High Performance VMM-Bypass I/O in Virtual Machines

Jiuxing Liu[†] Wei Huang[‡] Bulent Abali[†] Dhabaleswar K. Panda[‡]

[†] *IBM T. J. Watson Research Center*
19 Skyline Drive
Hawthorne, NY 10532
{jl, abali}@us.ibm.com

[‡] *Computer Science and Engineering*
The Ohio State University
Columbus, OH 43210
{huanwei, panda}@cse.ohio-state.edu

Abstract

Currently, I/O device virtualization models in virtual machine (VM) environments require involvement of a virtual machine monitor (VMM) and/or a privileged VM for each I/O operation, which may turn out to be a performance bottleneck for systems with high I/O demands, especially those equipped with modern high speed interconnects such as InfiniBand.

In this paper, we propose a new device virtualization model called *VMM-bypass* I/O, which extends the idea of OS-bypass originated from user-level communication. Essentially, VMM-bypass allows time-critical I/O operations to be carried out directly in guest VMs without involvement of the VMM and/or a privileged VM. By exploiting the intelligence found in modern high speed network interfaces, VMM-bypass can significantly improve I/O and communication performance for VMs without sacrificing safety or isolation.

To demonstrate the idea of VMM-bypass, we have developed a prototype called Xen-IB, which offers InfiniBand virtualization support in the Xen 3.0 VM environment. Xen-IB runs with current InfiniBand hardware and does not require modifications to existing user-level applications or kernel-level drivers that use InfiniBand. Our performance measurements show that Xen-IB is able to achieve nearly the same raw performance as the original InfiniBand driver running in a non-virtualized environment.

1 Introduction

Virtual machine (VM) technologies were first introduced in the 1960s [14], but are experiencing a resurgence in recent years and becoming more and more attractive to both the industry and the research communities [35]. A key component in a VM environment is the virtual machine monitor (VMM) (also called hypervisor), which is implemented directly on top of hardware and provides virtualized hardware interfaces to VMs. With the help of VMMs, VM technologies allow running many different virtual machines in a single physical box, with each virtual machine possibly hosting a different operating system. VMs can also provide secure and portable environments to meet the demanding resource requirements of modern computing systems [9].

In VM environments, device I/O access in guest operating systems can be handled in different ways. For instance, in VMware Workstation, device I/O relies on switching back to the host operating system and user-level emulation [37]. In VMware ESX Server, guest VM I/O operations trap into the VMM, which makes direct access to I/O devices [42]. In Xen [11], device I/O follows a split-driver model. Only an isolated device domain (IDD) has access to the hardware using native device drivers. All other virtual machines (guest VMs, or domains) need to pass the I/O requests to the IDD to access the devices. This control transfer between domains needs involvement of the VMM.

In recent years, network interconnects that provide very low latency (less than 5μs) and very high bandwidth (multiple Gbps) are emerging. Examples of these high speed interconnects include Virtual Interface Architecture (VIA) [12], InfiniBand [19], Quadrics [34], and Myrinet [25]. Due to their excellent performance, these interconnects have become strong players in areas such as high performance computing (HPC). To achieve high performance, these interconnects usually have intelligent network interface cards (NICs) which can be used to offload a large part of the host communication protocol processing. The intelligence in the NICs also supports user-level communication, which enables safe direct I/O access from user-level processes (OS-bypass I/O) and contributes to reduced latency and CPU overhead.

VM technologies can greatly benefit computing systems built from the aforementioned high speed interconnects by not only simplifying cluster management for these systems, but also offering much cleaner solutions to tasks such as check-pointing and fail-over. Recently, as these high speed interconnects become more and more commoditized with their cost going down, they are also used for providing remote I/O access in high-end enterprise systems, which increasingly run in virtualized environments. Therefore, it is very important to provide VM support to high-end systems equipped with these high speed interconnects. However, performance and scalability requirements of these systems pose some challenges. In all the VM I/O access approaches mentioned previously, VMMs have to be involved to make sure that I/O accesses are safe and do not compromise integrity of the system. Therefore, current device I/O access in virtual machines requires context switches between the VMM and guest VMs. Thus, I/O access can suffer from longer latency and higher CPU overhead compared to native I/O access in non-virtualized environments. In some cases, the VMM may also become a performance bottleneck which limits I/O performance in guest VMs. In some of the aforementioned approaches (VM Workstation and Xen), a host operating system or another virtual machine is also involved in the I/O access path. Although these approaches can greatly simplify VMM design by moving device drivers out of the VMM, they may lead to even higher I/O access overhead when requiring context switches between the host operating system and the guest VM or two different VMs.

In this paper, we present a *VMM-bypass* approach for I/O access in VM environments. Our approach takes advantages of features found in modern high speed intelligent network interfaces to allow time-critical operations to be carried out directly in guest VMs while still maintaining system integrity and isolation. With this method, we can remove the bottleneck of going through the VMM or a separate VM for many I/O operations and significantly improve communication and I/O performance. The key idea of our VMM-bypass approach is based on the OS-bypass design of modern high speed network interfaces, which allows user processes to access I/O devices directly in a safe way without going through operating systems. OS-bypass was originally proposed by research communities [41, 40, 29, 6, 33] and later adopted by some commercial interconnects such as InfiniBand. Our idea can be regarded as an extension of OS-bypass designs in the context of VM environments.

