NetVM: High Performance and Flexible Networking Using Virtualization on Commodity Platforms

Jinho Hwang, ***The George Washington University;*** **K. K. Ramakrishnan,** ***Rutgers University;***
Timothy Wood, ***The George Washington University***

https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/hwang

This paper is included in the Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI '14).

April 2–4, 2014 • Seattle, WA, USA

ISBN 978-1-931971-09-6

Open access to the Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI '14) is sponsored by USENIX

NetVM: High Performance and Flexible Networking using Virtualization on Commodity Platforms

Jinho Hwang[†] K. K. Ramakrishnan[*] Timothy Wood[†]
[†]*The George Washington University* [*]*WINLAB, Rutgers University*

Abstract

NetVM brings virtualization to the Network by enabling high bandwidth network functions to operate at near line speed, while taking advantage of the flexibility and customization of low cost commodity servers. NetVM allows customizable data plane processing capabilities such as firewalls, proxies, and routers to be embedded within virtual machines, complementing the control plane capabilities of Software Defined Networking. NetVM makes it easy to dynamically scale, deploy, and reprogram network functions. This provides far greater flexibility than existing purpose-built, sometimes proprietary hardware, while still allowing complex policies and full packet inspection to determine subsequent processing. It does so with dramatically higher throughput than existing software router platforms.

NetVM is built on top of the KVM platform and Intel DPDK library. We detail many of the challenges we have solved such as adding support for high-speed inter-VM communication through shared huge pages and enhancing the CPU scheduler to prevent overheads caused by inter-core communication and context switching. NetVM allows true zero-copy delivery of data to VMs both for packet processing and messaging among VMs within a trust boundary. Our evaluation shows how NetVM can compose complex network functionality from multiple pipelined VMs and still obtain throughputs up to 10 Gbps, an improvement of more than 250% compared to existing techniques that use SR-IOV for virtualized networking.

1 Introduction

Virtualization has revolutionized how data center servers are managed by allowing greater flexibility, easier deployment, and improved resource multiplexing. A similar change is beginning to happen within communication networks with the development of virtualization of network functions, in conjunction with the use of software defined networking (SDN). While the migration of network functions to a more software based infrastructure is likely to begin with edge platforms that are more "control plane" focused, the flexibility and cost-effectiveness obtained by using common off-the-shelf hardware and systems will make migration of other network functions attractive. One main deterrent is the achievable performance and scalability of such virtualized platforms compared to purpose-built (often proprietary) networking hardware or middleboxes based on custom ASICs.

Middleboxes are typically hardware-software packages that come together on a special-purpose appliance, often at high cost. In contrast, a high throughput platform based on virtual machines (VMs) would allow network functions to be deployed dynamically at nodes in the network with low cost. Further, the shift to VMs would let businesses run network services on existing cloud platforms, bringing multiplexing and economy of scale benefits to network functionality. Once data can be moved to, from and between VMs at line rate for all packet sizes, we approach the long-term vision where the line between data centers and network resident "boxes" begins to blur: both software and network infrastructure could be developed, managed, and deployed in the same fashion.

Progress has been made by network virtualization standards and SDN to provide greater configurability in the network [1–4]. SDN improves flexibility by allowing software to manage the network control plane, while the performance-critical data plane is still implemented with proprietary network hardware. SDN allows for new flexibility in how data is forwarded, but the focus on the control plane prevents dynamic management of many types of network functionality that rely on the data plane, for example the information carried in the packet payload.

This limits the types of network functionality that can be "virtualized" into software, leaving networks to continue to be reliant on relatively expensive network appliances that are based on purpose-built hardware.

Recent advances in network interface cards (NICs) allow high throughput, low-latency packet processing using technologies like Intel's Data Plane Development Kit (DPDK) [5]. This software framework allows end-host applications to receive data directly from the NIC, eliminating overheads inherent in traditional interrupt driven OS-level packet processing. Unfortunately, the DPDK framework has a somewhat restricted set of options for support of virtualization, and on its own cannot support the type of flexible, high performance functionality that network and data center administrators desire.

To improve this situation, we have developed NetVM, a platform for running complex network functionality at line-speed (10Gbps) using commodity hardware. NetVM takes advantage of DPDK's high throughput packet processing capabilities, and adds to it abstractions that enable in-network services to be flexibly created,

chained, and load balanced. Since these "virtual bumps" can inspect the full packet data, a much wider range of packet processing functionality can be supported than in frameworks utilizing existing SDN-based controllers manipulating hardware switches. As a result, NetVM makes the following innovations:

1. A virtualization-based platform for flexible network service deployment that can meet the performance of customized hardware, especially those involving complex packet processing.
2. A shared-memory framework that truly exploits the DPDK library to provide zero-copy delivery to VMs and between VMs.
3. A hypervisor-based switch that can dynamically adjust a flow's destination in a state-dependent (e.g., for intelligent load balancing) and/or data-dependent manner (e.g., through deep packet inspection).
4. An architecture that supports high speed inter-VM communication, enabling complex network services to be spread across multiple VMs.
5. Security domains that restrict packet data access to only trusted VMs.

We have implemented NetVM using the KVM and DPDK platforms—all the aforementioned innovations are built on the top of DPDK. Our results show how NetVM can compose complex network functionality from multiple pipelined VMs and still obtain line rate throughputs of 10Gbps, an improvement of more than 250% compared to existing SR-IOV based techniques. We believe NetVM will scale to even higher throughputs on machines with additional NICs and processing cores.

2 Background and Motivation

This section provides background on the challenges of providing flexible network services on virtualized commodity servers.

2.1 Highspeed COTS Networking

Software routers, SDN, and hypervisor based switching technologies have sought to reduce the cost of deployment and increase flexibility compared to traditional network hardware. However, these approaches have been stymied by the performance achievable with commodity servers [6–8]. These limitations on throughput and latency have prevented software routers from supplanting custom designed hardware [9–11].

There are two main challenges that prevent commercial off-the-shelf (COTS) servers from being able to process network flows at line speed. First, network packets arrive at unpredictable times, so interrupts are generally used to notify an operating system that data is ready for processing. However, interrupt handling can be expensive because modern superscalar processors use

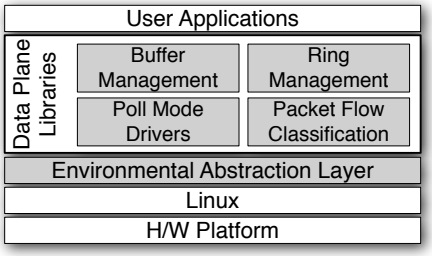


Figure 1: DPDK's run-time environment over Linux.

long pipelines, out-of-order and speculative execution, and multi-level memory systems, all of which tend to increase the penalty paid by an interrupt in terms of cycles [12, 13]. When the packet reception rate increases further, the achieved (receive) throughput can drop dramatically in such systems [14]. Second, existing operating systems typically read incoming packets into kernel space and then copy the data to user space for the application interested in it. These extra copies can incur an even greater overhead in virtualized settings, where it may be necessary to copy an additional time between the hypervisor and the guest operating system. These two sources of overhead limit the the ability to run network services on commodity servers, particularly ones employing virtualization [15, 16].

The Intel DPDK platform tries to reduce these overheads by allowing user space applications to directly poll the NIC for data. This model uses Linux's huge pages to pre-allocate large regions of memory, and then allows applications to DMA data directly into these pages. Figure 1 shows the DPDK architecture that runs in the application layer. The poll mode driver allows applications to access the NIC card directly without involving kernel processing, while the buffer and ring management systems resemble the memory management systems typically employed within the kernel for holding `sk_buffs`.

