Virtualization Polling Engine (VPE): Using Dedicated CPU Cores to Accelerate I/O Virtualization

Jiuxing Liu
IBM T. J. Watson Research Center
19 Skyline Drive
Hawthorne, NY 10605, USA
jl@us.ibm.com

Bulent Abali
IBM T. J. Watson Research Center
19 Skyline Drive
Hawthorne, NY 10605, USA
abali@us.ibm.com

ABSTRACT

Virtual machine (VM) technologies are making rapid progress and VM performance is approaching that of native hardware in many aspects. Achieving high performance for I/O virtualization remains a challenge, however, especially for high speed networking devices such as 10 Gigabit Ethernet (10 GbE) NICs. Traditional software-based approaches to I/O virtualization usually suffer significant performance degradation compared with native hardware. Hardware-based approaches that allow direct device access in VMs can achieve good performance, albeit at the expense of increased hardware cost and increased complexity in achieving tasks such as VM checkpointing, migration, and record/reply.

Recently, the trend in microprocessor design has shifted from achieving higher CPU frequencies to putting more cores in a single chip, thus the cost of each core is rapidly decreasing. In this paper, we propose a new I/O virtualization approach called the *Virtualization Polling Engine (VPE)*. VPE introduces a concept called *virtualization onload*, which takes advantage of dedicated CPU cores to help with the virtualization of I/O devices by using an event-driven execution model with dedicated polling threads. It can significantly reduce virtualization overhead and achieve performance close to the hardware-based approaches without requiring special hardware support.

Using our VPE approach, we developed a prototype called KVM-VPE to provide Ethernet virtualization support for KVM. Our experiments in a 10GbE testbed showed that VPE significantly outperformed the original KVM. In Netperf TCP tests our prototype achieved over 5 times the bandwidth for transmitting (Tx) and over 3 times the bandwidth for receiving (Rx) compared with the original KVM. KVM-VPE also supports direct user application access to the virtual Ethernet interfaces and achieved 7.4 μs end-to-end latency between two VMs on different machines in our testbed. Overall, our research demonstrated that VPE is a promising approach to high performance I/O virtualization in the coming multicore era.

ICS'09 June 8–12, 2009, Yorktown Heights, New York, USA

Categories and Subject Descriptors

C.2.2 [**Computer-Communication Networks**]: Network Protocols; D.4.m [**Operating Systems**]: Miscellaneous

General Terms

Design, measurement, performance

Keywords

Virtual machine, Ethernet, networking

1. INTRODUCTION

Virtual machine (VM) technologies are experiencing a resurgence after they were first introduced in the 1960s [13]. They are becoming attractive to both the industry and the academia, and used widely in many different areas [35] including server consolidation, desktop computing, and secure computing. However, due to the cost of virtualizing different system components such as CPUs, memory, and I/O devices, a VM typically suffers from performance degradation compared with native hardware. For the x86 architecture, recently introduced software techniques [48, 10, 1] and hardware features [47, 2] have significantly narrowed the performance gap between VMs and native hardware for CPU and memory virtualization. Improving the performance of I/O device virtualization remains a challenge, however, especially for high speed networking devices such as 10 Gigabit Ethernet (10GbE) NICs.

In this paper, we propose a new approach called the Virtualization Polling Engine (VPE) to improve the performance of I/O device virtualization by using dedicated CPU cores. Compared to the original KVM, our prototype (KVM-VPE) achieves over 5 times the bandwidth for transmitting (Tx) and over 3 times the bandwidth for receiving (Rx). KVM-VPE also supports direct user application access to the virtual Ethernet interfaces and obtained 7.4 μs end-to-end latency between two VMs on two different machines in our testbed. To motivate our proposal, we first analyze the current approaches and point out their respective advantages and disadvantages. Then, we discuss how VPE can offer a promising solution by attaining the benefits of current I/O virtualization models while addressing many of their drawbacks.

1.1 Current I/O Virtualization Approaches

Depending on how devices are accessed by VMs, we can divide I/O virtualization models into two major classes: *software-*

based approaches and *hardware-based approaches*. In software-based approaches, several software components work together to provide virtualized access points to VMs in a safe manner without special hardware support. In these approaches, devices cannot be accessed directly by a normal VM (also called a *guest VM*). Instead, each I/O operation requires the intervention of either the virtual machine monitor (VMM) or a privileged VM. Today, software-based approaches are used in many popular VM environments such as VMware [45], Xen [10] and KVM [21].

A major disadvantage of software-based approaches is that it suffers significant performance degradation compared with native hardware, especially for networking devices [27]. Although recent research has improved their performance considerably [26, 39, 28], there still exists an I/O performance gap between VMs and native hardware for several reasons. First, since software-based approaches involve multiple components such as guest VMs, privileged VMs, and the VMM, they may suffer from context switch and control transfer overheads between these components. For example, each I/O operation in KVM may result in multiple VM exits (and re-entries) which can cost thousands of CPU cycles according to previous data [1]. Secondly, since the components are often schedulable entities, they may suffer from suboptimal scheduling of system resources such as CPU cycles. For example, previous research [29] has demonstrated that scheduling in VM environments for I/O intensive workloads is very challenging and current schedulers cannot meet all the requirements of a wide range of applications. Lastly, software-based I/O virtualization is usually interrupt-driven and interrupt handling has high overhead and latency, especially in VMs. For example, in a KVM VM, an interrupt for a level-triggered virtual PCI device will result in at least two VM exits (and re-entries): the first one to inject the interrupt into the VM and the second one for the VM to acknowledge and clear the interrupt line. Therefore, applications that demand very low I/O latency, such as those in the area of high performance computing (HPC), are typically not suitable for running in VM environments.

To address the performance issue of software-based approaches, hardware-based I/O virtualization approaches were introduced which allow direct hardware access from within a guest VM [23, 32, 50]. In these approaches, performance critical I/O operations can be carried out by a guest VM by interacting with the hardware directly. To ensure VM safety and isolation, set-up and I/O management may still need to involve the VMM or a privileged VM. To achieve high performance at the application level, some hardware-based approaches [23] even support mapping I/O access points into the address spaces of user applications in a VM and let it perform I/O directly. Another hardware-based virtualization approach is to use self-virtualizing PCI Express I/O devices [30]. These devices can appear as multiple virtual PCI Express devices and each virtual device can be dedicated to a single VM.

Hardware-based virtualization, however, has its own limitations. First, it requires intelligent hardware support in the I/O devices, which increases the cost of the hardware. Second, safe direct hardware access to DMA-capable devices in VMs may require IOMMU (IOMMU stands for IO memory management unit and performs for IO devices the equivalent of page translation and protection hardware typically available for CPUs only.) support and previous studies have shown that using IOMMUs in VMs is a complicated task that can result in performance degradation [49, 4]. Finally, direct hardware access makes many common VM tasks such as checkpointing, migration and record/replay difficult to achieve.