To demonstrate the idea of VMM-bypass, we have designed and implemented a prototype called *Xen-IB* to provide virtualization support for InfiniBand in Xen. Basically, our implementation presents to each guest VM a para-virtualized InfiniBand device. Our design requires no modification to existing hardware. Also, through a technique called *high-level virtualization*, we allow current user-level applications and kernel-level modules that utilize InfiniBand to run without changes. Our performance results, which includes benchmarks at the basic InfiniBand level as well as evaluation of upper-layer InfiniBand protocols such as IP over InfiniBand (IPoIB) [1] and MPI [36], demonstrate that performance of our VMM-bypass approach comes close to that in a native, non-virtualized environment. Although our current implementation is for InfiniBand and Xen, the basic VMM-bypass idea and many of our implementation techniques can be readily applied to other high-speed interconnects and other VMMs.

In summary, the main contributions of our work are:

- We proposed the VMM-bypass approach for I/O accesses in VM environments for modern high speed interconnects. Using this approach, many I/O operations can be performed directly without involvement of a VMM or another VM. Thus, I/O performance can be greatly improved.

- Based on the idea of VMM-bypass, we implemented a prototype, Xen-IB, to virtualize InfiniBand devices in Xen guest VMs. Our prototype supports running existing InfiniBand applications and kernel modules in guest VMs without any modification.

- We carried out extensive performance evaluation of our prototype. Our results show that performance of our virtualized InfiniBand device is very close to native InfiniBand devices running in a non-virtualized environment.

The rest of the paper is organized as follows: In Section 2, we present background information, including the Xen VM environment and the InfiniBand architecture. In Section 3, we present the basic idea of VMM-bypass I/O. In Section 4, we discuss the detailed design and implementation of our Xen-IB prototype. In Section 5, we discuss several related issues and limitations of our current implementation and how they can be addressed in future. Performance evaluation results are given in Section 6. We discuss related work in Section 7 and conclude the paper in Section 8.

2 Background

In this section, we provide background information for our work. In Section 2.1, we describe how I/O device access is handled in several popular VM environments. In Section 2.3, we describe the OS-bypass feature in modern high speed network interfaces. Since our prototype is based on Xen and InfiniBand, we introduce them in Sections 2.2 and 2.4, respectively.

2.1 I/O Device Access in Virtual Machines

In a VM environment, the VMM plays the central role of virtualizing hardware resources such as CPUs, memory, and I/O devices. To maximize performance, the VMM can let guest VMs access these resources directly whenever possible. Taking CPU virtualization as an example, a guest VM can execute all non-privileged instructions natively in hardware without intervention of the VMM. However, privileged instructions executed in guest VMs will generate a trap into the VMM. The VMM will then take necessary steps to make sure that the execution can continue without compromising system integrity. Since many CPU intensive workloads seldom use privileged instructions (This is especially true for applications in HPC area.), they can achieve excellent performance even when executed in a VM.

I/O device access in VMs, however, is a completely different story. Since I/O devices are usually shared among all VMs in a physical machine, the VMM has to make sure that accesses to them are legal and consistent. Currently, this requires VMM intervention on every I/O access from guest VMs. For example, in VMware ESX Server [42], all physical I/O accesses are carried out within the VMM, which includes device drivers for popular server hardware. System integrity is achieved with every I/O access going through the VMM. Furthermore, the VMM can serve as an arbitrator/multiplexer/demultiplexer to implement useful features such as QoS control among VMs. However, VMM intervention also leads to longer I/O latency and higher CPU overhead due to the context switches between guest VMs and the VMM. Since the VMM serves as a central control point for all I/O accesses, it may also become a performance bottleneck for I/O intensive workloads.

Having device I/O access in the VMM also complicates the design of the VMM itself. It significantly limits the range of supported physical devices because new device drivers have to be developed to work within the VMM. To address this problem, VMware workstation [37] and Xen [13] carry out I/O operations in a host operating system or a special privileged VM called isolated device domain (IDD), which can run popular operating systems such as Windows and Linux that have a large number of existing device drivers. Although this approach can greatly simplify the VMM design and increase the range of supported hardware, it does not directly address performance issues with the approach used in VMware ESX Server. In fact, I/O accesses now may result in expensive operations called a world switch (a switch between the host OS and a guest VM) or a domain switch (a switch between two different VMs), which can lead to even worse I/O performance.

2.2 Overview of the Xen Virtual Machine Monitor

Xen is a popular high performance VMM. It uses para-virtualization [43], in which host operating systems need to be explicitly ported to the Xen architecture. This architecture is similar to native hardware such as the x86 architecture, with only slight modifications to support efficient virtualization. Since Xen does not require changes to the application binary interface (ABI), existing user applications can run without any modification.

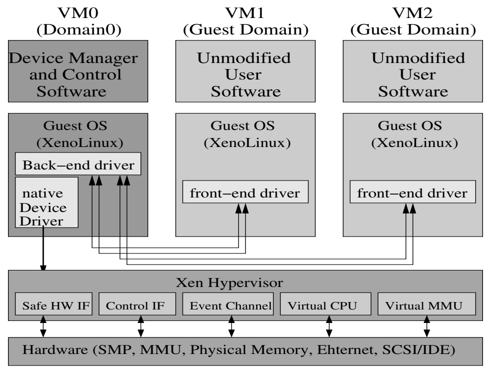

Figure 1: The structure of the Xen hypervisor, hosting three xenoLinux operating systems (courtesy [32])

Figure 1 illustrates the structure of a physical machine running Xen. The Xen hypervisor is at the lowest level and has direct access to the hardware. The hypervisor, instead of the guest operating systems, is running in the most privileged processor-level. Xen provides basic control interfaces needed to perform complex policy decisions. Above the hypervisor are the Xen domains (VMs). There can be many domains running simultaneously. Guest VMs are prevented from directly executing privileged processor instructions. A special domain called *domain0*, which is created at boot time, is allowed to access the control interface provided by the hypervisor. The guest OS in domain0 hosts application-level management software and perform the tasks to create, terminate or migrate other domains through the control interface.