While DPDK enables high throughput user space applications, it does not yet offer a complete framework for constructing and interconnecting complex network services. Further, DPDK's passthrough mode that provides direct DMA to and from a VM can have significantly lower performance than native IO[1]. For example, DPDK supports Single Root I/O Virtualization (SR-IOV[2]) to allow multiple VMs to access the NIC, but packet "switching" (i.e., demultiplexing or load balancing) can only be performed based on the L2 address. As depicted in Figure 2(a), when using SR-IOV, packets are switched on

[1] Until Sandy-bridge, the performance was close to half of native performance, but with the next generation Ivy-bridge processor, the claim has been that performance has improved due to IOTLB (I/O Translation Lookaside Buffer) super page support [17]. But no performance results have been released.

[2] SR-IOV makes it possible to logically partition a NIC and expose to each VM a separate PCI-based NIC called a "Virtual Function" [18].

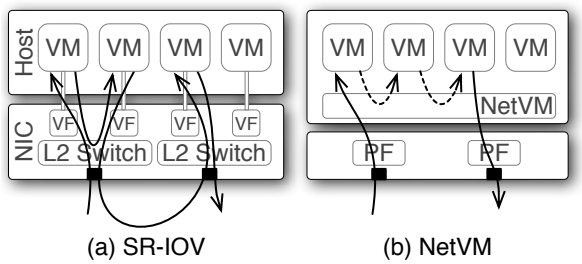


Figure 2: DPDK uses per-port switching with SR-IOV, whereas NetVM provides a global switch in the hypervisor and shared-memory packet transfer (dashed lines).

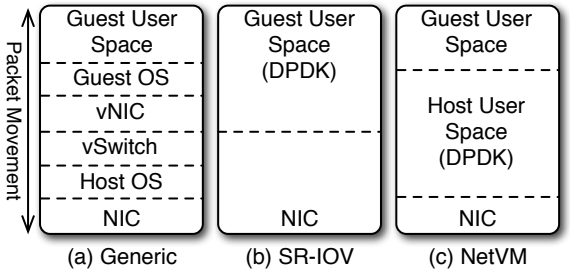


Figure 3: Architectural Differences for Packet Delivery in Virtualized Platform.

a per-port basis in the NIC, which means a second data copy is required if packets are forwarded between VMs on a shared port. Even worse, packets must go out of the host and come back via an external switch to be transmitted to a VM that is connected to another port's virtual function. Similar overheads appear for other VM switching platforms, e.g., Open vSwitch [19] and VMware's vNetwork distributed switch [20]. We seek to overcome this limitation in NetVM by providing a flexible switching capability without copying packets as shown in Figure 2(b). This improves performance of communication between VMs, which plays an important role when chained services are deployed.

Intel recently released an integration of DPDK and Open vSwitch [21] to reduce the limitations of SR-IOV switching. However, the DPDK vSwitch still requires copying packets between the hypervisor and the VM's memory, and does not support directly-chained VM communication. NetVM's enhancements go beyond DPDK vSwitch by providing a framework for flexible state- or data-dependent switching, efficient VM communication, and security domains to isolate VM groups.

2.2 Flexible Network Services

While platforms like DPDK allow for much faster processing, they still have limits on the kind of flexibility they can provide, particularly for virtual environments. The NIC based switching supported by DPDK + SR-IOV is not only expensive, but is limited because the NIC only has visibility into Layer 2 headers. With current techniques, each packet with a distinct destination MAC can be delivered to a different destination VM. However, in a network resident box (such as a middlebox acting as a firewall, a proxy, or even if the COTS platform is acting as a router), the destination MAC of incoming packets is the same. While advances in NIC design could reduce these limitations, a hardware based solution will never match the flexibility of a software-based approach.

By having the hypervisor perform the initial packet switching, NetVM can support more complex and dynamic functionality. For example, each application that supports a distinct function may reside in a separate VM, and it may be necessary to exploit flow classification to properly route packets through VMs based on mechanisms such as shallow (header-based) or deep (data-based) packet analysis. At the same time, NetVM's switch may use state-dependent information such as VM load levels, time of day, or dynamically configured policies to control the switching algorithm. Delivery of packets based on such rules is simply not feasible with current platforms.

2.3 Virtual Machine Based Networking

Network providers construct overall network functionality by combining middleboxes and network hardware that typically have been built by a diverse set of vendors. While NetVM can enable fast packet processing in software, it is the use of virtualization that will permit this diverse set of services to "play nice" with each other—virtualization makes it trivial to encapsulate a piece of software and its OS dependencies, dramatically simplifying deployment compared to running multiple processes on one bare-metal server. Running these services within VMs also could permit user-controlled network functions to be deployed into new environments such as cloud computing platforms where VMs are the norm and isolation between different network services would be crucial.

The consolidation and resource management benefits of virtualization are also well known. Unlike hardware middleboxes, VMs can be instantiated on demand when and where they are needed. This allows NetVM to multiplex one server for several related network functions, or to dynamically spawn VMs where new services are needed. Compared to network software running on bare metal, using a VM for each service simplifies resource allocation and improves performance isolation. These characteristics are crucial for network services that often have strict performance requirements.

3 System Design

Figure 3 compares two existing, commonly implemented network virtualization techniques against NetVM. In the first case, representing traditional virtualization plat-

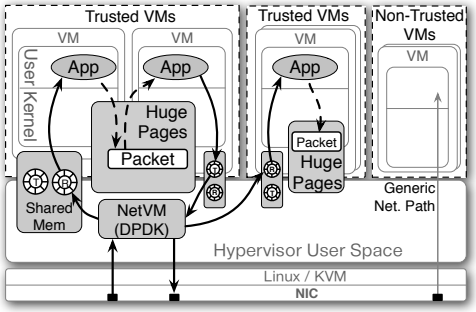


Figure 4: NetVM only requires a simple descriptor to be copied via shared memory (solid arrows), which then gives the VM direct access to packet data stored in huge pages (dashed arrow).

forms, packets arrive at the NIC and are copied into the hypervisor. A virtual switch then performs L2 (or a more complex function, based on the full 5-tuple packet header) switching to determine which VM is the recipient of the packet and notifies the appropriate virtual NIC. The memory page containing the packet is then either copied or granted to the Guest OS, and finally the data is copied to the user space application. Not surprisingly, this process involves significant overhead, preventing line-speed throughput.

In the second case (Figure 3(b)), SR-IOV is used to perform L2 switching on the NIC itself, and data can be copied directly into User Space of the appropriate VM. While this minimizes data movement, it does come at the cost of limited flexibility in how packets are routed to the VM, since the NIC must be configured with a static mapping and packet header information other than the MAC address cannot be used for routing.

The architecture of NetVM is shown in Figure 3(c). It does not rely on SR-IOV, instead allowing a user space application in the hypervisor to analyze packets and decide how to forward them. However, rather than copy data to the Guest, we use a shared memory mechanism to directly allow the Guest user space application to read the packet data it needs. This provides both flexible switching and high performance.

3.1 Zero-Copy Packet Delivery

Network providers are increasingly deploying complex services composed of routers, proxies, video transcoders, etc., which NetVM could consolidate onto a single host. To support fast communication between these components, NetVM employs two communication channels to quickly move data as shown in Figure 4. The first is a small, shared memory region (shared between the hypervisor and each individual VM) that is used to transmit packet descriptors. The second is a huge page region shared with a group of trusted VMs that allows chained applications to directly read or write packet data. Memory sharing through a "grant" mechanism is commonly used to transfer control of pages between the hypervisor and guest; by expanding this to a region of memory accessible by all trusted guest VMs, NetVM can enable efficient processing of flows traversing multiple VMs.