1.2 Virtualization Polling Engine (VPE)

Recently, the trend in microprocessor design has shifted from achieving higher CPU frequencies to adding more cores in a single chip [15], thus the cost of each core is decreasing rapidly. In this paper, we propose a new I/O virtualization model which achieves performance close to hardware-based approaches while attaining the benefits of software-based approaches by making use of dedicated CPU cores. We call this approach the Virtualization Polling Engine (VPE). In VPE, device interrupts are disabled and dedicated VPE polling threads are used to access hardware devices and present their virtual access points to different VMs. Communication between the VPE threads and the VMs happens in the form of simple reads/writes through cache-coherent memory, instead of expensive operations such as VM exits and interrupts. (A VM can still choose to receive interrupts if desired.) Similar to other software-based approaches, VPE does not require special hardware support and VM checkpointing, migration, and record/replay can be implemented in straightforward ways. On the other hand, VPE has a number of advantages over traditional software-based approaches which help it achieve better performance. First, VPE suffers much less from suboptimal CPU scheduling because it uses dedicated cores. Second, I/O operations in a guest VM no longer result in context switches or control transfers (such as VM exits). Third, VPE can eliminate the use of interrupts for both real hardware devices and virtual devices in VMs to achieve very low I/O latency. Last, similar to previous work in [23], mapping of virtual I/O access points into the address spaces of user applications in VMs is possible with VPE. Demanding applications can use this method to achieve maximum performance. Our VPE approach can also be called *virtualzation onload*, as it bears some similarities to the *TCP onload* approach [34].

To demonstrate the idea of VPE, we have developed a prototype called KVM-VPE which virtualizes 10 GbE NICs for KVM. KVM-VPE uses a dedicated CPU core to provide efficient and virtualized Ethernet access to KVM VMs through the virtio [36] interface. The prototype requires only small changes to the current network device driver in guest KVM VMs. Although our implementation currently only works for Chelsio T3b 10GbE NICs [7], support for other NICs can be easily added. Compared with the original KVM (which uses a fairly optimized paravirtualized network driver), our prototype achieves over 5 times the bandwidth for transmitting (Tx) and more than 3 times the bandwidth for receiving (Rx) using the Netperf TCP benchmark [20] in our testbed. It also results in significant improvement in TCP ping-pong latency for short messages. Furthermore, KVM-VPE achieves the performance gain without requiring any special hardware support.

KVM-VPE also supports direct access to virtual Ethernet interfaces from user applications in VMs which may be useful for low latency, user mode message passing libraries such as MPI [44]. To take advantage of this feature, we have designed a lightweight protocol called LWEP which provides reliability and flow control over the standard Ethernet pro-

tocol. In our testbed, user applications running in VMs using LWEP deliver end-to-end latencies of around 7.4 μs for small messages. For large messages, they can achieve peak Tx bandwidths of over 1133 MB/s, and end-to-end bandwidths of more than 812 MB/s. Thus, our performance evaluation has shown that VPE has a clear performance advantage over traditional software based I/O virtualization approaches and also provides more flexibility in accessing I/O devices. It should also be noted that all the above performance results were achieved using the standard 1500 byte Ethernet MTU.

In summary, the main contributions of our work are:

- We propose the Virtualization Polling Engine (VPE) approach of using dedicated CPU cores to virtualize I/O devices. VPE can significantly reduce I/O virtualization overhead while retaining many advantages of software-based virtualization approaches.

- Based on the idea of VPE, we implemented a software prototype called KVM-VPE which can provide efficient virtualization of 10GbE Ethernet devices for KVM.

- We conducted detailed performance evaluation of KVM-VPE. Our results show that VPE significantly outperforms the original KVM and provides performance much closer to that of native hardware.

Next, we introduce KVM and virtio. Section 3 discusses the general architecture of VPE. In section 4, we present the detailed design and implementation of the KVM-VPE prototype. Performance results of our prototype are presented in section 5. We discuss related work in section 6 and conclude in section 7.

2. KVM AND VIRTIO

Our work is done in the context of KVM, and our prototype implementation is based on virtio, which is an infrastructure for paravirtualized drivers in VMs. In this section, we provide brief overviews of both KVM and virtio. Readers are referred to [21] and [36] for detailed descriptions.

The core part of KVM (Kernel-based Virtual Machine) is a Linux kernel module that enables the Linux OS to function as a VMM. KVM takes advantage of hardware virtualization support such as that provided by AMD and Intel [47, 2] to achieve efficient virtualization of CPUs and memory. In KVM, a VM is just a Linux process. A new execution mode called the *guest mode* is introduced to execute VM instructions natively with the help of hardware virtualization support.

Similar to other Linux processes, the virtual address space of a VM process in KVM consists of both user and kernel parts. Furthermore, all the physical pages of the VM are mapped into the user part of the address space. Therefore, the physical memory of the VM can be accessed easily when the VM process is in the user mode.

KVM handles I/O operations from VMs in a different way than other VM environments such as Xen [11, 10] and VMware [45]. For each I/O operation, a VM exits from the guest mode and enters the host kernel mode. Then, the I/O operation is redirected to the host user mode (also called *KVM userspace*) and emulated.

Virtio was proposed to unify all the paravirtualized device drivers in the Linux kernel by presenting a set of standard APIs for handling I/O requests.

In virtio, I/O operations involve two parties: a *frontend* and a *backend*. Basically, the frontend is a device driver in a VM, and virtio provides an interface for it to submit I/O requests and also query the status of the requests. Once submitted, the requests are fulfilled by the backend. In virtio, a basic I/O access point for the frontend is a virtio queue, or *virtqueue*. (A paravirtualized I/O device can have more than one virtqueues.) A virtqueue supports several operations, including those to submit I/O requests (*add_buf*) and to notify the backend (*kick*).

Each virtqueue has an important data structure called a *vring* associated with it. Essentially, a vring is a shared buffer between the frontend and the backend. It should be noted that vring only provides a mechanism for the frontend and the backend to share information about submitted and finished I/O request buffers. Communication between them also needs a signalling mechanism. The signalling mechanism is usually implementation dependent.

3. VPE BASIC ARCHITECTURE

The VPE approach assumes that virtualization of I/O devices follows a frontend/backend structure similar to those in Xen and virtio. The central part of the architecture is a set of polling threads that control a set of physical devices and service a number of virtual access points to the devices. It should be noted that the VPE polling threads and the physical devices do not need to have one-to-one correspondence. However, to simplify explanation, we assume one polling thread controls exactly one physical device in this section. Essentially, the VPE polling thread serves as a backend for virtualizing the real device. The thread runs on a dedicated CPU core and is never de-scheduled. The execution of a VPE polling thread is a tightly coupled event loop: The thread constantly polls all the virtual access points and the physical device for new events, and then process them.