There is no guarantee that a domain will get a continuous stretch of physical memory to run a guest OS. Xen makes a distinction between *machine memory* and *pseudo-physical memory*. Machine memory refers to the physical memory installed in a machine, while pseudo-physical memory is a per-domain abstraction, allowing a guest OS to treat its memory as a contiguous range of physical pages. Xen maintains the mapping between the machine and the pseudo-physical memory. Only a certain parts of the operating system needs to understand the difference between these two abstractions. Guest OSes allocate and manage their own hardware page tables, with minimal involvement of the Xen hypervisor to ensure safety and isolation.

In Xen, domains can communicate with each other through shared pages and *event channels*. Event channels provide an asynchronous notification mechanism between domains. Each domain has a set of end-points (or ports) which may be bounded to an event source. When a pair of end-points in two domains are bound together, a “send” operation on one side will cause an event to be received by the destination domain, which may in turn cause an interrupt. Event channels are only intended for sending notifications between domains. So if a domain wants to send data to another, the typical scheme is for a source domain to grant access to local memory pages to the destination domain. Then, these shared pages are used to transfer data.

Virtual machines in Xen usually do not have direct access to hardware. Since most existing device drivers assume they have complete control of the device, there cannot be multiple instantiations of such drivers in different domains for a single device. To ensure manageability and safe access, device virtualization in Xen follows a split device driver model [13]. Each device driver is expected to run in an *isolated device domain (IDD)*, which hosts a *backend* driver to serve access requests from guest domains. Each guest OS uses a *frontend* driver to communicate with the backend. The split driver organization provides security: misbehaving code in a guest domain will not result in failure of other guest domains. The split device driver model requires the development of frontend and backend drivers for each device class. A number of popular device classes such as virtual disk and virtual network are currently supported in guest domains.

2.3 OS-bypass I/O

Traditionally, device I/O accesses are carried out inside the OS kernel on behalf of application processes. However, this approach imposes several problems such as overhead caused by context switches between user processes and OS kernels and extra data copies which degrade I/O performance [5]. It can also result in *QoS crosstalk* [17] due to lacking of proper accounting for costs of I/O accesses carried out by the kernel on behalf of applications.

To address these problems, a concept called user-level communication was introduced by the research community. One of the notable features of user-level communication is *OS-bypass*, with which I/O (communication) operations can be achieved directly by user processes without involvement of OS kernels. OS-bypass was later adopted by commercial products, many of which have become popular in areas such as high performance computing where low latency is vital to applications. It should be noted that OS-bypass does not mean all I/O operations bypass the OS kernel. Usually, devices allow OS-bypass for frequent and time-critical operations while other operations, such as setup and management operations, can go through OS kernels and are handled by a privileged module, as illustrated in Figure 2.

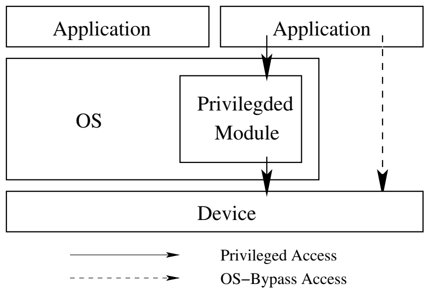

Figure 2: OS-Bypass Communication and I/O

The key challenge to implement OS-bypass I/O is to enable safe access to a device shared by many different applications. To achieve this, OS-bypass capable devices usually require more intelligence in the hardware than traditional I/O devices. Typically, an OS-bypass capable device is able to present virtual access points to different user applications. Hardware data structures for virtual access points can be encapsulated into different I/O pages. With the help of an OS kernel, the I/O pages can be mapped into the virtual address spaces of different user processes. Thus, different processes can access their own virtual access points safely, thanks to the protection provided by the virtual memory mechanism. Although the idea of user-level communication and OS-bypass was developed for traditional, non-virtualized systems, the intelligence and self-virtualizing characteristic of OS-bypass devices lend themselves nicely to a virtualized environment, as we will see later.

2.4 InfiniBand Architecture

InfiniBand [19] is a high speed interconnect offering high performance as well as features such as OS-bypass. InfiniBand host channel adapters (HCAs) are the equivalent of network interface cards (NICs) in traditional networks. InfiniBand uses a queue-based model for communication. A *Queue Pair (QP)* consists of a send queue and a receive queue. The send queue holds instructions to transmit data and the receive queue holds instructions that describe where received data is to be placed. Communication instructions are described in *Work Queue Requests (WQR)*, or descriptors, and submitted to the queue pairs. The completion of the communication is reported through *Completion Queues (CQs)* using *Completion Queue Entries (CQEs)*. CQEs can be accessed by using polling or event handlers.

Initiating data transfers (posting descriptors) and notification of their completion (polling for completion) are time-critical tasks which use OS-bypass. In the Mellanox [21] approach, which represents a typical implementation of the InfiniBand specification, posting descriptors is done by ringing a doorbell. Doorbells are rung by writing to the registers that form the *User Ac-*

cess Region (UAR). Each UAR is a 4k I/O page mapped into a process's virtual address space. Posting a work request includes putting the descriptors to a QP buffer and writing the doorbell to the UAR, which is completed without the involvement of the operating system. CQ buffers, where the CQEs are located, can also be directly accessed from the process virtual address space. These OS-bypass features make it possible for InfiniBand to provide very low communication latency.

InfiniBand also provides a comprehensive management scheme. Management communication is achieved by sending management datagrams (MADs) to well-known QPs (QP0 and QP1).