NetVM Core, running as a DPDK enabled user application, polls the NIC to read packets directly into the huge page area using DMA. It decides where to send each packet based on information such as the packet headers, possibly content, and/or VM load statistics. NetVM inserts a descriptor of the packet in the ring buffer that is setup between the individual destination VM and hypervisor. Each individual VM is identified by a "role number"−a representation of each network function, that is assigned by the VM manager. The descriptor includes a mbuf location (equivalent to a sk_buff in the Linux kernel) and huge page offset for packet reception. When transmitting or forwarding packets, the descriptor also specifies the action (transmit through the NIC, discard, or forward to another VM) and role number (i.e., the destination VM role number when forwarding). While this descriptor data must be copied between the hypervisor and guest, it allows the guest application to then directly access the packet data stored in the shared huge pages.

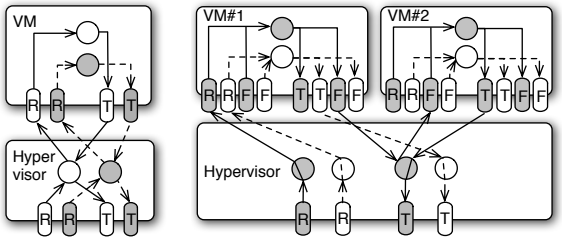
After the guest application (typically implementing some form of network functionality like a router or firewall) analyzes the packet, it can ask NetVM to forward the packet to a different VM or transmit it over the network. Forwarding simply repeats the above process—NetVM copies the descriptor into the ring buffer of a different VM so that it can be processed again; the packet data remains in place in the huge page area and never needs to be copied (although it can be independently modified by the guest applications if desired).

3.2 Lockless Design

Shared memory is typically managed with locks, but locks inevitably degrade performance by serializing data accesses and increasing communication overheads. This is particularly problematic for high-speed networking: to maintain full 10 Gbps throughput independent of packet size, a packet must be processed within 67.2 ns [22], yet context switching for a contested lock takes on the order of micro-seconds [23, 24], and even an uncontested lock operation may take tens of nanoseconds [25]. Thus a single context switch could cause the system to fall behind, and thus may result in tens of packets being dropped.

We avoid these issues by having parallelized queues with dedicated cores that service them. When working with NICs that have multiple queues and Receive Side Scaling (RSS) capability[1], the NIC receives pack-

[1] Modern NICs support RSS, a network driver technology to allow packet receive processing to be load balanced across multiple processors or cores [26].



(a) Single VM (b) Multiple VMs (Inter-VM)

Figure 5: Lockless and NUMA-Aware Queue/Thread Management (R = Receive Queue, T = Transmit Queue, and F = Forward Queue).

ets from the link and places them into one of several flow queues based on a configurable (usually an n-tuple) hash [27]. NetVM allows only two threads to manipulate this shared circular queue—the (producer) DPDK thread run by a core in the hypervisor and the (consumer) thread in the guest VM that performs processing on the packet. There is only a single producer and a single consumer, so synchronization is not required since neither will read or write simultaneously to the same region.

Our approach eliminates the overhead of locking by dedicating cores to each queue. This still permits scalability, because we can simply create additional queues (each managed by a pair of threads/cores). This works with the NIC's support for RSS, since incoming flows can automatically be load balanced across the available queues. Note that synchronization is not required to manage the huge page area either, since only one application will ever have control of the descriptor containing a packet's address.

Figure 5(a) depicts how two threads in a VM deliver packets without interrupting each other. Each core (marked as a circle) in the hypervisor receives packets from the NIC and adds descriptors to the tail of its own queue. The guest OS also has two dedicated cores, each of which reads from the head of its queue, performs processing, and then adds the packet to a transmit queue. The hypervisor reads descriptors from the tail of these queues and causes the NIC to transmit the associated packets. This thread/queue separation guarantees that only a single entity accesses the data at a time.

3.3 NUMA-Aware Design

Multi-processor systems exhibit NUMA characteristics, where memory access time depends on the memory location relative to a processor. Having cores on different sockets access memory that maps to the same cache line should be avoided, since this will cause expensive cache invalidation messages to ping pong back and forth between the two cores. As a result, ignoring the NUMA aspects of modern servers can cause significant performance degradation for latency sensitive tasks like network processing [28, 29].

Quantitatively, a last-level-cache (L3) hit on a 3GHz Intel Xeon 5500 processor takes up to 40 cycles, but the miss penalty is up to 201 cycles [30]. Thus if two separate sockets in NetVM end up processing data stored in nearby memory locations, the performance degradation can potentially be up to five times, since cache lines will end up constantly being invalidated.

Fortunately, NetVM can avoid this issue by carefully allocating and using huge pages in a NUMA-aware fashion. When a region of huge pages is requested, the memory region is divided uniformly across all sockets, thus each socket allocates a total of *(total huge page size / number of sockets)* bytes of memory from DIMMs that are local to the socket. In the hypervisor, NetVM then creates the same number of receive/transmit threads as there are sockets, and each is used only to process data in the huge pages local to that socket. The threads inside the guest VMs are created and pinned to the appropriate socket in a similar way. This ensures that as a packet is processed by either the host or the guest, it always stays in a local memory bank, and cache lines will never need to be passed between sockets.

Figure 5 illustrates how two sockets (gray and white) are managed. That is, a packet handled by gray threads is never moved to white threads, thus ensuring fast memory accesses and preventing cache coherency overheads. This also shows how NetVM pipelines packet processing across multiple cores—the initial work of handling the DMAed data from the NIC is performed by cores in the hypervisor, then cores in the guest perform packet processing. In a multi-VM deployment where complex network functionality is being built by chaining together VMs, the pipeline extends to an additional pair of cores in the hypervisor that can forward packets to cores in the next VM. Our evaluation shows that this pipeline can be extended as long as there are additional cores to perform processing (up to three separate VMs in our testbed).

3.4 Huge Page Virtual Address Mapping

While each individual huge page represents a large contiguous memory area, the full huge page region is spread across the physical memory both because of the per-socket allocations described in Section 3.3, and because it may be necessary to perform multiple huge page allocations to reach the desired total size if it is bigger than the default unit of huge page size−the default unit size can be found under /proc/meminfo. This poses a problem since the address space layout in the hypervisor is not known by the guest, yet guests must be able to find packets in the shared huge page region based on the address in the descriptor. Thus the address where a packet is placed by the NIC is only meaningful to the hypervisor; the address must be translated so that the guest will

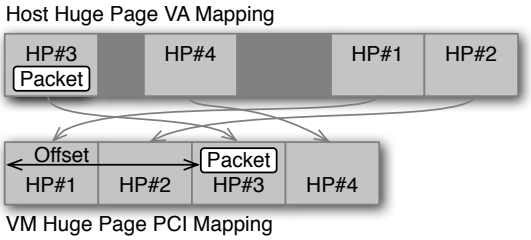


Figure 6: The huge pages spread across the host's memory must be contiguously aligned within the VM. NetVM must be able to quickly translate the address of a new packet from the host's virtual address space to an offset within the VM's address space.

be able to access it in the shared memory region. Further, looking up these addresses must be as fast as possible in order to perform line-speed packet processing.

NetVM overcomes the first challenge by mapping the huge pages into the guest in a contiguous region, as shown in Figure 6. NetVM exposes these huge pages to guest VMs using an emulated PCI device. The guest VM runs a driver that polls the device and maps its memory into user space, as described in Section 4.3. In effect, this shares the entire huge page region among all trusted guest VMs and the hypervisor. Any other untrusted VMs use a regular network interface through the hypervisor, which means they are not able to see the packets received from NetVM.

Even with the huge pages appearing as a contiguous region in the guest's memory space, it is non-trivial to compute where a packet is stored. When NetVM DMAs a packet into the huge page area, it receives a descriptor with an address *in the hypservisor's virtual address space*, which is meaningless to the guest application that must process the packet. While it would be possible to scan through the list of allocated huge pages to determine where the packet is stored, that kind of processing is simply too expensive for high-speed packet rates because every packet needs to go through this process. To resolve this problem, NetVM uses only bit operations and precomputed lookup tables; our experiments show that this improves throughput by up to 10% (with 8 huge pages) and 15% (with 16 huge pages) in the worst case compared to a naive lookup.