VPE uses memory reads/writes to access the virtual access points it controls and to find out about new requests submitted by VMs. As a result, it requires the virtual access points to be represented by memory buffers (as in the case of vring in virtio). In VPE, frontends also use memory (write) operations to signal the polling thread for new requests. The polling thread polls the corresponding memory locations to discover new requests. A similar approach can be used to notify the frontends about completed I/O requests. However, typical frontend device drivers which work in the kernel mode cannot use polling to check the status of submitted requests. To accommodate these cases, VPE can optionally inject interrupts into the VMs where the frontends reside, depending on the configurations of the virtual access points.

The use of polling, as well as using normal memory operations for signalling between the frontends and the backends, brings several performance advantages over the traditional software-based I/O virtualization approach:

- *VPE is less prone to suboptimal scheduling.* Since I/O handling in a VM environment usually requires the cooperation of other schedulable entities (such as the dom0 in Xen or host kernel I/O threads in KVM) besides the VM itself, I/O performance is heavily depen-

dent on the scheduling decisions made by the system. Unfortunately, current schedulers mainly focus on providing fair sharing of CPU cycles instead of efficient I/O handling, which often leads to poor and unpredictable I/O performance [29]. In VPE, host (backend) I/O handling uses dedicated CPU cores. Therefore, system scheduling will much less likely to affect I/O performance adversely, because it can only impact the the VMs which initiate I/O operations.

- *VPE results in fewer context switches and control transfers.* In traditional software-based I/O virtualization, an I/O operation often results in multiple context switches and control transfers. For instance, I/O operations in KVM will result in frequent VM exits, which were shown to have significant overhead [1]. To make things worse, KVM also causes reduced parallelism in I/O handling: The I/O processing is serialized between the frontend running in the guest mode and the backend running in the host (user or kernel) mode, even though in many cases they can potentially run in parallel. In VPE, simple memory operations are used as the signalling mechanism between the frontends and the backends, which will not cause any context switches or control transfers. Furthermore, VPE allows better parallelism because the polling threads can run at the same time as the VMs.
- *VPE can achieve very low latency by eliminating the use of interrupts in the backend and the frontend.* I/O interrupts can result in increased latency and unpredictable performance. Thus, they are not desirable for applications that demand very low I/O latency or have real-time constraints. To address this issue, VPE uses polling exclusively instead of interrupts on the backend side. However, VPE provides flexibility for the use of interrupts in frontends: Each I/O access points can be easily configured to allow either memory-based or interrupt-based notification.

VPE also enables direct accesses to virtual I/O interfaces for user applications running in VMs. Later in this section, we will show how applications can use this feature to achieve high performance without sacrificing safety or isolation.

VPE does have some limitations. First, it requires dedicated CPU cores, which means increased system cost or reduced available resources. However, microprocessor design is rapidly shifting toward the "many-core" era [15, 3]. Recently, Intel has developed an 80 core processor [18], and chips with hundreds and even thousands of cores may not be far in the future [8]. As the cost of each core decreases, VPE will become more attractive.

Second, since the VPE polling thread and the VMs communicate through shared memory mechanisms, the caching structure of the system can affect I/O performance. Fortunately, many multicore processors have shared caches which can speed up the communication between the VPE thread and the VMs.

One may also question the scalability of using memory-based signalling because the polling thread polls the flags for every virtual interface and it may not scale well as the number of interfaces increases. Although we do not explore this issue further in this paper, we believe that this problem can be addressed by using a hierarchical scheme: In addition to their own notification flags, a group of frontends can also share a single flag (or counter) which they access using atomic memory operations. Therefore, the polling thread normally only needs to check the shared flag. It only checks individual flags after it has found the shared flag to be set.

Another issue with the VPE approach is that it requires a polling-based driver for the physical device, which may be incompatible with existing interrupt-driven drivers. However, in the case of network drivers, it is straightforward to convert an existing interrupt-based NIC driver to a polling-based one, as we will show in the next section. Given the performance advantage of VPE, we think that this requirement can be justified.

3.1 Accessing Virtual I/O Devices from Userspace

As mentioned earlier, VPE allows virtual I/O devices in a VM to be accessed directly by user applications. This is possible because virtual I/O devices in the VPE approach are represented by memory buffers. Thus, user applications can use Unix system call *mmap* to gain access to the buffers and initiating I/O operations without any involvement of the OS. This is similar to the approach used in [23], where InfiniBand access points (represented by memory buffers and I/O pages) are mapped into the address spaces of user applications and accessed directly. However, the VPE approach does not require any special device support to achieve this, unlike the previous work [23].

Usually, virtual I/O devices in the kernel are interrupt-driven. However, userspace virtual devices can use polling to achieve better performance. This is especially attractive to applications that monopolize CPUs and demand very low latency, such as Message Passing Interface (MPI) [44] applications in the area of high performance computing (HPC).

Direct I/O access from userspace raises the question of system safety and isolation. Specifically, we must prevent one application from tampering with another's memory. The VPE approach addresses this issue through the following: First, we require that I/O requests for userspace virtual devices use *guest virtual addresses* instead of *guest physical addresses*. Second, VPE requires every buffer used for userspace I/O to be registered with VPE backend (the polling thread) in advance. This requirement of buffer registration is similar to those found in InfiniBand [17] and iWARP [16].

4. KVM-VPE PROTOTYPE DESIGN AND IMPLEMENTATION

To demonstrate the idea of VPE, we have developed a software prototype called KVM-VPE. KVM-VPE uses the VPE approach to virtualize Ethernet interfaces such as 10GbE NICs for KVM virtual machines. KVM-VPE's virtual interfaces are based on virtio [36]. The basic architecture of the KVM-VPE prototype follows what we described in Section 3. Its structure is shown in Figure 1. Next, we describe its main components, followed by other important design and implementation issues.

4.1 Main Components of KVM-VPE

As shown in Figure 1, the central part of KVM-VPE is the *VPE polling thread* running in the host kernel. It controls the network interface through the *NIC driver polling interface* whose implementation is provided by a *polling-based NIC driver*. The VPE polling thread also controls a set of virtual Ethernet interfaces which are represented by virtio ring buffers. Associated with the VPE polling thread

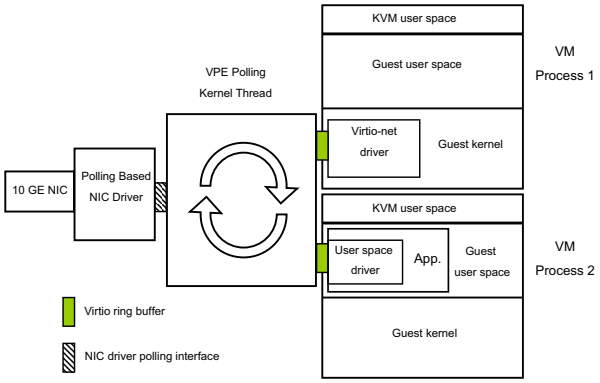

Figure 1: KVM-VPE Structure

is a Linux kernel module which also provides an interface to handle management tasks such as adding/removing virtual interfaces, activating the physical interface, and getting run-time performance data.