InfiniBand requires all buffers involved in communication be registered before they can be used in data transfers. In Mellanox HCAs, the purpose of registration is two-fold. First, an HCA needs to keep an entry in the Translation and Protection Table (TPT) so that it can perform virtual-to-physical translation and protection checks during data transfer. Second, the memory buffer needs to be pinned in memory so that HCA can DMA directly into the target buffer. Upon the success of registration, a local key and a remote key are returned, which can be used later for local and remote (RDMA) accesses. QP and CQ buffers described above are just normal buffers that are directly allocated from the process virtual memory space and registered with HCA.

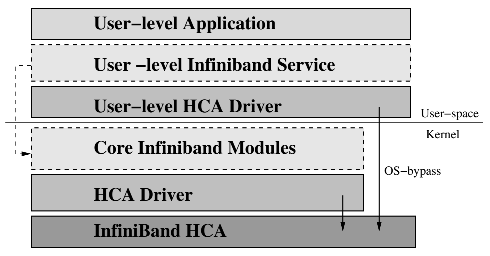

Figure 3: Architectural overview of OpenIB Gen2 stack

There are two popular stacks for InfiniBand drivers. VAPI [23] is the Mellanox implementation and OpenIB Gen2 [28] recently have come out as a new generation of IB stack provided by the OpenIB community. In this paper, our prototype implement is based on OpenIB Gen2, whose architecture is illustrated in Figure 3.

3 VMM-Bypass I/O

VMM-bypass I/O can be viewed as an extension to the idea of OS-bypass I/O in the context of VM environments. In this section, we describe the basic design of VMM-bypass I/O. Two key ideas in our design are *para-virtualization* and *high-level virtualization*.

In some VM environments, I/O devices are virtualized at the hardware level [37]. Each I/O instruction to access a device is virtualized by the VMM. With this approach, existing device drivers can be used in the guest VMs without any modification. However, it significantly increases the complexity of virtualizing devices. For example, one popular InfiniBand card (MT23108 from Mellanox [24]) presents itself as a PCI-X device to the system. After initialization, it can be accessed by the OS using memory mapped I/O. Virtualizing this device at the hardware level would require us to not only understand all the hardware commands issued through memory mapped I/O, but also implement a virtual PCI-X bus in the guest VM. Another problem with this approach is performance. Since existing physical devices are typically not designed to run in a virtualized environment, the interfaces presented at the hardware level may exhibit significant performance degradation when they are virtualized.

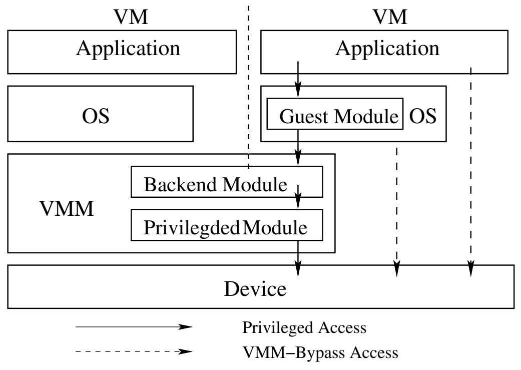

Figure 4: VM-Bypass I/O (I/O Handled by VMM Directly)

Our VMM-bypass I/O virtualization design is based on the idea of para-virtualization, similar to [11] and [44]. We do not preserve hardware interfaces of existing devices. To virtualize a device in a guest VM, we implement a device driver called *guest module* in the OS of the guest VM. The guest module is responsible for handling all the privileged accesses to the device. In order to achieve VMM-bypass device access, the guest module also needs to set things up properly so that I/O operations can be carried out directly in the guest VM. This means that the guest module must be able to create virtual access points on behalf of the guest OS and map them into the addresses of user processes. Since the guest module does not have direct access to the device hardware, we need to introduce another software component called *backend module*, which provides device hardware access for different guest modules. If devices are accessed inside the VMM, the backend module can be implemented as part of the VMM. It is possible to let the backend module talk to the device directly. However, we can greatly simplify its design by reusing the original privilege module of the OS-bypass device driver. In addition to serving as a proxy for device hardware access, the backend module also coordinates accesses among different VMs so that system integrity can be maintained. The VMM-bypass I/O design is illustrated in Figure 4.

If device accesses are provided by another VM (device driver VM), the backend module can be implemented within the device driver VM. The communication between guest modules and the backend module can be achieved through the inter-VM communication mechanism provided by the VM environment. This approach is shown in Figure 5.

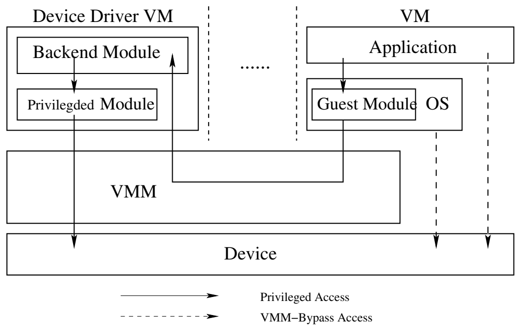

Figure 5: VM-Bypass I/O (I/O Handled by Another VM)

Para-virtualization can lead to compatibility problems because a para-virtualized device does not conform to any existing hardware interfaces. However, in our design, these problems can be addressed by maintaining existing interfaces which are at a higher level than the hardware interface (a technique we dubbed *high-level virtualization*). Modern interconnects such as InfiniBand have their own standardized access interfaces. For example, InfiniBand specification defines a *VERBS* interface for a host to talk to an InfiniBand device. The VERBS interface is usually implemented in the form of an API set through a combination of software and hardware. Our high-level virtualization approach maintains the same VERBS interface within a guest VM. Therefore, existing kernel drivers and applications that use InfiniBand will be able to run without any modification. Although in theory a driver or an application can bypass the VERBS interface and talk to InfiniBand devices directly, this seldom happens because it leads to poor portability due to the fact that different InfiniBand devices may have different hardware interfaces.