When a packet is received, we need to know which huge page it belongs to. Firstly, we build up an index map that converts a packet address to a huge page index. The index is taken from the upper 8 bits of its address (31^{st} bit to 38^{th} bit). The first 30 bits are the offset in the corresponding huge page, and the rest of the bits (left of the 38^{th} bit) can be ignored. We denote this function as $IDMAP(h) = (h >> 30)\&0xFF$, where h is a memory address. This value is then used as an index into an array $HMAP[i]$ to determine the huge page number.

To get the address base (i.e., a starting address of each huge page in the ordered and aligned region) of the huge page where the packet belongs to, we need to establish an accumulated address base. If all the huge pages have the same size, we do not need this address base−instead, just multiplying is enough, but since there can be different huge page sizes, we need to keep track of an accumulated address base. A function $HIGH(i)$ keeps a starting address of each huge page index i. Lastly, the residual address is taken from last 30 bits of a packet address using $LOW(a) = a\&0x3FFFFFFF$. $OFFSET(p) = HIGH(HMAP[IDMAP(p)]) \mid LOW(p)$ returns an address offset of contiguous huge pages in the emulated PCI.

3.5 Trusted and Untrusted VMs

Security is a key concern in virtualized cloud platforms. Since NetVM aims to provide zero-copy packet transmission while also having the flexibility to steer flows between cooperating VMs, it shares huge pages assigned in the hypervisor with multiple guest VMs. A malicious VM may be able to guess where the packets are in this shared region to eavesdrop or manipulate traffic for other VMs. Therefore, there must be a clear separation between trusted VMs and non-trusted VMs. NetVM provides a group separation to achieve the necessary security guarantees. When a VM is created, it is assigned to a trust group, which determines what range of memory (and thus which packets) it will have access to.

While our current implementation supports only trusted or untrusted VMs, it is possible to subdivide this further. Prior to DMAing packet data into a huge page, DPDK's classification engine can perform a shallow analysis of the packet and decide which huge page memory pool to copy it to. This would, for example, allow traffic flows destined for one cloud customer to be handled by one trust group, while flows for a different customer are handled by a second NetVM trust group on the same host. In this way, NetVM enables not only greater flexibility in network function virtualization, but also greater security when multiplexing resources on a shared host.

Figure 4 shows a separation between trusted VM groups and a non-trusted VM. Each trusted VM group gets its own memory region, and each VM gets a ring buffer for communication with NetVM. In constrast, non-trusted VMs only can use generic network paths such as those in Figure 3 (a) or (b).

4 Implementation Details

NetVM's implementation includes the NetVM Core Engine (the DPDK application running in the hypervisor), a NetVM manager, drivers for an emulated PCI device, modifications to KVM's CPU allocation policies, and NetLib (our library for building in-network functional-

ity in VM's userspace). Our implementation is built on QEMU 1.5.0 (KVM included), and DPDK 1.4.1.

KVM and QEMU allow a regular Linux host to run one or more VMs. Our functionality is split between code in the guest VM, and code running in user space of the host operating system. We use the terms host operating system and hypervisor interchangeably in this discussion.

4.1 NetVM Manager

The NetVM manager runs in the hypervisor and provides a communication channel so that QEMU can pass information to the NetVM core engine about the creation and destruction of VMs, as well as their trust level. When the NetVM manager starts, it creates a server socket to communicate with QEMU. Whenever QEMU starts a new VM, it connects to the socket to ask the NetVM Core to initialize the data structures and shared memory regions for the new VM. The connection is implemented with a socket-type chardev with "-chardev socket,path=<path>,id=<id>" in the VM configuration. This is a common approach to create a communication channel between a VM and an application running in the KVM host, rather than relying on hypervisor-based messaging [31].

NetVM manager is also responsible for storing the configuration information that determines VM trust groups (i.e., which VMs should be able to connect to NetVM Core) and the switching rules. These rules are passed to the NetVM Core Engine, which implements these policies.

4.2 NetVM Core Engine

The NetVM Core Engine is a DPDK userspace application running in the hypervisor. NetVM Core is initialized with user settings such as the processor core mapping, NIC port settings, and the configuration of the queues. These settings determine how many queues are created for receiving and transmitting packets, and which cores are allocated to each VM for these tasks. NetVM Core then allocates the Huge Page region and initializes the NIC so it will DMA packets into that area when polled.

The NetVM core engine has two roles: the first role is to receive packets and deliver/switch them to VMs (using zero-copy) following the specified policies, and the other role is to communicate with the NetVM manager to synchronize information about new VMs. The main control loop first polls the NIC and DMAs packets to huge pages in a burst (batch), then for each packet, NetVM decides which VM to notify. Instead of copying a packet, NetVM creates a tiny packet descriptor that contains the huge page address, and puts that into the private shared ring buffer (shared between the VM and NetVM Core). The actual packet data is accessible to the VM via shared memory, accessible over the emulated PCI device described below.

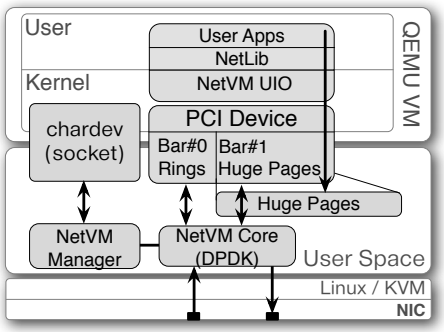


Figure 7: NetVM's architecture spans the guest and host systems; an emulated PCI device is used to share memory between them.

4.3 Emulated PCI

QEMU and KVM do not directly allow memory to be shared between the hypervisor and VMs. To overcome this limitation, we use an emulated PCI device that allows a VM to map the device's memory—since the device is written in software, this memory can be redirected to any memory location owned by the hypervisor. NetVM needs two seperate memory regions: a private shared memory (the address of which is stored in the device's BAR#0 register) and huge page shared memory (BAR#1). The private shared memory is used as ring buffers to deliver the status of user applications (VM $\rightarrow$ hypervisor) and packet descriptors (bidirectional). Each VM has this individual private shared memory. The huge page area, while not contiguous in the hypervisor, must be mapped as one contiguous chunk using the memory_region_add_subregion function. We illustrated how the huge pages map to virtual addresses, earlier in Section 3.4. In our current implementation, all VMs access the same shared huge page region, although this could be relaxed as discussed in 3.5.

Inside a guest VM that wishes to use NetVM's high-speed IO, we run a front-end driver that accesses this emulated PCI device using Linux's Userspace I/O framework (UIO). UIO was introduced in Linux 2.6.23 and allows device drivers to be written almost entirely in userspace. This driver maps the two memory regions from the PCI device into the guest's memory, allowing a NetVM user application, such as a router or firewall, to directly work with the incoming packet data.

4.4 NetLib and User Applications

Application developers do not need to know anything about DPDK or NetVM's PCI device based communication channels. Instead, our NetLib framework provides an interface between PCI and user applications. User applications only need to provide a structure containing

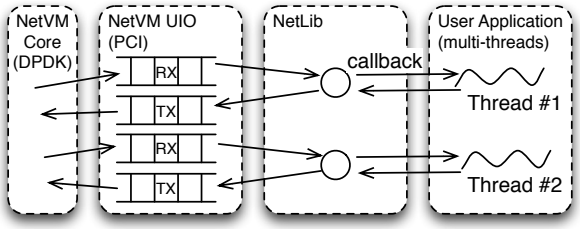


Figure 8: NetLib provides a bridge between PCI device and user applications.