In KVM, each virtio device is implemented as a virtual PCI device. The emulation of the device, which includes handling of actual I/O data transfers, is done in KVM userspace. Within each VM, the *guest interface drivers* provide access to the virtual Ethernet interfaces either from kernel or user space.

4.2 Polling-Based NIC Driver

KVM-VPE uses a special, polling-based NIC driver for the physical Ethernet interface it controls. The interface between this driver and the VPE polling thread provides the following functions:

- Sending an Ethernet Frame: This function is invoked by the VPE polling thread when it wants to send an Ethernet Frame out to the external network.
- Checking for completion of send and receive operations: This function is used by the polling thread to check if an outstanding send operation has completed or if there are any incoming Ethernet frames.
- Processing the completion of send and receive events and invoking the completion handlers as discussed next.
- Registering completion handlers specific to the implementation of the VPE polling thread, which provides a more portable way of implementing the NIC driver.

We have used the Linux kernel data structure *sk_buff* to represent outgoing and incoming Ethernet frames. Existing NIC drivers in the Linux kernel are interrupt-driven and cannot be used directly for KVM-VPE. Fortunately, it is straightforward to convert an existing Linux NIC driver to one required by VPE because many Linux NIC drivers implement the NAPI interface [38]. which provides polling-based access to the NIC.

We have implemented a polling-based NIC driver for the Chelsio T3b 10GbE NICs [7]. Our modifications to the existing driver are only about 100 lines of code, many of which are used to adapt the existing code to our new interface.

4.3 Signalling Mechanisms

As we have mentioned, signalling (notification) mechanisms in VPE are considerably different from the traditional software-based I/O virtualization approaches. First, notifications from the frontends to the backends are memory-based. In our implementation, we associated two shared flags with each virtual Ethernet interfaces, one for sending and one for receiving. The flags are set by the frontends and cleared by the backends.

Notifications from the backends to the frontends can be either memory-based or interrupt-based, depending on the configuration of the virtual interfaces. When interrupt-based notifications are used, the VPE polling thread (backend) injects an virtual interrupt into the VM.

4.4 VPE Polling Thread Event Loop

The VPE polling thread works in an event loop to handle both the events from the physical interface and the requests from virtual interfaces. It also acts as a virtual Ethernet switch which connects the physical link with all the virtual interfaces. Currently, we use a simple weighted round-robin approach in servicing the interfaces. Next, we discuss how send (Tx) and receive (Rx) operations of virtual interfaces are handled by the polling thread.

4.4.1 Tx Processing

A Tx request consists of a set of buffers to be sent. (In virtio, a Tx request also has a header which provides extra control information such as that related to checksum offload and TCP Segmentation Offload, or TSO.) After being notified by a frontend about new Tx requests, the polling thread fetches the I/O buffer addresses through the shared ring buffer. For kernel virtual interfaces, the addresses are guest physical addresses which can be easily translated to virtual addresses of the VM process.

In order to get access to the content of the I/O buffers, we pin the pages corresponding to the virtual addresses and translate them into host physical addresses using the Linux kernel function *get_user_pages*. Then, the beginning of the I/O buffers is mapped into the virtual address space of the polling thread and the destination address in the Ethernet header is checked. If the destination is another virtual interface, the rest of the buffers are mapped and memory copy is used to transfer the data into the Rx buffers of the interface. (Before that, the Rx buffers also need to be pinned and mapped.) Then, we notify both the sender and the receiver about completion of their respective operations and free the allocated resources.

If the destination address does not correspond to any virtual interface, we must transmit the buffers out to the external network. First, we construct an *sk_buff* data structure from the pinned buffers. Then, we invoked the send function of the polling-based network driver. The completion handler of the send function, which notifies the VM about Tx completion and frees allocated resources, will be invoked when the buffer transmission is done. Note that sending buffers to the physical interface is essential a *zero-copy* operation and no content of the I/O buffer is touched except for the Ethernet header.

The destination address can also be a broadcast or multicast address. In this case, we send the buffers to both the physical interface and all other virtual interfaces.

4.4.2 Rx Processing

In KVM-VPE, we process Rx buffers from virtual interfaces lazily and notifications of posted Rx buffers are simply

ignored by the polling thread. Thus, most of the Rx processing happens in the receive completion handler of the polling-based network driver. When we find an incoming packet through the driver, we first check the Ethernet header to find out which virtual interface corresponds to its destination address. (If no corresponding virtual interface is found, the packet is dropped. If the address is a broadcast or multicast address, it is sent to all virtual interfaces.) Next, we fetch the first set of Rx buffers from the virtual interface, pin them in memory, and map them into the polling thread's address space. (The packet is dropped if there is no available Rx buffers.) After that, we use a *memory copy* to transfer the packet into the Rx buffers. Finally, we notify the destination virtual interface about the completion of the Rx operation and free the resources which are no longer being used.

There are a couple of things to be noted in the above process. First, unlike Tx, Rx processing in KVM-VPE is not zero-copy. While it may possible to use page swapping to achieve zero-copy Rx in VPE, previous studies have found that this approach may not achieve the best overall performance [26, 39]. Second, we have used an optimization to reduce the number of interrupt injections by coalescing the notifications and send them in batches.

4.5 Guest Kernel Driver Changes

KVM-VPE requires only a few changes to the original virtio network driver in guest kernels. The first change is in the network interface setup and shutdown processing where code for handling VPE capable interfaces is added. The second change is to replace the original notification method to the backend with the memory-based method. In addition to the above mandatory changes, we have also introduced a couple of performance enhancements for the guest kernel driver including reducing the number of interrupts in Tx processing and adding software-based Large Receive Offload (LRO). The basic idea of LRO is to aggregate multiple packets from the same TCP streams and process them in batches to reduce protocol and buffer management overhead [14, 28]. Our LRO implementation is based on the framework proposed in [46], which is now a part of the official Linux kernel.

4.6 Handling Stateless TCP Offload

KVM-VPE supports common stateless TCP offload features such as TSO and checksum offload. In virtio, each network Tx request has a header describing information related to TSO and checksum offload. When the VPE polling thread finds a request with such features, it will take special actions. If the packet is to be sent out to the external network, it will use the stateless offload supports in the physical interface to transmit the buffers. If the destination is a local virtual interface, our KVM-VPE prototype can emulate TSO in software by breaking down the Tx buffers and transmitting them to the destination interface.

4.7 Userspace Direct Access Support

In our current implementation, userspace direct access is achieved by allocating a virtual interface in the kernel and then mapping it (using mmap) to the address space of a user process. Currently, we allow polling-based access from the frontends for such interfaces.

Virtual interfaces mapped into userspace must use guest virtual addresses for I/O buffers. These addresses need to be registered before they can be used for I/O. The registration process mostly involves translating and pinning the I/O buffers and making the translation information available to the polling thread.