4 Prototype Design and Implementation

In this section, we present the design and implementation of Xen-IB, our InfiniBand virtualization driver for Xen. We describe details of the design and how we enable accessing the HCA from guest domains directly for time-critical tasks.

4.1 Overview

Like many other device drivers, InfiniBand drivers cannot have multiple instantiations for a single HCA. Thus, a split driver model approach is required to share a single HCA among multiple Xen domains.

Figure 6 illustrates a basic design of our Xen-IB driver. The backend runs as a kernel daemon on top of the native InfiniBand driver in the isolated device domain (IDD), which is domain0 is our current implementation. It waits for incoming requests from the frontend drivers in the guest domains. The frontend driver, which corresponds to the guest module mentioned in Section 3, replaces the kernel HCA driver in OpenIB Gen2 stack. Once the frontend is loaded, it establishes two event channels with the backend daemon. The first channel, together with shared memory pages, forms a device channel [13] which is used to process requests initiated from the guest domain. The second channel is used for sending InfiniBand CQ and QP events to the guest domain and will be discussed in detail later.

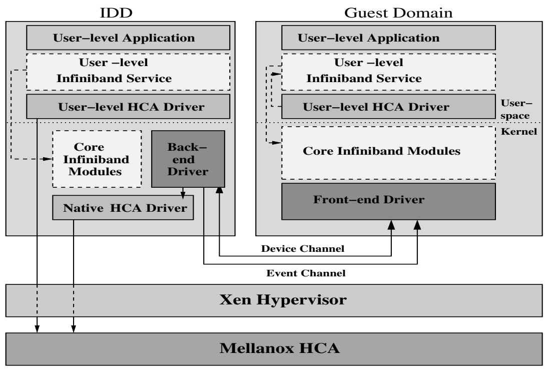

Figure 6: The Xen-IB driver structure with the split driver model

The Xen-IB frontend driver provides the same set of interfaces as a normal Gen2 stack for kernel modules. It is a relatively thin layer whose tasks include packing a request together with necessary parameters and sending it to the backend through the device channel. The backend driver reconstructs the commands, performs the operation using the native kernel HCA driver on behalf of the guest domain, and returns the result to the frontend driver.

The split device driver model in Xen poses difficulties for user-level direct HCA access in Xen guest domains. To enable VMM-bypass, we need to let guest domains have direct access to certain HCA resources such as the UARs and the QP/CQ buffers.

4.2 InfiniBand Privileged Accesses

In the following, we discuss in general how we support all privileged InfiniBand operations, including initialization, InfiniBand resource management, memory registration and event handling.

Initialization and resource management: Before applications can communicate using InfiniBand, it must finish several preparation steps including opening HCA, creating CQ, creating QP, and modifying QP status, etc. Those operations are usually not in the time critical path and can be implemented in a straightforward way. Basically, the guest domains forward these commands to the device driver domain (IDD) and wait for the acknowledgments after the operations are completed. All the resources are managed in the backend and the frontends refer to these resources by handles. Validation checks must be conducted in IDD to ensure that all references are legal.

Memory Registration: The InfiniBand specification requires all the memory regions involved in data transfers to be registered with the HCA. With Xen's paravirtualization approach, real machine addresses are directly visible to user domains. (Note that access control is still achieved because Xen makes sure a user domain cannot arbitrarily map a machine page.) Thus, a domain can easily figure out the DMA addresses of buffers and there is no extra need for address translation (assuming that no IOMMU is used). The information needed by memory registration is a list of DMA addresses that describes the physical locations of the buffers, access flags and the virtual address that the application will use when accessing the buffers. Again, the registration happens in the device domain. The frontend driver sends above information to the backend driver and get back the local and remote keys. Note that since the Translation and Protection Table (TPT) on HCA is indexed by keys, multiple guest domains are allowed to register with the same virtual address.

For security reasons, the backend driver can verify if the frontend driver offers valid DMA addresses belonging to the specific domain in which it is running. This check makes sure that all later communication activities of guest domains are within the valid address spaces.

Event Handling: InfiniBand supports several kinds of CQ and QP events. The most commonly used is the completion event. Event handlers are associated with CQs or QPs when they are created. An application can subscribe for event notification by writing a command to the UAR page. When those subscribed events happen, the HCA driver will first be notified by the HCA and then dispatch the event to different CQs or QPs according to the event type. Then the application/driver that owns the CQ/QP will get a callback on the event handler.

For Xen-IB, events are generated for the device domain, where all QPs and CQs are actually created. But the device domain cannot directly give a callback on the event handlers in the guest domains. To address this issue, we create a dedicated event channel between a frontend and the backend driver. The backend driver associates a special event handler to each CQ/QP created due to requests from guest domains. Each time the HCA generates an event to these CQs/QPs, this special event handler gets executed and forwards information such as the event type and the CQ/QP identifier to the guest domain through the event channel. The frontend driver binds an event dispatcher as a callback handler to one end of the event channel after the channel is created. The event handlers given by the applications are associated to the CQs or QPs after they are successfully created. Frontend driver also maintains a translation table between the CQ/QP identifiers and the actual CQ/QPs. Once the event dispatcher gets an event notification from the backend driver, it checks the identifier and gives the corresponding CQ/QP a callback on the associated handler.

4.3 VMM-Bypass Accesses

In InfiniBand, QP accesses (posting descriptors) include writing WQEs to the QP buffers and ringing doorbells (writing to UAR pages) to notify the HCA. Then the HCA can use DMA to transfer the WQEs to internal HCA memory and perform the send/receive or RDMA operations. Once a work request is completed, HCA will put a completion entry (CQE) in the CQ buffer. In InfiniBand, QP access functions are used for initiating communication. To detect completion of communication, CQ polling can be used. QP access and CQ polling functions are typically used in the critical path of communication. Therefore, it is very important to optimize their performance by using VMM-bypass. The basic architecture of the VMM-bypass design is shown in Figure 7.

