET/P implementation can deliver an MPI latency of 55 μs and bandwidth of 520 MB/s on 10 Gbps Ethernet hardware.

Figure 11 shows the results of the MPI SendRecv microbenchmark in which each nodes simultaneously sends and receives. There

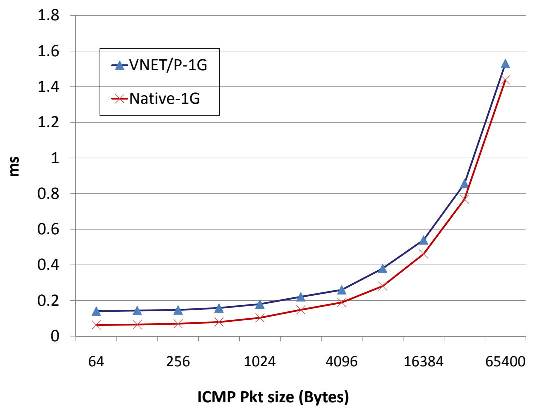


(a) 1 Gbps network with host MTU=1500 Bytes

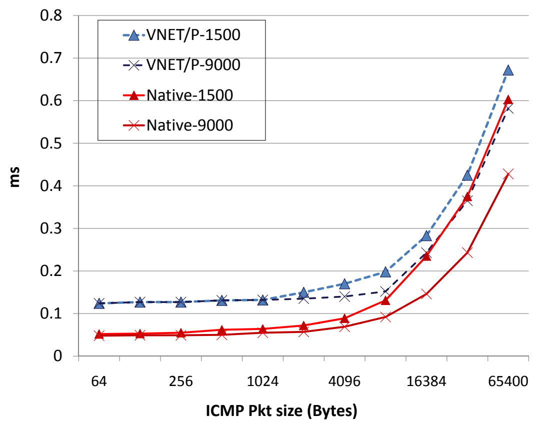


(b) 10 Gbps network (with Host MTU=1500, 9000 Bytes)

Figure 8: End-to-end TCP throughput and UDP goodput of VNET/P on 1 and 10 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=1500Bytes)



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

Figure 9: 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 $\sim$130 μs.

is no decrease in performance because VNET/P is able to recruit sufficient idle cores.

5.4 HPCC benchmark

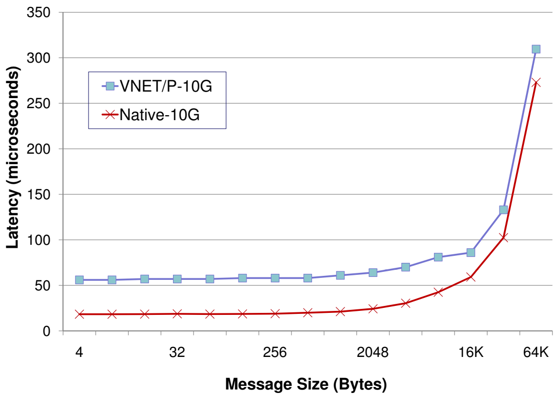
HPC Challenge (HPCC) [1] is a widely used benchmark suite for evaluating likely MPI application performance. Since our focus is on communication performance and we are testing with two machines, We evaluated VNET/P using the HPCC b_eff Latency and Bandwidth components.

Latency is determined via a ping pong benchmark executing between two processes, one on each node. The client process sends a message ("ping") to the server process, which bounces it back to the client ("pong"). Communication uses the basic MPI blocking send and receive functions. The ping-pong process is repeated until convergence. Bandwidth is measured by sending 2,000,000 byte messages repeatedly in a tight loop using the basic MPI blocking primitives. The average one-way latency and bandwidth are reported, along with minimums and maximums.

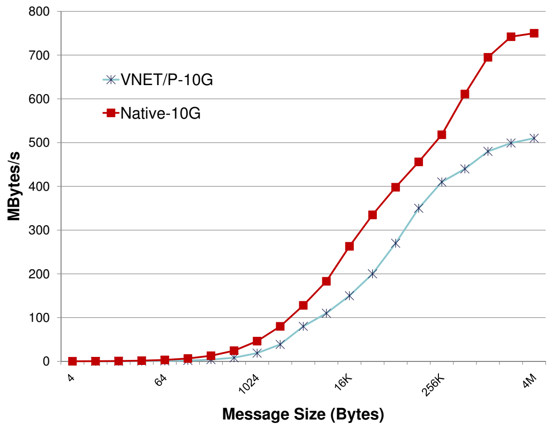
Figure 12 shows the results of the HPCC benchmarks. For the 1 Gbps network, VNET/P achieves the same bandwidth as native, while its latency is about 2 times larger. For the 10 Gbps network, VNET/P achieves a throughput of 502 MB/s, comparing to native performance of 718 MB/s, while VNET/P's latency is slightly more than 2 times higher.