Our KVM-VPE prototype supplies a user-level driver to manage and access interfaces in userspace. Although it is possible to implement TCP/IP in userspace, we have implemented our own customized, lightweight protocol called LWEP (LightWeight Ethernet Protocol) over Ethernet to get a better idea of the performance potential of using user-level direct access. The main benefit of LWEP would be a high performance implementation of a user space library on top of it such as MPI. Next, we will provide more information about this protocol.

4.7.1 LightWeight Ethernet Protocol (LWEP) Overview

LWEP is a very lightweight protocol compared with TCP/IP. Unlike stream-based TCP, LWEP is message-based. Each LWEP message is encapsulated into exactly one Ethernet frame. LWEP also assumes that an interface is used exclusively by a single application, and therefore does not provide further de-multiplexing. LWEP is connection-based and maintains state information (such as per connection message sequence number) for each connection. However, unlike TCP, LWEP connections can be setup through out-of-band mechanisms. An LWEP header, which is placed immediately after the Ethernet header, contains information such as message length, message type, message sequence number, acknowledgment of last received message, etc. LWEP uses acknowledgments and retransmissions to make sure that each message will arrive at its destination. Integrity of each message is ensured by Ethernet CRC, although we provide checksum protection similar to TCP if it is desired. LWEP also implements flow control through a simple sliding-window based protocol.

4.7.2 Implementing Stateless Offload for LWEP

Many Ethernet NICs incorporate stateless offload features such as large send offload (LSO) and checksum offload for TCP and UDP. However, very few of them provide such support to other protocols. In order to implement LWEP efficiently, we have come up with a scheme to support LSO and checksum offload for LWEP by taking advantage of the TSO and TCP checksum offload features of modern NICs. The basic idea of our scheme is to re-organize the headers of LWEP to be exactly like TCP/IP headers and trick the NIC into thinking that it is dealing with normal TCP stateless offload. For example, we have used the IP total length field to represent message length and TCP checksum field to store the message checksum. Since the IPID field is incremented for each packet, we have used it for the message sequence number. Other fields in the LWEP header are the same for all the packets generated through the LSO process. Therefore, we map them to the fields in the TCP/IP header which stays the same during the TSO process. Because the LWEP header is considerable shorter than the combined TCP and IP header (which is at least 40 bytes), the mapping can be easily done. Some other fields in the TCP/IP header, including the TCP sequence number, are simply left unused.

5. EVALUATION

Our testbed consists of a cluster of IBM x3550 servers with Intel Quad-core E5345 CPUs. The machines are equipped

with Chelsio 10GbE T3b PCIe NICs and connected by a Fulcrum 10GbE switch. We used Linux 2.6.24 as the host kernel and 2.6.25-rc3 as the guest VM kernel. Our polling-based NIC driver is based on the *cxgb3* driver included in the host kernel. The version of KVM used in the experiments was 63. In all the tests, the MTU of the Ethernet network was 1500 bytes. The Intel E5345 CPUs used in our tests have 4 cores. However, the four cores are on two different chips, with the two on the same chip sharing an L2 cache. Each of our servers has 4 GB memory and we assigned 256 MB memory for each KVM VM. In many of the tests with KVM-VPE, we ran a uniprocessor VM. To achieve better performance, we binded the VM and the polling thread to cores on a single chip using the CPUSET mechanism in Linux [9] so that they shared the L2 cache.

5.1 TCP Performance

To characterize the TCP performance of our prototype, we used NetPIPE [43] to measure TCP ping-pong latency and Netperf [20] to measure TCP bandwidth. In all the tests, the server running the experiments communicate with another server running a native Linux OS. In Figure 2, we show the NetPIPE results for KVM, KVM-VPE, and the native case. As we can see, the original KVM has very high TCP latencies (over 160 μs) even for small messages. For KVM-VPE, the latencies have been reduced to around 29 μs. The smallest latency for the native case is around 14 μs.

Figure 3 shows the Netperf TCP bandwidth results. (We used the default message size (16 KBytes) in all the tests. For Tx tests, we used the *TCP_SENDFILE* option to reduce overhead.) For the native case, Netperf can achieve a peak bandwidth of over 6600 Mbps. The bottleneck was the Rx side CPU which had 100% utilization. The Tx sides were still not saturated, suggesting that they could achieve even higher performance. In comparison, the original KVM only achieved 1229 Mbps for Tx and 1421 Mbps for Rx. (Note that we incorporated our LRO support into the original KVM driver. Without such support, KVM achieved around 1200 Mbps RX bandwidth.) For both Tx and Rx, the CPU running the VM were 100% utilized and became the bottleneck. KVM-VPE was able to improve bandwidth significantly compared with KVM. Its peak Tx performance was 6677 Mbps, which was limited by the receiver side. For Rx, its bandwidth had a peak of 4319 Mbps, with the CPU running the VM becoming 100% saturated at the receiver side.

In order to find out if the VPE polling thread was the performance bottleneck in the tests, we need to get an idea of how heavily it was utilized. Since the thread uses polling, we cannot get a CPU utilization number in the traditional sense. The thread basically operates in two different modes: the *free mode* in which it polls constantly but finds nothing to do and the *busy mode* in which it actively processes events from the NIC or requests from the virtual interfaces. Therefore, we define the *VPE utilization* as the percentage of time the polling thread spends in the busy mode. We have added profiling support for VPE utilization to our code by using the Linux high resolution timer [12] to sample the current mode of the polling thread. For the previous Netperf experiments, we found that the VPE utilization was 26.6% for Tx tests and 71.8% for Rx tests, suggesting that the VPE was not the performance bottleneck in either case.

We also measured the number of VM exits for KVM-VPE and the original KVM in the Netperf tests. To ensure fair comparisons, we converted the results to the number of VM exits per Mbits transferred. Figure 4 shows the results. We can see that KVM-VPE can significantly reduce the number of VM exits for Tx. For Rx, KVM-VPE results in only a small improvement. One reason for this is that the original KVM network driver supports NAPI [38], which reduces the number of VM interrupts (and VM exits) during high load.

KVM-VPE also improves the TCP performance of inter-VM communication. In Figure 5, we see that KVM-VPE had a latency of around 51 μs for small messages, comparing to KVM's 330 μs. For bandwidth, KVM-VPE achieved 3530 Mbps while the original KVM only had 1107 Mbps.

5.2 LWEP Performance

In this subsection, we measured performance results for two VMs communicating with each other using the LWEP protocol from userspace. In Figure 6, we can see that LWEP achieved an end-to-end latency of just 7.4 μs for VMs on two different machines. In the same figure, we can also see that the inter-VM latency of LWEP was just a little more than 2 μs.

Figure 7 shows that bandwidth results. As we see, LWEP can get an end-to-end bandwidth of over 811 MB/s. (Note that MB/s is commonly used in HPC. 1 MB/s = 8 Mbps) This result was limited by the receiver side. For inter-VM communication, the peak bandwidth was 1173 MB/s. In order to get an idea of peak LWEP Tx performance, we modified our code to remove the bottleneck of copying message payloads at the receiver side. (Copying of the protocol headers was still necessary to ensure correct execution of the programs.) By using this method, we found that LWEP achieved a peak Tx bandwidth of 1133 MB/s.