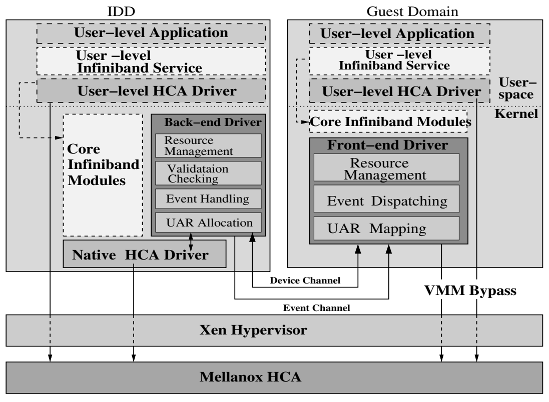

Figure 7: VMM-Bypass design of Xen-IB driver

Supporting VMM-bypass for QP access and CQ polling imposes two requirements on our design of Xen-IB: first, UAR pages must be accessible from a guest domain; second, both QP and CQ buffers should be directly visible in the guest domain.

When a frontend driver is loaded, the backend driver allocates a UAR page and returns its page frame number (machine address) to the frontend. The frontend driver then remaps this page to its own address space so that it can directly access the UAR in the guest domain to serve requests from the kernel drivers. (We have applied a small patch to Xen to enable access to I/O pages in guest domains.) In the same way, when a user application starts, the frontend driver applies for a UAR page from the backend and remaps the page to the application's virtual memory address space, which can be later accessed directly from the user space. Since all UARs are managed in a centralized manner in the IDD, there will be no conflicts between UARs in different guest domains.

To make QP and CQ buffers accessible to guest domains, creating CQs/QPs has to go through two stages. In the first stage, QP or CQ buffers are allocated in the guest domains and registered through the IDD. During the second stage, the frontend sends the CQ/QP creation commands to the IDD along with the keys returned from the registration stage to complete the creation process. Address translations are indexed by keys, so in later operations the HCA can directly read WQRs from and write the CQEs back to the buffers (using DMA) located in the guest domains.

Since we also allocate UARs to user space applications in guest domains, the user level InfiniBand library now keeps its OS-bypass feature. The VMM-bypass IB-Xen workflow is illustrated in Figure 8.

It should be noted that since VMM-bypass accesses directly interact with the HCA, they are usually hardware dependent and the frontends need to know how to deal with different types of InfiniBand HCAs. However, existing InfiniBand drivers and user-level libraries already include code for direct access and it can be reused without spending new development efforts.

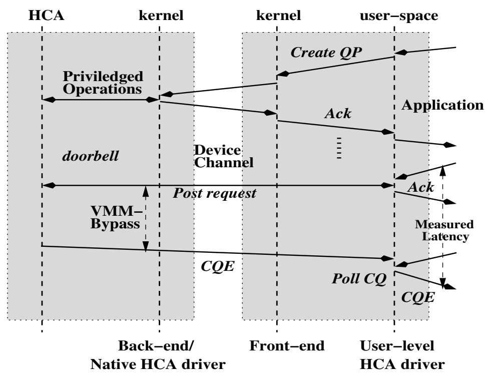

Figure 8: Working flow of the VMM-bypass Xen-IB driver

4.4 Virtualizing InfiniBand Management Operations

In an InfiniBand network, management and administrative tasks are achieved through the use of Management Datagrams (MADs). MADs are sent and received just like normal InfiniBand communication, except that they must use two well-known queue-pairs: QP0 and QP1. Since there is only one set of such queue pairs in every HCA, their access must be virtualized for accessing from many different VMs, which means we must treat them differently than normal queue-pairs. However, since queue-pair accesses can be done directly in guest VMs in our VMM-bypass approach, it would be very difficult to track each queue-pair access and take different actions based on whether it is a management queue-pair or a normal one.

To address this difficulty, we use the idea of high-level virtualization. This is based on the fact that although MAD is the basic mechanism for InfiniBand management, applications and kernel drivers seldom use it directly. Instead, different management tasks are achieved through more user-friendly and standard API sets which are implemented on top of MADs. For example, the kernel IPoIB protocol makes use of the subnet administration (SA) services, which are offered through a high-level, standardized SA API. Therefore, instead of tracking each queue-pair access, we virtualize management functions at the API level by providing our own implementation for guest VMs. Most functions can be implemented in a similar manner as privileged InfiniBand operations, which typically includes sending a request to the backend driver, executing the request (backend), and getting a reply. Since management functions are rarely in time-critical paths, the implementation will not bring any significant performance degradation. However, it does require us to implement every function provided by all the different management interfaces. Fortunately, there are only a couple of such interfaces and the implementation effort is not significant.

5 Discussions

In this section, we discuss issues related to our prototype implementation such as how safe device access is ensured, how performance isolation between different VMs can be achieved, and challenges in implementing VM check-pointing and migration with VMM-bypass. We also point out several limitations of our current prototype and how we can address them in future.

5.1 Safe Device Access

To ensure that accesses to virtual InfiniBand devices by different VMs will not compromise system integrity, we need to make sure that both privileged accesses and VMM-bypass accesses are safe. Since all privileged accesses need to go through the backend module, access checks are implemented there to guarantee safety. VMM-bypass operations are achieved through accessing the memory-mapped UAR pages which contain virtual access points. Setting-up these mappings is privileged and can be checked. InfiniBand allows using both virtual and physical addresses for sending and receiving messages or carrying out RDMA operations, as long as a valid memory key is presented. Since the key is obtained through InfiniBand memory registration, which is also a privileged operation, we implement necessary safety checks in the backend module to ensure that a VM can only carry out valid memory registration operations. It should be noted that once a memory buffer is registered, its physical memory pages cannot be reclaimed by the VMM. Therefore, we should limit the total size of buffers that can be registered by a single VM. This limit check can also be implemented in the backend module.