6. VNET/P FOR INFINIBAND

In support of hardware independence, the 3rd goal of VNET articulated in Section 2.1, we have developed an implementation of VNET/P that allows guests that only support Ethernet NICs to be seamlessly run on top of an InfiniBand network, or to span InfiniBand networks and other networks. Regardless of the underlying networking hardware, the guests see a simple Ethernet LAN.



(a) One-way latency



(b) One-way bandwidth

Figure 10: Intel MPI PingPong microbenchmark showing one-way latency and bandwidth as a function of message size on the 10 Gbps hardware. The host MTU is set to 9000.

1 Gbps Network			
		Native	VNET/P
Message Length: 8	Latency (μs) min/avg/max	24.1/24.1/24.1	58.3/59.1/59.6
Message Length: 2000000	Bandwidth(MByte/s) min/avg/max	116.8/116.9/116.9	113.8/115.1/116.1
10 Gbps Network			
		Native	VNET/P
Message Length: 8	Latency (μs) min/avg/max	18.2/18.3/18.4	53.2/58.3/59.4
Message Length: 2000000	Bandwidth (MByte/s) min/avg/max	713.5/718.8/720.2	490.5/502.8/508.2

Figure 12: HPCC benchmark results giving one-way latency and bandwidth for MPI communication. We compare native performance and VNET/P performance on both 1 Gbps hardware (MTU=1500) and 10 Gbps hardware (MTU=9000).

Figure 13 shows the architecture of VNET/P over InfiniBand and can be compared and contrasted with the VNET/P over Ethernet architecture shown in Figure 1. Note the abstraction that the guest VMs see is identical: the guests still believe they are connected by a simple Ethernet network.

For the current Infiniband implementation, the host OS that is used is Sandia National Labs' Kitten lightweight kernel. Kitten has, by design, a minimal set of in-kernel services. For this reason, the VNET/P Bridge functionality is not implemented in the kernel, but rather in a privileged service VM called the Bridge VM that has direct access to the physical Infiniband device.

In place of encapsulating Ethernet packets in UDP packets for transmission to a remote VNET/P core, VNET/P's InfiniBand support simply maps Ethernet packets to InfiniBand frames. These frames are then transmitted through an InfiniBand queue pair accessed via the Linux IPoIB framework.

We conducted preliminary performance tests of VNET/P on InfiniBand using 8900 byte TCP payloads running on ttcp on a testbed similar to the one described in Section 5.1. Here, each node was a dual quadcore 2.3 GHz 2376 AMD Opteron machine with 32 GB of RAM and a Mellanox MT26428 InfiniBand NIC in a PCI-e slot. The Infiniband NICs were connected via a Mellanox MTS 3600 36-port 20/40Gbps InfiniBand switch.

It is important to point out that VNET/P over Infiniband is a work in progress and we present it here as a proof of concept. Nonetheless, on this testbed it achieved 4.0 Gbps end-to-end TCP throughput, compared to 6.5 Gbps when run natively on top of IP-over-InfiniBand in Reliable Connected (RC) mode.

7. CONCLUSION AND FUTURE WORK

We have described the VNET model of overlay networking in a distributed virtualized computing environment and our efforts in 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 our design and implementation of VNET for such environments. Its design goal is to achieve near-native throughput and latency on 1 and 10 Gbps Ethernet, InfiniBand, and other high performance interconnects.

To achieve performance, VNET/P relies on several key techniques and systems, including lightweight virtualization in the Palacios virtual machine monitor, high-performance I/O, and multicore overlay routing support. Together, these techniques enable VNET/P to provide a simple and flexible level 2 Ethernet network abstraction in a large range of systems no matter the actual underlying networking technology is. While our VNET/P implementation is tightly integrated into our Palacios virtual machine monitor, the principles involved could be used in other environments as well.