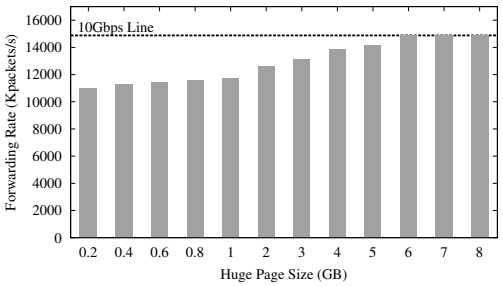


Figure 9: Huge page size can degrade throughput up to 26% (64-byte packets). NetVM needs 6GB to achieve the line rate speed.

configuration settings such as the number of cores, and a callback function. The callback function works similar to NetFilter in the linux kernel [32], a popular framework for packet filtering and manipulation. The callback function is called when a packet is received. User applications can read and write into packets, and decide what to do next. Actions include discard, send out to NIC, and forward to another VM. As explained in Section 4.1, user applications know the role numbers of other VMs. Therefore, when forwarding packets to another VM, user applications can specify the role number, not network addresses. This abstraction provides an easy way to implement communication channels between VMs.

Figure 8 illustrates a packet flow. When a packet is received from the hypervisor, a thread in NetLib fetches it and calls back a user application with the packet data. Then the user application processes the packet (read or/and write), and returns with an action. NetLib puts the action in the packet descriptor and sends it out to a transmit queue. NetLib supports multi-threading by providing each user thread with its own pair of input and output queues. There are no data exchanges between threads since NetLib provides a lockless model as NetVM does.

5 Evaluation

NetVM enables high speed packet delivery in-and-out of VMs and between VMs, and provides flexibility to steer traffic between function components that reside in distinct VMs on the NetVM platform. In this section, we evaluate NetVM with the following goals:

- Demonstrate NetVM's ability to provide high speed packet delivery with typical applications such as: Layer 3 forwarding, a userspace software router, and a firewall (§ 5.2),
- Show that the added latency with NetVM functioning as a middlebox is minimal (§ 5.3),
- Analyze the CPU time based on the task segment (§ 5.4), and
- Demonstrate NetVM's ability to steer traffic flexibly between VMs(§ 5.5).

In our experimental setup, we use two Xeon CPU X5650 @ 2.67GHz (2x6 cores) servers—one for the system under test and the other acting as a traffic generator—each of which has an Intel 82599EB 10G Dual Port NIC (with one port used for our performance experiments) and 48GB memory. We use 8GB for huge pages because Figure 9 shows that at least 6GB is needed to achieve the full line-rate (we have seen in Intel's performance reports setting 8GB as a default huge page size). The host OS is Red Hat 6.2 (kernel 2.6.32), and the guest OS is Ubuntu 12.10 (kernel 3.5). DPDK-1.4.1 and QEMU-1.5.0 are used. We use PktGen from WindRiver to generate traffic [33]. The base core assignment otherwise mentioned differently follows 2 cores to receive, 4 cores to transmit/forward, and 2 cores per VM.

We also compare NetVM with SR-IOV, the high performance IO pass-through system popularly used. SR-IOV allows the NIC to be logically partitioned into "virtual functions", each of which can be mapped to a different VM. We measure and compare the performance and flexibility provided by these architectures.

5.1 Applications

L3 Forwarder [34]: We use a simple layer-3 router. The forwarding function uses a hash map for the flow classification stage. Hashing is used in combination with a flow table to map each input packet to its flow at runtime. The hash lookup key is represented by a 5-tuple. The ID of the output interface for the input packet is read from the identified flow table entry. The set of flows used by the application is statically configured and loaded into the hash at initialization time (this simple layer-3 router is similar to the sample L3 forwarder provided in the DPDK library).

Click Userspace Router [10]: We also use Click, a more advanced userspace router toolkit to measure the performance that may be achieved by 'plugging in' an existing router implementation as-is into a VM, treating it as a 'container'. Click supports the composition of elements that each performs simple computations, but together can provide more advanced functionality such as IP routing. We have slightly modified Click by adding new receive and transmit elments that use Netlib for faster network IO. In total our changes comprise approximately 1000

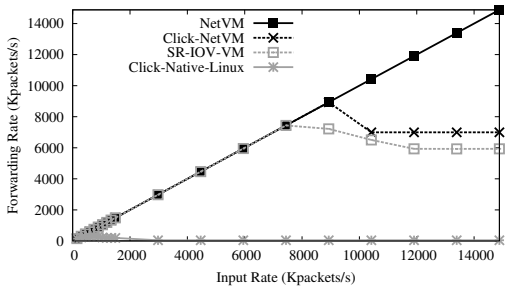


Figure 10: Forwarding rate as a function of input rate for NetVM, Click using NetVM, SR-IOV (DPDK in VM), and Native Linux Click-NetVM router (64-byte packets).

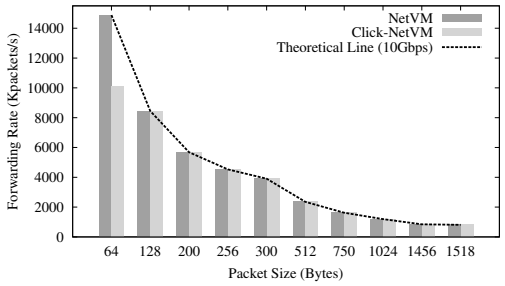


Figure 11: NetVM provides a line-rate speed regardless of packet sizes. Due to large application overhead, Click-NetVM achieves 6.8Gbps with 64-byte packet size.

lines of code. We test both a standard version of Click using Linux IO and our Netlib zero-copy version.

Firewall [35]: Firewalls control the flow of network traffic based on security policies. We use Netlib to build the foundational feature for firewalls—the packet filter. Firewalls with packet filters operate at layer 3, the network layer. This provides network access control based on several pieces of information in a packet, including the usual 5-tuple: the packet's source and destination IP address, network or transport protocol id, source and destination port; in addition its decision rules would also factor in the interface being traversed by the packet, and its direction (inbound or outbound).

5.2 High Speed Packet Delivery

Packet Forwarding Performance: NetVM's goal is to provide line rate throughput, despite running on a virtualized platform. To show that NetVM can indeed achieve this, we show the L3 packet forwarding rate vs. the input traffic rate. The theoretical value for the nominal 64-byte IP packet for a 10G Ethernet interface−with preamble size of 8 bytes, a minimum inter-frame gap 12 bytes−is 14,880,952 packets.

Figure 10 shows the input rate and the forwarded rate in packets/sec for three cases: NetVM's simple L3 forwarder, the Click router using NetVM (Click-NetVM), and Click router using native Linux (Click-Native-Linux). NetVM achieves the full line-rate, whereas Click-NetVM has a maximum rate of around 6Gbps. This is because Click has added overheads for scheduling elements (confirmed by the latency analysis we present subsequently in Table 1). Notice that increasing the input rate results in either a slight drop-off in the forwarding rate (as a result of wasted processing of packets that are ultimately dropped), or plateaus at that maximum rate. We believe Click-NetVM's performance could be further improved by either adding multi-threading support or using a faster processor, but SR-IOV can not achieve better performance this way. Not surprisingly, Click-Native-Linux performance is extremely poor (max 327Mbps), illustrating the dramatic improvement provided simply by zero-copy IO. [10].

With SR-IOV, the VM has two virtual functions associated with it and runs DPDK with two ports using two cores. SR-IOV achieves a maximum throughput of 5Gbps. We have observed that increasing the number of virtual functions or cores does not improve the maximum throughput. We speculate this limitation comes from the speed limitation on hardware switching.

Figure 11 now shows the forwarding rate as the packet size is varied. Since NetVM does not have further overheads as a consequence of the increased packet size (data is delivered by DMA), it easily achieves the full line-rate. Also, Click-NetVM also can provide the full line-rate for 128-byte and larger packet sizes.