Memory registration is an expensive operation in InfiniBand. In our virtual InfiniBand implementation, memory registration cost is even higher due to inter-domain communication. This may lead to performance degradation in cases where buffers cannot be registered in advance. Techniques such as *pin-down cache* can be applied when buffers are reused frequently, but it is not always effective. To address this issue, some existing InfiniBand kernel drivers creates and uses an *DMA key* through which all physical pages can be accessed. Currently, our prototype supports DMA keys. However, this leaves a security hole because all physical memory pages (including those belonging to other VMs) can be accessed. In future, we plan to address this problem by letting the DMA keys only authorize access to physical pages in the current VM. However, this also means that we need to update the keys whenever the VMM changes the physical pages allocated to a VM.

5.2 Performance Isolation

Although our current prototype does not yet implement performance isolation or QoS among different VMs, this issue can be addressed by taking advantage of QoS mechanisms which are present in the current hardware. For example, Mellanox InfiniBand HCAs support a QoS scheme in which a weighted round-robin algorithm is used to schedule different queue-pairs. In this scheme, QoS policy parameters are assigned when queue-pairs are created and initialized. After that, the HCA hardware is responsible for taking necessary steps to ensure QoS policies. Since queue-pair creations are privileged, we can create desired QoS policies in the backend when queue-pairs are created. These QoS policies will later be enforced by device hardware. We plan to explore more along this direction in future.

5.3 VM Check-pointing and Migration

VMM-bypass I/O poses new challenges for implementing VM check-pointing and migration. This is due to two reasons. First, the VMM does not have complete knowledge of VMs with respect to device accesses. This is in contrast to traditional device virtualization approaches in which the VMM is involved in every I/O operation and it can easily suspend and buffer these operations when check-pointing or migration starts. The second problem is that VMM-bypass I/O exploits intelligent devices which can store a large part of the VM system states. For example, an InfiniBand HCA has onboard memory which stores information such as registered buffers, queue-pair data structures, and so on. Some of the state information on an HCA can only be changed as side effects of VERBS functions calls. It does not allow changing it in an arbitrary way. This makes it difficult for check-pointing and migrations because when a VM is restored from a previous checkpoint or migrated to another node, the corresponding state information on the HCA needs to be restored also.

There are two directions to address the above problems. The first one is to involve VMs in the process of check-pointing and migration. For example, the VMs can bring themselves to some determined states which simplify check-pointing and migration. Another way is to introduce some hardware/firmware changes. We are currently working on both directions.

6 Performance Evaluation

In this section, we first evaluate the performance of our Xen-IB prototype using a set of InfiniBand layer micro-benchmarks. Then, we present performance results for the IPoIB protocol based on Xen-IB. We also provide performance numbers of MPI on Xen-IB at both micro-benchmark and application levels.

6.1 Experimental Setup

Our experimental testbed is an InfiniBand cluster. Each system in the cluster is equipped with dual Intel Xeon 3.0GHz CPUs, 2 GB memory and a Mellanox MT23108 PCI-X InfiniBand HCA. The PCI-X buses on the systems are 64 bit and run at 133 MHz. The systems are connected with an InfiniScale InfiniBand switch. The operating systems are RedHat AS4 with 2.6.12 kernel. Xen 3.0 is used for all our experiments, with each guest domain ran with single virtual CPU and 512 MB memory.

6.2 InfiniBand Latency and Bandwidth

In this subsection, we compared user-level latency and bandwidth performance between Xen-IB and native InfiniBand. Xen-IB results were obtained from two guest domains on two different physical machines. Polling was used for detecting completion of communication.

The latency tests were carried out in a ping-pong fashion. They were repeated many times and the average half round-trip time was reported as one-way latency. Figures 9 and 10 show the latency for InfiniBand RDMA write and send/receive operations, respectively. There is very little performance difference between Xen-IB and native InfiniBand. This is because in the tests, InfiniBand communication was carried out by directly accessing the HCA from the guest domains with VMM-bypass. The lowest latency achieved by both was around 4.2 μs for RDMA write and 6.6 μs for send/receive.

In the bandwidth tests, a sender sent a number of messages to a receiver and then waited for an acknowledgment. The bandwidth was obtained by dividing the number of bytes transferred from the sender by the elapsed time of the test. From Figures 11 and 12, we again see virtually no difference between Xen-IB and native InfiniBand. Both of them were able to achieve bandwidth up to 880 MByte/s, which was limited by the bandwidth of the PCI-X bus.

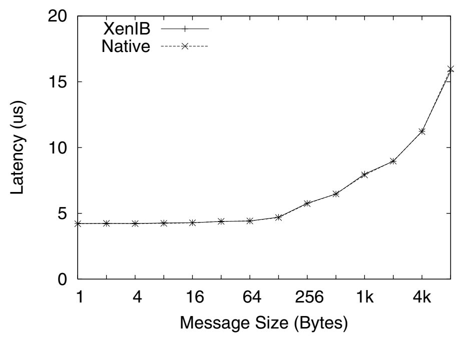

Figure 9: InfiniBand RDMA Write Latency

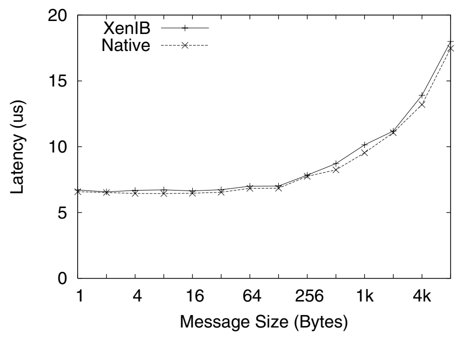

Figure 10: InfiniBand Send/Receive Latency

6.3 Event/Interrupt Handling Overhead

The latency numbers we showed in the previous subsection were based on polling schemes. In this section, we characterize the overhead of event/interrupt handling