We are currently working to further enhance VNET/P's performance. One approach we are currently evaluating is injecting VNET/P

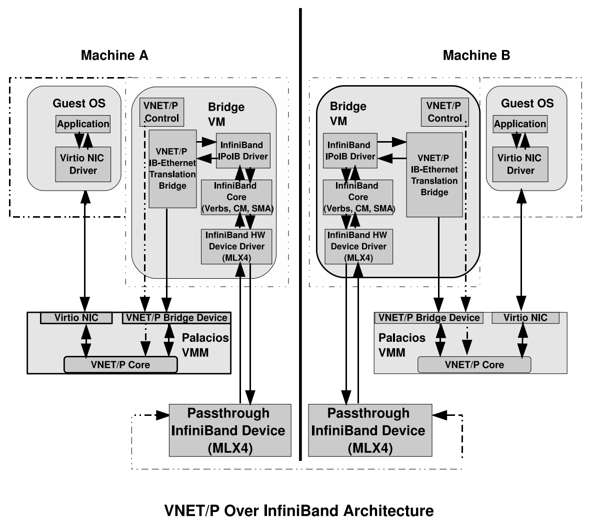

Machine A
Machine B
Guest OS
Application
Virtio NIC Driver
VNET/P Control
Bridge VM
VNET/P IB-Ethernet Translation Bridge
InfiniBand IPoIB Driver
InfiniBand Core (Verbs, CM, SMA)
InfiniBand HW Device Driver (MLX4)
Virtio NIC
VNET/P Bridge Device
Palacios VMM
VNET/P Core
Passthrough InfiniBand Device (MLX4)
VNET/P Over InfiniBand Architecture

Figure 13: The architecture of VNET/P over InfiniBand. Palacios is embedded in the Kitten lightweight kernel and forwards packets to/receives packets from a service VM called the bridge VM.

directly into the guest. For a symbiotic guest [21], the VMM could supply a driver module that internally implemented VNET/P functionality and had direct hardware access. This arrangement would allow a guest VM to route its network traffic through the VNET overlay without any VMM involvement. For a non-symbiotic guest, it may be possible to nonetheless forcibly inject a protected driver and restrict physical device access to only when that driver is run. We are also working on functionality enhancements of VNET/P, particularly broader support on InfiniBand and on the Cray SeaStar interconnect on XT-class supercomputers.

8. REFERENCES

[1] Hpc challenge benchmark. http://icl.cs.utk.edu/hpcc/.

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

[3] AMD Corporation. AMD64 virtualization codenamed "pacific" technology: Secure virtual machine architecture reference manual, May 2005.

[4] Corporation, I. Intel cluster toolkit 3.0 for linux. http://software.intel.com/en-us/articles/intel-mpi-benchmarks/.

[5] Evangelinos, C., and Hill, C. 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 2008)* (October 2008).

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

[7] Gabriel, E., Fagg, G. E., Bosilca, G., Angskun, T., Dongarra, J. J., Squyres, J. M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R. H., Daniel, D. J., Graham, R. L., and Woodall, T. S. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings of the 11th European PVM/MPI Users' Group Meeting* (September 2004), pp. 97–104.

[8] Ganguly, A., Agrawal, A., Boykin, P. O., and Figueiredo, R. 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 2006)* (April 2006).

[9] Garfinkel, S. An evaluation of amazon's grid computing services: Ec2, s3, and sqs. Tech. Rep. TR-08-07, Center for Research on Computation and Society, Harvard University, 2008.

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

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

[12] Gupta, A. *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.

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

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

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

[16] Intel Corporation. Intel virtualization technology specification for the ia-32 intel architecture, April 2005.

[17] Keahey, K., Foster, I., Freeman, T., and Zhang, X. Virtual workspaces: Achieving quality of service and quality of life in the grid. *Scientific Programming 13*, 3 (2005), 265–276.

[18] Kim, C., Caesar, M., and Rexford, J. Floodless in seattle: a scalable ethernet architecture for large enterprises. In *Proceedings of SIGCOMM 2008* (2008).

[19] Kumar, S., Raj, H., Schwan, K., and Ganev, I. 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).

[20] Lange, J., and Dinda, P. 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).

[21] Lange, J., and Dinda, P. Symcall: Symbiotic

virtualization through vmm-to-guest upcalls. In *Proceedings of the 2011 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)* (March 2011).

[22] LANGE, J., PEDRETTI, K., DINDA, P., BAE, C., BRIDGES, P., SOLTERO, P., AND MERRITT, A. 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).

[23] LANGE, J., PEDRETTI, K., HUDSON, T., DINDA, P., CUI, Z., XIA, L., BRIDGES, P., GOCKE, A., JACONETTE, S., LEVENHAGEN, M., AND BRIGHTWELL, R. Palacios and kitten: New high performance operating systems for scalable virtualized and native supercomputing. In *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2010)* (April 2010).

[24] LANGE, J., SUNDARARAJ, A., AND DINDA, P. Automatic dynamic run-time optical network reservations. In *Proceedings of the 14th IEEE International Symposium on High Performance Distributed Computing (HPDC)* (July 2005).