Inter-VM Packet Delivery: NetVM's goal is to build complex network functionality by composing chains of VMs. To evaluate how pipelining VM processing elements affects throughput, we measure the achieved throughput when varying the number of VMs through which a packet must flow. We compare NetVM to a set of SR-IOV VMs, the state-of-the-art for virtualized networking.

Figure 12 shows that NetVM achieves a significantly higher base throughput for one VM, and that it is able to maintain nearly the line rate for chains of up to three VMs. After this point, our 12-core system does not have enough cores to dedicate to each VM, so there begins to be a processing bottleneck (e.g., four VMs require a total of 14 cores: 2 cores−one from each processor for NUMA-awareness−to receive packets in the host, 4 cores to transmit/forward between VMs, and 2 cores per VM for application-level processing). We believe that more powerful systems should easily be able to support longer chains using our architecture.

For a more realistic scenario, we consider a chain where 40% of incoming traffic is processed only by the first VM (an L2 switch) before being transmitted out the wire, while the remaining 60% is sent from the L2 switch VM through a Firewall VM, and then an L3 switch VM (e.g., a load balancer). In this case, our test machine has sufficient CPU capacity to achieve the line-rate for

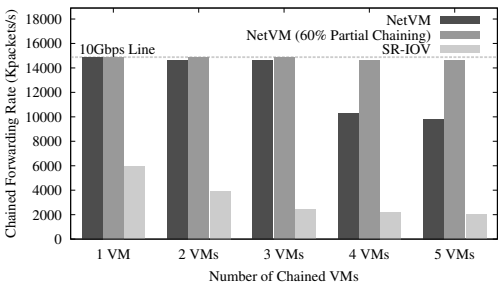


Figure 12: Inter-VM communication using NetVM can achieve a line-rate speed when VMs are well scheduled in different CPU cores (here, up to 3 VMs).

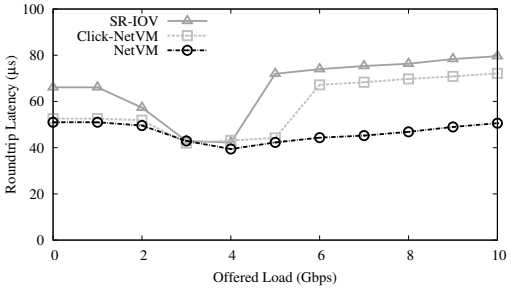


Figure 13: Average roundtrip latency for L3 forwarding.

the three VM chain, and sees only a small decrease if additional L3 switch VMs are added to the end of the chain. In contrast, SR-IOV performance is affected by the negative impact of IOTLB cache-misses, as well as a high data copy cost to move between VMs. Input/output memory management units (IOMMUs) use an IOTLB to speed up address resolution, but still each IOTLB cache-miss renders a substantial increase in DMA latency and performance degradation of DMA-intensive packet processing [36, 37].

5.3 Latency

While maintaining line-rate throughput is critical for in-network services, it is also important for the latency added by the processing elements to be minimized. We quantify this by measuring the average roundtrip latency for L3 forwarding in each platform. The measurement is performed at the traffic generator by looping back 64-byte packets sent through the platform. We include a timestamp on the packet transmitted. Figure 13 shows the roundtrip latency for the three cases: NetVM, Click-NetVM, and SR-IOV using identical L3 Forwarding function. Latency for Click-NetVM and SR-IOV increases especially at higher loads when there are additional packet processing delays under overload. We speculate that at very low input rates, none of the systems are able to make full benefit of batched DMAs and pipelining between cores, explaining the initially slightly worse performance for all approaches. After the offered load exceeds 5Gbps, SR-IOV and Click are unable to keep up, causing a significant portion of packets to be dropped. In this experiment, the queue lengths are relatively small, preventing the latency from rising significantly. The drop rate of SR-IOV rises to 60% at 10Gbps, while NetVM drops zero pockets.

5.4 CPU Time Breakdown

Table 1 breaks down the CPU cost of forwarding a packet through NetVM. Costs were converted to nanoseconds from the Xeon's cycle counters [38]. Each measurement is the average over a 10 second test. These measurements are larger than the true values because using Xeon cycle counters has significant overhead (the achieved throughput drops from 10Gbps to 8.7Gbps). Most of the tasks performed by a NetVM's CPU are included in the table.

"NIC → Hypervisor" measures the time it takes DPDK to read a packet from the NIC's receive DMA ring. Then NetVM decides which VM to send the packet to and puts a small packet descriptor in the VM's receive ring ("Hypervisor → VM"). Both of these actions are performed by a single core. "VM → APP" is the time NetVM needs to get a packet from a ring buffer and delivers it to the user application; the application then spends "APP (L3 Forwarding)" time; the forwarding application (NetVM or Click) sends the packet back to the VM ("APP → VM") and NetVM puts it into the VM's transmit ring buffer ("VM → Hypervisor"). Finally, the hypervisor spends "Hypervisor → NIC" time to send out a packet to the NIC's transmit DMA ring.

The Core# column demonstrates how packet descriptors are pipelined through different cores for different tasks. As was explained in Section 3.3, packet processing is restricted to the same socket to prevent NUMA overheads. In this case, only "APP (L3 Forwarding)" reads/writes the packet content.

5.5 Flexibility

NetVM allows for flexible switching capabilities, which can also help improve performance. Whereas Intel SR-IOV can only switch packets based on the L2 address, NetVM can steer traffic (per-packet or per-flow) to a spe-

Core#	Task	Time (ns/packet)	
		Simple	Click
0	NIC → Hypervisor	27.8	27.8
0	Hypervisor → VM	16.7	16.7
1	VM → APP	1.8	29.4
1	APP (L3 Forwarding)	37.7	41.5
1	APP → VM	1.8	129.0
1	VM → Hypervisor	1.8	1.8
2	Hypervisor → NIC	0.6	0.6
Total		88.3	246.8

Table 1: CPU Time Cost Breakdown for NetLib's Simple L3 router and Click L3 router.

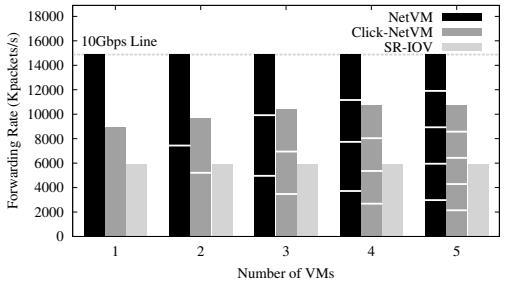


Figure 14: State-dependent (or data-dependent) load-balancing enables flexible steering of traffic. The graph shows a uniformly distributed load-balancing.

cific VM depending on system load (e.g., using the occupancy of the packet descriptor ring as an indication), shallow packet inspection (header checking), or deep packet inspection (header + payload checking) in the face of performance degradation. Figure 14 illustrates the forwarding rate when load-balancing is based on load of packets queued−the queue with the smallest number of packets has the highest priority. The stacked bars show how much traffic each VM receives and the total. NetVM is able to evenly balance load across VMs. Click-NetVM shows a significant performance improvement with multiple VMs (up to 20%) since additional cores are able to load balance the more expensive application-level processing. The SR-IOV system is simply unable to make use of multiple VMs in this way since the MAC addresses coming from the packet generator are all same. Adding more cores to the single SR-IOV VM does also not improve performance. We believe this will be a realistic scenario in the network (not just in our testbed) as the MAC addresses of incoming packets at a middlebox or a router will likely be the same across all packets.

We also have observed the same performance graph for NetVM's shallow packet inspection that load-balances based on the protocol type; deep-packet inspection overhead will depend on the amount of computation required while analyzing the packet. With many different network functions deployed, more dynamic workloads with SDN capability are left for the future works.