The VPE utilization numbers for the LWEP end-to-end bandwidth and inter-VM tests are shown in Figure 8. We see that the Rx side VPE was the performance bottleneck in the tests because it was almost fully utilized. In contrast, The VPE at the Tx side was only about 4% utilized. This was mostly due to the zero-copy and LSO implementation for Tx which reduced overhead dramatically.

5.3 Impact of LSO

Figure 9 shows the bandwidth results with and without LSO support. (Note that we have converted Netperf TCP results to MB/s.) We can see that disabling LSO resulted in a moderate performance decrease for LWEP: a 4% drop for end-to-end performance and a 10% drop for Tx performance. However, for Netperf TCP, it was much more significant: Bandwidth dropped from over 834 MB/s to 401 MB/s. The effect of LSO on VPE utilization can be seen from Figure 10. We can see that without LSO, the VPE utilization in the Netperf tests increased to over 80%. The increase in VPE utilization for LWEP tests were even more significant without LSO: In the end-to-end tests, the utilization increased from 4% to 49%, while in the Tx tests, it increased from 6% to over 68%.

5.4 Impact of CPU Binding

Binding the VPE polling thread and the VM to CPUs on the same chip can speed up communication between them because of the shared L2 cache. Its effect can be seen from Figure 11 and Figure 12, which show the Netperf TCP bandwidth and LWEP bandwidth, respectively. From the figures,

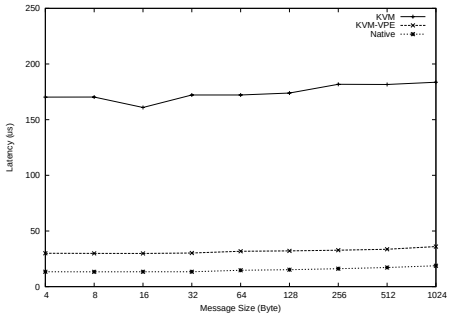

Figure 2: NetPipe TCP Latency

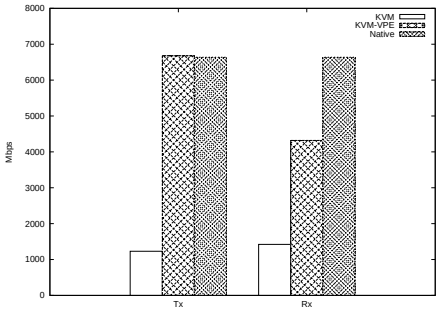

Figure 3: Netperf TCP Bandwidth

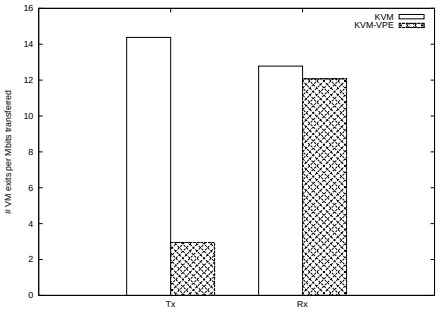

Figure 4: Netperf Test VM Exits

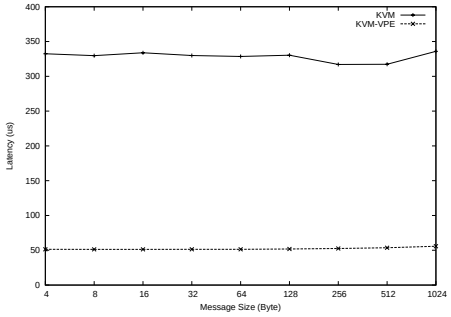

Figure 5: NetPipe Inter-VM Latency

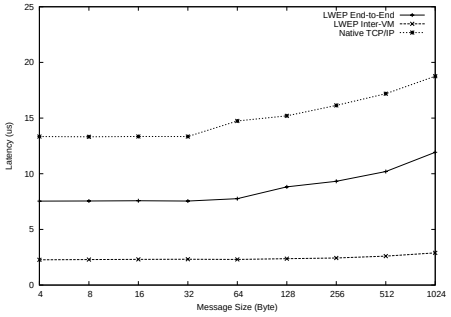

Figure 6: LWEP Latency

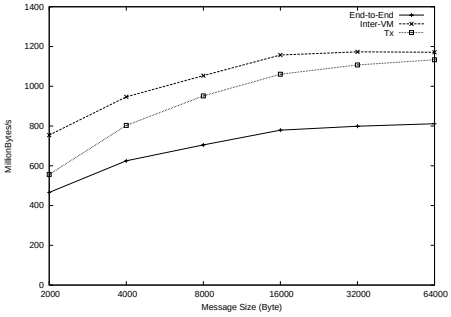

Figure 7: LWEP Bandwidth

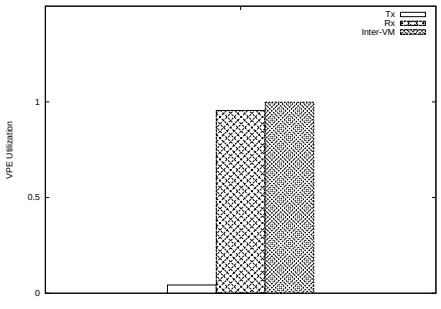

Figure 8: LWEP VPE Utilization

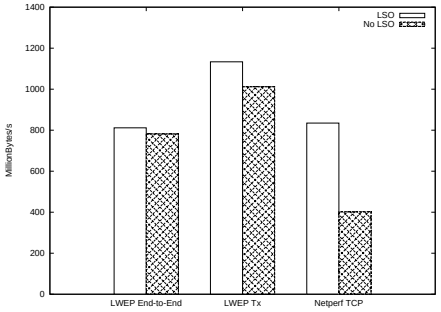

Figure 9: Impact of LSO on Bandwidth

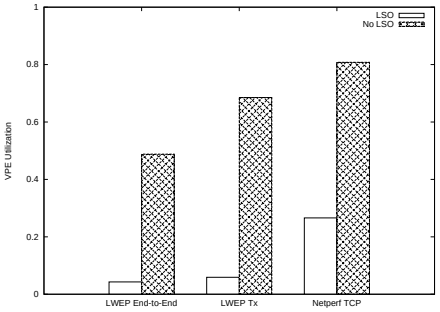

Figure 10: Impact of LSO on VPE Utilization

we notice that CPU binding increased Netperf Tx bandwidth from 6090 Mbps to 6677 Mbps, and Rx bandwidth from 3271 Mbps to 4319 Mbps. For LWEP, CPU binding increased end-to-end performance from 651 MB/s to 811 MB/s, and Tx performance from 1117 MB/s to 1133 MB/s.