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

[26] LIN, B., SUNDARARAJ, A., AND DINDA, P. Time-sharing parallel applications with performance isolation and control. In *Proceedings of the 4th IEEE International Conference on Autonomic Computing (ICAC)* (June 2007). An extended version appears in the Journal of Cluster Computing, Volume 11, Number 3, September 2008.

[27] LIU, J., HUANG, W., ABALI, B., AND PANDA, D. High performance vmm-bypass i/o in virtual machines. In *Proceedings of the USENIX Annual Technical Conference* (May 2006).

[28] MERGEN, M. F., UHLIG, V., KRIEGER, O., AND XENIDIS, J. Virtualization for high-performance computing. *Operating Systems Review 40*, 2 (2006), 8–11.

[29] MESSAGE PASSING USER INTERFACE FORUM. MPI: a messsage-passing interface standard, version 2.2. Tech. rep., MPI Forum, September 2009.

[30] MYSORE, R. N., PAMBORIS, A., FARRINGTON, N., HUANG, N., MIRI, P., RADHAKRISHNAN, S., SUBRAMANYA, V., AND VAHDAT, A. Portland: A scalable fault-tolerant layer 2 data center network fabric. In *Proceedings of SIGCOMM 2009* (August 2009).

[31] NURMI, D., WOLSKI, R., GRZEGORZYK, C., OBERTELLI, G., SOMAN, S., YOUSEFF, L., AND ZAGORODNOV, D. The eucalyptus open-source cloud-computing system. In *Proceedings of the 9th IEEE/ACM International Symposium on Cluster Computing and the Grid* (May 2009).

[32] OSTERMANN, S., IOSUP, A., YIGITBASI, N., PRODAN, R., FAHRINGER, T., AND EPEMA, D. An early performance analysis of cloud computing services for scientific computing. Tech. Rep. PDS2008-006, Delft University of Technology, Parallel and Distributed Systems Report Series, December 2008.

[33] RAJ, H., AND SCHWAN, K. 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).

[34] RIESEN, R., BRIGHTWELL, R., BRIDGES, P., HUDSON, T., MACCABE, A., WIDENER, P., AND FERREIRA, K. Designing and implementing lightweight kernels for capability computing. *Concurrency and Computation: Practice and Experience 21*, 6 (April 2009), 793–817.

[35] RUSSELL, R. virtio: towards a de-facto standard for virtual i/o devices. *SIGOPS Oper. Syst. Rev. 42*, 5 (2008), 95–103.

[36] RUTH, P., JIANG, X., XU, D., AND GOASGUEN, S. Towards virtual distributed environments in a shared infrastructure. *IEEE Computer* (May 2005).

[37] RUTH, P., MCGACHEY, P., JIANG, X., AND XU, D. Viocluster: Virtualization for dynamic computational domains. In *Proceedings of the IEEE International Conference on Cluster Computing (Cluster)* (September 2005).

[38] SHOYKHET, A., LANGE, J., AND DINDA, P. Virtuoso: A system for virtual machine marketplaces. Tech. Rep. NWU-CS-04-39, Department of Computer Science, Northwestern University, July 2004.

[39] SUNDARARAJ, A. *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.

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

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

[42] SUNDARARAJ, A., SANGHI, M., LANGE, J., AND DINDA, P. An optimization problem in adaptive virtual environmnets. In *Proceedings of the seventh Workshop on Mathematical Performance Modeling and Analysis (MAMA)* (June 2005).

[43] UHLIG, R., NEIGER, G., RODGERS, D., SANTONI, A., MARTIN, F., ANDERSON, A., BENNETTT, S., KAGI, A., LEUNG, F., AND SMITH, L. Intel virtualization technology. *IEEE Computer* (May 2005), 48–56.

[44] WALKER, E. Benchmarking amazon ec2 for high performance scientific computing. *USENIX ;login: 3*, 8 (October 2008), 18–23.

[45] WOLINSKY, D., LIU, Y., JUSTE, P. S., VENKATASUBRAMANIAN, G., AND FIGUEIREDO, R. On the design of scalable, self-configuring virtual networks. In *Proceedings of 21st ACM/IEEE International Conference ofr High Performance Computing, Networking, Storage, and Analysis (SuperComputing / SC 2009)* (November 2009).

[46] XIA, L., LANGE, J., DINDA, P., AND BAE, C. Investigating virtual passthrough i/o on commodity devices. *Operating Systems Review 43*, 3 (July 2009). Initial version appeared at WIOV 2008.