6 Discussion

We have shown NetVM's zero-copy packet delivery framework can effectively bring high performance for network traffic moving through a virtualized network platform. Here we discuss related issues, limitations, and future directions.

Scale to next generation machines: In this work, we have used the first CPU version (Nehalem architecture) that supports Intel's DPDK. Subsequent generations of processors from Intel, the Sandy-bridge and Ivy-bridge processors have significant additional hardware capabilities (i.e., cores), so we expect that this will allow both greater total throughput (by connecting to multiple NIC ports in parallel), and deeper VM chains. Reports in the commercial press and vendor claims indicate that there is almost a linear performance improvement with the number of cores for native Linux (i.e., non-virtualized). Since NetVM eliminates the overheads of other virtual IO techniques like SR-IOV, we also expect to see the same linear improvement by adding more cores and NICs.

Building Edge Routers with NetVM: We recognize that the capabilities of NetVM to act as a network element, such as an edge router in an ISP context, depends on having a large number of interfaces, albeit at lower speeds. While a COTS platform may have a limited number of NICs, each at 10Gbps, a judicious combination of a low cost Layer 2 (Ethernet) switch and NetVM will likely serve as an alternative to (what are generally high cost) current edge router platforms. Since the features and capabilities (in terms of policy and QoS) required on an edge router platform are often more complex, the cost of ASIC implementations tend to rise steeply. This is precisely where the additional processing power of the recent processors combined with the NetVM architecture can be an extremely attractive alternative. The use of the low cost L2 switch provides the necessary multiplexing/demultiplexing required to complement NetVM's ability to absorb complex functions, potentially with dynamic composition of those functions.

Open vSwitch and SDN integration: SDN allows greater flexibility for control plane management. However, the constraints of the hardware implementations of switches and routers often prevent SDN rules from being based on anything but simple packet header information. Open vSwitch has enabled greater network automation and reconfigurability, but its performance is limited because of the need to copy data. Our goal in NetVM is to build a base platform that can offer greater flexibility while providing high speed data movement underneath. We aim to integrate Open vSwitch capabilities into our NetVM Manager. In this way, the inputs that come from a SDN Controller using OpenFlow could be used to guide NetVM's management and switching behavior. NetVM's flexibility in demultiplexing can accommodate more complex rule sets, potentially allowing SDN control primitives to evolve.

Other Hypervisors: Our implementation uses KVM, but we believe the NetVM architecture could be applied to other virtualization platforms. For example, a similar setup could be applied to Xen; the NetVM Core would run in Domain-0, and Xen's grant table functionality would be used to directly share the memory regions used to store packet data. However, Xen's limited support for huge pages would have to be enhanced.

7 Related Work

The introduction of multi-core and multi-processor systems has led to significant advances in the capabilities of software based routers. The RouteBricks project sought to increase the speed of software routers by exploiting parallelism at both the CPU and server level [39]. Similarly, Kim et. al. [11] demonstrate how batching I/O and CPU operations can improve routing performance on multi-core systems. Rather than using regular CPU cores, PacketShader [28] utilizes the power of general purpose graphic processing units (GPGPU) to accelerate packet processing. Hyper-switch [40] on the other hand uses a low-overhead mechanism that takes into account CPU cache locality, especially in NUMA systems. All of these approaches demonstrate that the memory access time bottlenecks that prevented software routers such as Click [10] from performing line-rate processing are beginning to shift. However, none of these existing approaches support deployment of network services in virtual environments, a requirement that we believe is crucial for lower cost COTS platforms to replace purpose-built hardware and provide automated, flexible network function management.

The desire to implement network functions in software, to enable both flexibility and reduced cost because of running on COTS hardware, has recently taken concrete shape with a multitude of network operators and vendors beginning to work together in various industry forums. In particular, the work spearheaded by European Telecommunications Standards Institute (ETSI) on network function virtualization (NFV) has outlined the concept recently [41, 42]. While the benefits of NFV in reducing equipment cost and power consumption, improving flexibility, reduced time to deploy functionality and enabling multiple applications on a single platform (rather than having multiple purpose-specific network appliances in the network) are clear, there is still the outstanding problem of achieving high-performance. To achieve a fully capable NFV, high-speed packet delivery and low latency is required. NetVM provides the fundamental underlying platform to achieve this.

Improving I/O speeds in virtualized environments has long been a challenge. Santos et al. narrow the performance gap by optimzing Xen's driver domain model to reduce execution costs for gigabit Ethernet NICs [43]. vBalance dynamically and adaptively migrates the interrupts from a preempted vCPU to a running one, and hence avoids interrupt processing delays to improve the I/O performance for SMP-VMs [44]. vTurbo accelerates I/O processing for VMs by offloading that task to a designated core called a turbo core that runs with a much smaller time slice than the cores shared by production VMs [45]. VPE improves the performance of I/O device virtualization by using dedicated CPU cores [46]. However, none of these achieve full line-rate packet forwarding (and processing) for network links operating at 10Gbps or higher speeds. While we base our platform on DPDK, other approaches such as netmap [47] also provide highspeed NIC to userspace I/O.

Researchers have looked into middlebox virtualization on commodity servers. Split/Merge [48] describes a new abstraction (Split/Merge), and a system (FreeFlow), that enables transparent, balanced elasticity for stateful virtual middleboxes to have the ability to migrate flows dynamically. xOMB [6] provides flexible, programmable, and incrementally scalable middleboxes based on commodity servers and operating systems to achieve high scalability and dynamic flow management. CoMb [8] addresses key resource management and implementation challenges that arise in exploiting the benefits of consolidation in middlebox deployments. These systems provide flexible management of networks and are complementary to the the high-speed packet forwarding and processing capability of NetVM.

8 Conclusion

We have described a high-speed network packet processing platform, NetVM, built from commodity servers that use virtualization. By utilizing Intel's DPDK library, NetVM provides a flexible traffic steering capability under the hypervisor's control, overcoming the performance limitations of the existing, popular SR-IOV hardware switching techniques. NetVM provides the capability to chain network functions on the platform to provide a flexible, high-performance network element incorporating multiple functions. At the same time, NetVM allows VMs to be grouped into multiple trust domains, allowing one server to be safely multiplexed for network functionality from competing users.

We have demonstrated how we solve NetVM's design and implementation challenges. Our evaluation shows NetVM outperforms the current SR-IOV based system for forwarding functions and for functions spanning multiple VMs, both in terms of high throughput and reduced packet processing latency. NetVM provides greater flexibility in packet switching/demultiplexing, including support for state-dependent load-balancing. NetVM demonstrates that recent advances in multi-core processors and NIC hardware have shifted the bottleneck away from software-based network processing, even for virtual platforms that typically have much greater IO overheads.

Acknowledgments

We thank our shepherd, KyoungSoo Park, and reviewers for their help improving this paper. This work was supported in part by NSF grant CNS-1253575.

References

[1] Minlan Yu, Lavanya Jose, and Rui Miao. Software defined traffic measurement with opensketch. In *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation*, nsdi'13, pages 29–42, Berkeley, CA, USA, 2013. USENIX Association.

[2] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. Composing software-defined networks. In *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation*, nsdi'13, pages 1–14, Berkeley, CA, USA, 2013. USENIX Association.

[3] Ahmed Khurshid, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. Veriflow: verifying network-wide invariants in real time. In *Proceedings of the first workshop on Hot topics in software defined networks*, HotSDN '12, pages 49–54, New York, NY, USA, 2012. ACM.

[4] Ben Pfaff, Justin Pettit, Teemu Koponen, Keith Amidon, Martin Casado, and Scott Shenker. Extending networking into the virtualization layer. In *8th ACM Workshop on Hot Topics inNetworks (HotNets-VIII).New YorkCity,NY(October 2009)*.

[5] Intel Corporation. Intel data plane development kit: Getting started guide. 2013.