Overall, CPU binding has a bigger impact on Rx performance due to the fact that our VPE prototype uses an extra memory copy to transfer data on the Rx side. However, we should note that even without CPU binding, KVM-VPE still significantly outperformed the original KVM.

6. RELATED WORK

Improving the performance of networking I/O in VMs has become an important research topic recently. Software-based techniques to improve the paravirtualized network driver in Xen have been presented in [26], [28] and [39]. Vringfd [37] was proposed to improve the efficiency of virtio vring based communication between Linux kernel and userspace. It can potentially improve the performance of KVM virtio network driver. Researchers are also exploring hardware-based I/O virtualization approaches and how to provide efficient software support for them [50, 32, 23, 39].

The idea of using dedicated CPU cores to help with I/O processing is not new. Previous research has focused on using them for protocol processing such as TCP/IP, an approach called *TCP onload* [34]. Similar approaches have been used in other studies such as [33, 41, 6]. In [25], the authors proposed to extend this approach by encapsulating I/O processing into a VM. However, all the previous studies focused on the processing of I/O protocols instead of providing virtualization of I/O devices. We believe that these approaches (*I/O protocol onloading*) can be combined with our approach (*I/O virtualization onloading*) to further reduce the I/O processing overhead of VMs.

Recently, there have been industry and academic efforts to integrate the network interface controllers with the host CPUs and use some of the CPUs for network processing [40, 5] These approaches exploit hardware support, such as giving host CPU access to FIFO buffers on the NICs, to improve networking performance. Although our VPE approach does not require special hardware support, we believe that it can benefit from the hardware features provided in the studies.

Our LWEP protocol essentially provides a thin layer for user-level direct access to virtual Ethernet NICs. Previous, lightweight user-level access to Ethernet interfaces was used in studies such as [31, 42, 40]. And in other works, user-level TCP implementations [31, 24] were used instead of a lightweight protocol. In [19], Jacobson and Felderman proposed a concept called *network channels* to speed up Linux networks. Network channels bear some similarity to virtio vrings and they can be mapped to userspace for fast access. However, network channels do not directly address the performance of network I/O virtualization.

In [22], the authors proposed the *sidecore* approach to reduce overhead of virtualization. Their work is similar to ours in spirit. However, they concentrated on memory virtualization and interrupt virtualization for a self-virtualizing I/O device while we focused general I/O virtualization.

7. CONCLUSIONS

In this paper, we proposed a new I/O virtualization approach called the Virtualization Polling Engine (VPE). VPE takes advantage of dedicated cores to help with the virtualization of I/O devices by using an event-driven model with dedicated polling threads. Unlike hardware-based I/O virtualization approaches, VPE is based on software and does not require special hardware support. Compared with other software-based I/O virtualization approaches. VPE can significantly reduce virtualization overhead and achieve performance close to the hardware-based approaches.

To demonstrate the idea of VPE, we developed a prototype called KVM-VPE to provide Ethernet virtualization support for KVM. Our experiments in a 10GbE network show that the VPE approach significantly outperformed the original KVM and achieved performance much closer to native hardware. In addition, VPE also provides more flexibility in accessing the network devices: User applications running in VMs can have direct access to virtual devices without involving the OSes and achieve very low latencies without causing VM exits. Overall, our research demonstrated that VPE is a promising approach to I/O virtualization in the coming multicore era.

8. REFERENCES

[1] K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 2–13, 2006.

[2] AMD. AMD Secure Virtual Machine Architecture Reference Manual, Rev. 3.01, May 2005.

[3] K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, et al. The Landscape of Parallel Computing Research: A View from Berkeley. *Electrical Engineering and Computer Sciences, University of California at Berkeley, Technical Report No. UCB/EECS-2006-183, December*, 18(2006-183):19, 2006.

[4] M. Ben-Yehuda, J. Xenidis, M. Mostrows, K. Rister, A. Bruemmer, and L. Van Doorn. The price of safety: Evaluating IOMMU performance. *OLS 2007: Proceedings of the 2007 Ottawa Linux Symposium*.

[5] N. Binkert, A. Saidi, and S. Reinhardt. Integrated network interfaces for high-bandwidth TCP/IP. *ACM SIGPLAN Notices*, 41(11):315–324, 2006.

[6] T. Brecht, G. Janakiraman, B. Lynn, V. Saletore, and Y. Turner. Evaluating network processing efficiency with processor partitioning and asynchronous I/O. In *Proceedings of the 2006 EuroSys conference*, volume 40, pages 265–278, 2006.

[7] Chelsio Communications. Chelsio 10GbE NICs. http://www.chelsio.com.

[8] CNet. Intel says to prepare for 'thousands of cores'. http://news.cnet.com/8301-13924%5f3-9981760-64.html.

[9] S. Derr and S. Jeaugey. CPUSETS for Linux. http://www.bullopensource.org/cpuset/.

[10] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. Xen and the Art of Virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 164–177, October 2003.

[11] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. W. and M. Williamson. Safe hardware access with the xen virtual machine monitor. In *Proceedings of OASIS ASPLOS Workshop*, 2004.

[12] T. Gleixner and D. Niehaus. Hrtimers and Beyond: Transforming the Linux Time Subsystems. *Proceedings of the Ottawa Linux Symposium, Ottawa, Ontario, Canada, July*, 2006.

[13] R. P. Goldberg. Survey of Virtual Machine Research. *IEEE Computer*, pages 34–45, June 1974.

[14] L. Grossman. Large Receive Offload Implementation in Neterion 10 GbE Ethernet Driver. *Ottawa Linux Symposium*, 2005.

[15] J. Held, J. Bautista, and S. Koehl. From a Few Cores to Many: A Tera-scale Computing Research Overview. *Intel Research (White Paper)*, 2006.

[16] J. Hilland, P. Culley, J. Pinkerton, and R. Recio. RDMA Protocol Verbs Specification. *RDMA Consortium Specification*, 2003.

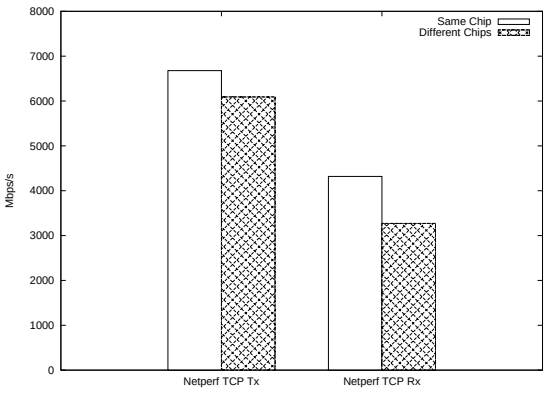

Figure 11: Impact of CPU Binding (Netperf TCP)

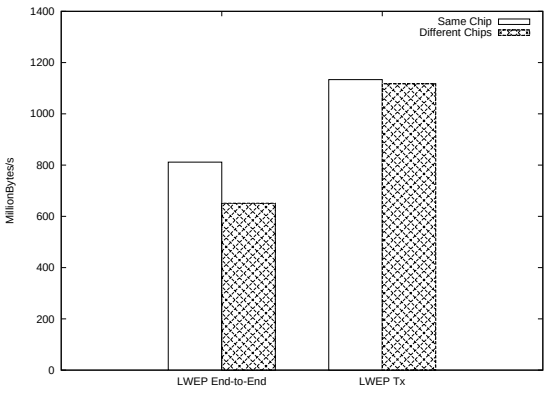

Figure 12: Impact of CPU Binding (LWEP)

[17] InfiniBand Trade Association. InfiniBand Architecture Specification, Release 1.2. http://www.infinibandta.org/members/specs/.

[18] Intel. Teraflops Research Chip. http://techresearch.intel.com/articles/Tera-Scale/1449.htm.

[19] V. Jacobson and B. Felderman. Network Channels (Speedup Networking). http://www.linux.org.au/conf/2006/abstract8204.html.

[20] R. Jones. Netperf. *HP Information Networks Division, http://www.netperf.org*.

[21] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the Linux Virtual Machine Monitor. *Linux Symposium*, 2006.

[22] S. Kumar, H. Raj, K. Schwan, and I. Ganev. Re-architecting VMMs for Multicore Systems: The Sidecore Approach. *WIOSCA 2007*.

[23] J. Liu, W. Huang, B. Abali, and D. Panda. High Performance VMM-Bypass I/O in Virtual Machines. *Proc. of USENIX Annual Technical Conference*, 2006.

[24] K. Mansley. Engineering a user-level TCP for the CLAN network. *Proceedings of the ACM SIGCOMM workshop on Network-I/O convergence: experience, lessons, implications*, pages 228–236, 2003.

[25] D. McAuley and R. Neugebauer. A case for virtual channel processors. *Proceedings of the ACM SIGCOMM workshop on Network-I/O convergence: experience, lessons, implications*, pages 237–242, 2003.

[26] A. Menon, A. Cox, and W. Zwaenepoel. Optimizing Network Virtualization in Xen. *Proc. USENIX Annual Technical Conference (USENIX 2006)*, pages 15–28, 2006.

[27] A. Menon, J. R. Santos, Y. Turner, G. Janakiraman, and W. Zwaenepoel. Diagnosing Performance Overheads in the Xen Virtual Machine Environment. In *Proceedings of the First ACM/Usenix Conference on Virtual Execution Environments (VEE'05)*, June 2005.

[28] A. Menon and W. Zwaenepoel. Optimizing TCP Receive Performance. *Proc. USENIX Annual Technical Conference (USENIX 2008)*, 2008.

[29] D. Ongaro, A. L. Cox, and S. Rixner. Scheduling I/O in Virtual Machine Monitors. *Proc. ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 2008)*, 2008.

[30] PCI-SIG. PCI I/O Virtualization. http://www.pcisig.com-/news_room/news/press_releases/2005_06_06.

[31] I. Pratt and K. Fraser. Arsenic: A user-accessible gigabit ethernet interface. In *INFOCOM*, pages 67–76, 2001.

[32] H. Raj and K. Schwan. High performance and scalable I/O virtualization via self-virtualized devices. *Proceedings of the 16th international symposium on High performance distributed computing*, pages 179–188, 2007.

[33] M. Rangarajan, A. Bohra, K. Banerjee, E. Carrera, R. Bianchini, and L. Iftode. TCP Servers: Offloading TCP Processing in Internet Servers. Design, Implementation and Performance. *Computer Science Department, Rutgers University, Mar*, 2002.

[34] G. Regnier, S. Makineni, I. Illikkal, R. Iyer, D. Minturn, R. Huggahalli, D. Newell, L. Cline, and A. Foong. TCP onloading for data center servers. *IEEE Computer*, 37(11):48–58, 2004.

[35] M. Rosenblum and T. Garfinkel. Virtual Machine Monitors: Current Technology and Future Trends. *IEEE Computer*, 38(5):39–47, 2005.

[36] R. Russell. virtio infrastructure. http://lwn.net/Articles/241104/.

[37] R. Russell. vringfd. http://lwn.net/Articles/276729/.

[38] J. Salim, R. Olsson, and A. Kuznetsov. Beyond softnet. *Proceedings of the 5th conference on 5th Annual Linux Showcase & Conference*, 2001.

[39] J. R. Santos, Y. Turner, G. Janakiraman, and I. Pratt. Bridging the Gap between Software and Hardware Techniques for I/O Virtualization. *Proc. USENIX Annual Technical Conference (USENIX 2008)*, 2008.

[40] M. Schlansker, N. Chitlur, E. Oertli, P. Stillwell Jr, L. Rankin, D. Bradford, R. Carter, J. Mudigonda, N. Binkert, and N. Jouppi. High-performance ethernet-based communications for future multi-core processors. *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, 2007.

[41] L. Shalev, V. Makhervaks, Z. Machulsky, G. Biran, J. Satran, M. Ben-Yehuda, and I. Shimony. Loosely Coupled TCP Acceleration Architecture. *High-Performance Interconnects, 14th IEEE Symposium on*, pages 3–8, 2006.

[42] P. Shivam, P. Wyckoff, and D. Panda. EMP: zero-copy OS-bypass NIC-driven gigabit ethernet message passing. *Conference on High Performance Networking and Computing: Proceedings of the 2001 ACM/IEEE conference on Supercomputing(CDROM)*, 10(16):57–57, 2001.

[43] Q. Snell, A. Mikler, and J. Gustafson. NetPIPE: A Network Protocol Independent Performance Evaluator. *IASTED International Conference on Intelligent Information Management and Systems*, 6, 1996.

[44] M. Snir, S. Otto, D. Walker, J. Dongarra, and S. Huss-Lederman. *MPI: The Complete Reference*. MIT Press Cambridge, MA, USA, 1995.

[45] J. Sugerman, G. Venkitachalam, and B. H. Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *Proceedings of USENIX*, 2001.

[46] J.-B. Theman. lro: Generic Large Receive Offload for TCP traffic. http://lwn.net/Articles/243950/.

[47] R. Uhlig, G. Neiger, D. Rodgers, A. Santoni, F. Martins, A. Anderson, S. Bennett, A. Kagi, F. Leung, and L. Smith. Intel Virtualization Technology. *IEEE Computer*, May 2005.

[48] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and Performance in the Denali Isolation Kernel. In *Proceedings of 5th USENIX OSDI, Boston, MA*, Dec 2002.

[49] P. Willmann, S. Rixner, and A. L. Cox. Protection Strategies for Direct Access to Virtualized I/O Devices. *Proc. USENIX Annual Technical Conference (USENIX 2008)*, 2008.

[50] P. Willmann, J. Shafer, D. Carr, A. Menon, S. Rixner, A. Cox, and W. Zwaenepoel. Concurrent direct network access for virtual machine monitors. *Proceedings of the International Symposium on High-Performance Computer Architecture, Feb*, 2007.