[6] James W. Anderson, Ryan Braud, Rishi Kapoor, George Porter, and Amin Vahdat. xomb: extensible open middleboxes with commodity servers. In *Proceedings of the eighth ACM/IEEE symposium on Architectures for networking and communications systems*, ANCS '12, pages 49–60, New York, NY, USA, 2012. ACM.

[7] Adam Greenhalgh, Felipe Huici, Mickael Hoerdt, Panagiotis Papadimitriou, Mark Handley, and Laurent Mathy. Flow processing and the rise of commodity network hardware. *SIGCOMM Comput. Commun. Rev.*, 39(2):20–26, March 2009.

[8] Vyas Sekar, Norbert Egi, Sylvia Ratnasamy, Michael K. Reiter, and Guangyu Shi. Design and implementation of a consolidated middlebox architecture. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, NSDI'12, pages 24–24, Berkeley, CA, USA, 2012. USENIX Association.

[9] Raffaele Bolla and Roberto Bruschi. Pc-based software routers: high performance and application service support. In *Proceedings of the ACM workshop on Programmable routers for extensible services of tomorrow*, PRESTO '08, pages 27–32, New York, NY, USA, 2008. ACM.

[10] Eddie Kohler. The click modular router. *PhD Thesis*, 2000.

[11] Joongi Kim, Seonggu Huh, Keon Jang, KyoungSoo Park, and Sue Moon. The power of batching in the click modular router. In *Proceedings of the Asia-Pacific Workshop on Systems*, APSYS '12, pages 14:1–14:6, New York, NY, USA, 2012. ACM.

[12] Constantinos Dovrolis, Brad Thayer, and Parameswaran Ramanathan. Hip: Hybrid interrupt-polling for the network interface. *ACM Operating Systems Reviews*, 35:50–60, 2001.

[13] Jisoo Yang, Dave B. Minturn, and Frank Hady. When poll is better than interrupt. In *Proceedings of the 10th USENIX conference on File and Storage Technologies*, FAST'12, pages 3–3, Berkeley, CA, USA, 2012. USENIX Association.

[14] Jeffrey C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems*, 15:217–252, 1997.

[15] Wenji Wu, Matt Crawford, and Mark Bowden. The performance analysis of linux networking - packet receiving. *Comput. Commun.*, 30(5):1044–1057, March 2007.

[16] Younggyun Koh, Calton Pu, Sapan Bhatia, and Charles Consel. Efficient packet processing in user-level os: A study of uml. In *in Proceedings of the 31th IEEE Conference on Local Computer Networks (LCN06)*, 2006.

[17] Intel Corporation. Intel virtualization technology for directed i/o. 2007.

[18] Intel Corp. Intel data plane development kit: Programmer's guide. 2013.

[19] Open vSwitch. http://www.openvswitch.org.

[20] VMWare White Paper. Vmware vnetwork distributed switch. *2013*.

[21] Intel Open Source Technology Center. https://01.org/packet-processing.

[22] Intel Corp. Intel data plane development kit: Getting started guide. 2013.

[23] Francis M. David, Jeffrey C. Carlyle, and Roy H. Campbell. Context switch overheads for linux on arm platforms. In *Proceedings of the 2007 workshop on Experimental computer science*, ExpCS '07, New York, NY, USA, 2007. ACM.

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

[25] Jeff Dean. Designs, Lessons and Advice from Building Large Distributed Systems. LADIS Keynote, 2009.

[26] Srihari Makineni, Ravi Iyer, Partha Sarangam, Donald Newell, Li Zhao, Ramesh Illikkal, and Jaideep Moses. Receive side coalescing for ac-

celerating tcp/ip processing. In *Proceedings of the 13th International Conference on High Performance Computing*, HiPC'06, pages 289–300, Berlin, Heidelberg, 2006. Springer-Verlag.

[27] Wind River White Paper. High-performance multicore networking software design options. 2013.

[28] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. Packetshader: a gpu-accelerated software router. In *Proceedings of the ACM SIGCOMM 2010 conference*, SIGCOMM '10, pages 195–206, New York, NY, USA, 2010. ACM.

[29] Yinan Li, Ippokratis Pandis, Rene Mueller, Vijayshankar Raman, and Guy Lohman. Numa-aware algorithms: the case of data shuffling. *The biennial Conference on Innovative Data Systems Research (CIDR)*, 2013.

[30] David Levinthal. Performance analysis guide for intel core i7 processor and intel xeon 5500. 2013.

[31] A. Cameron Macdonell. Shared-memory optimizations for virtual machines. *PhD Thesis*.

[32] Rusty Russell and Harald Welte. Linux netfilter hacking howto. *http://www.netfilter.org/documentation/HOWTO /netfilter-hacking-HOWTO.html*.

[33] Wind River Technical Report. Wind river application acceleration engine. 2013.

[34] Intel Corp. Intel data plane development kit: Sample application user guide. 2013.

[35] Karen Scarfone and Paul Hoffman. Guidelines on firewalls and firewall policy. *National Institute of Standards and Technology*, 2009.

[36] Nadav Amit, Muli Ben-Yehuda, and Ben-Ami Yassour. Iommu: strategies for mitigating the iotlb bottleneck. In *Proceedings of the 2010 international conference on Computer Architecture*, ISCA'10, pages 256–274, Berlin, Heidelberg, 2012. Springer-Verlag.

[37] Muli Ben-Yehuda, Jimi Xenidis, Michal Ostrowski, Karl Rister, Alexis Bruemmer, and Leendert Van Doorn. The price of safety: Evaluating iommu performance. In *In Proceedings of the 2007 Linux Symposium*, 2007.

[38] Intel Corporation. Intel 64 and ia-32 architectures software developer's manual. 2013.

[39] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. RouteBricks: exploiting parallelism to scale software routers. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, page 1528, New York, NY, USA, 2009. ACM.

[40] Kaushik Kumar Ram, Alan L. Cox, Mehul Chadha, and Scott Rixner. Hyper-switch: A scalable software virtual switching architecture. *USENIX Annual Technical Conference (USENIX ATC)*, 2013.

[41] SDN and OpenFlow World Congress Introductory White Paper. Network functions virtualisation. *http://portal.etsi.org/NFV/NFV_White_Paper.pdf*, 2012.

[42] Frank Yue. Network functions virtualization - everything old is new again. *http://www.f5.com/pdf/white-papers/service-provider-nfv-white-paper.pdf*, 2013.

[43] Jose Renato Santos, Yoshio Turner, G. Janakiraman, and Ian Pratt. Bridging the gap between software and hardware techniques for i/o virtualization. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, ATC'08, pages 29–42, Berkeley, CA, USA, 2008. USENIX Association.

[44] Luwei Cheng and Cho-Li Wang. vbalance: using interrupt load balance to improve i/o performance for smp virtual machines. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 2:1–2:14, New York, NY, USA, 2012. ACM.

[45] Cong Xu, Sahan Gamage, Hui Lu, Ramana Kompella, and Dongyan Xu. vturbo: Accelerating virtual machine i/o processing using designated turbo-sliced core. *USENIX Annual Technical Conference*, 2013.

[46] Jiuxing Liu and Bulent Abali. Virtualization polling engine (vpe): using dedicated cpu cores to accelerate i/o virtualization. In *Proceedings of the 23rd international conference on Supercomputing*, ICS '09, pages 225–234, New York, NY, USA, 2009. ACM.

[47] Luigi Rizzo. netmap: A novel framework for fast packet I/O. In *USENIX Annual Technical Conference*, pages 101–112, Berkeley, CA, 2012. USENIX.

[48] Shriram Rajagopalan, Dan Williams, Hani Jamjoom, and Andrew Warfield. Split/merge: system support for elastic execution in virtual middleboxes. In *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation*, nsdi'13, pages 227–240, Berkeley, CA, USA, 2013. USENIX Association.