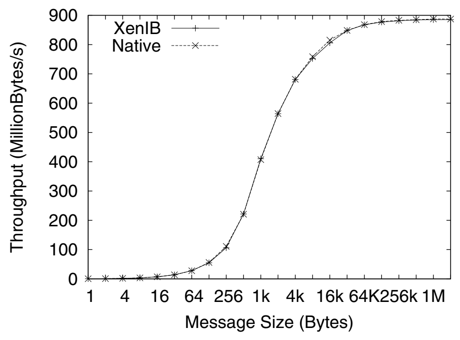

Figure 11: InfiniBand RDMA Write Bandwidth

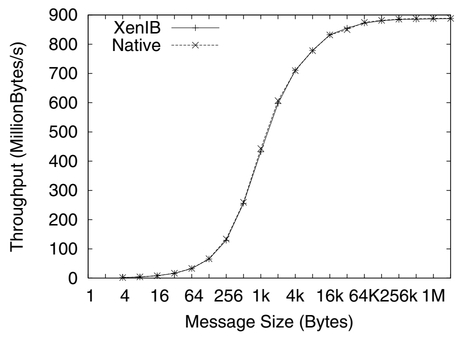

Figure 12: InfiniBand Send/Receive Bandwidth

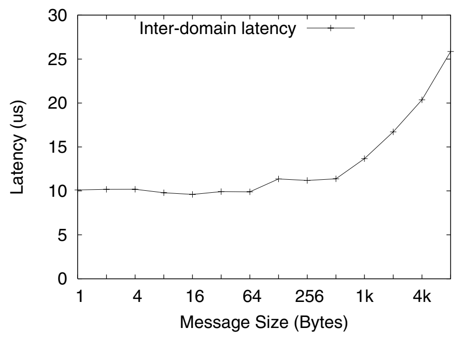

Figure 13: Inter-domain Communication One Way Latency

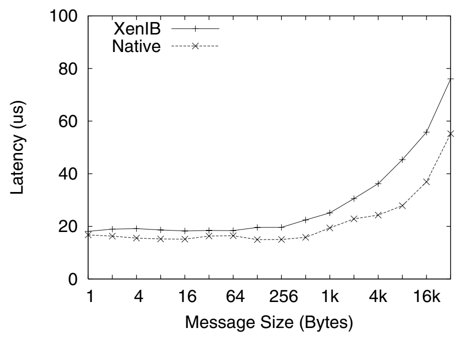

Figure 14: Send/Receive Latency Using Blocking VERBS Functions

in Xen-IB by showing send/receive latency results with blocking InfiniBand user-level VERBS functions.

Compared with native InfiniBand event/interrupt processing, Xen-IB introduces extra overhead because it requires forwarding an event from domain0 to a guest domain, which involves Xen inter-domain communication. In Figure 13, we show performance of Xen inter-domain communication. We can see that the overhead increases with the amount of data transferred. However, even with very small messages, there is an overhead of about 10 μs.

Figure 14 shows the send/receive one-way latency using blocking VERBS. The test is almost the same as the send/receive latency test using polling. The difference is that a process will block and wait for a completion event instead of busy polling on the completion queue. From the figure, we see that Xen-IB has higher latency due to overhead caused by inter-domain communication. For each message, Xen-IB needs to use inter-domain communication twice, one for send completion and one for receive completion. For large messages, we observe that the difference between Xen-IB and native InfiniBand is around 18–20 μs, which is roughly twice the inter-domain communication latency. However, for small messages, the difference is much less. For example, native InfiniBand latency is only 3 μs better for 1 byte messages. This difference gradually increases with message sizes until it reaches around 20 μs. Our profiling reveals that this is due to "event batching". For small messages, the inter-domain latency is much higher than InfiniBand latency. Thus, when a send completion event is delivered to a guest domain, a reply may have already come back from the other side. Therefore, the guest domain can process two completions with a single inter-domain communication operation, which results in reduced latency. For small messages, event batching happens very often. As message size increases, it becomes less and less frequent and the difference between Xen-IB and native IB increases.

6.4 Memory Registration

Memory registration is generally a costly operation in InfiniBand. Figure 15 shows the registration time of Xen-IB and native InfiniBand. The benchmark registers and unregisters a trunk of user buffers multiple times and measures the average time for each registration.

As we can see from the graph, Xen-IB adds consistently around 25%-35% overhead to the registration cost. The overhead increases with the number of pages involved in registration. This is because Xen-IB needs to use inter-domain communication to send a message which contains machine addresses of all the pages. The more pages we register, the bigger the size of message we need to send to the device domain through the inter-domain device channel. This observation indicates that

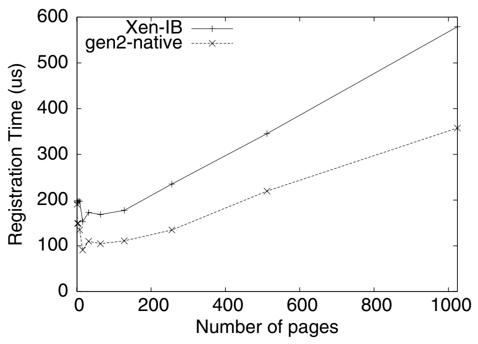

Figure 15: Memory Registration Time

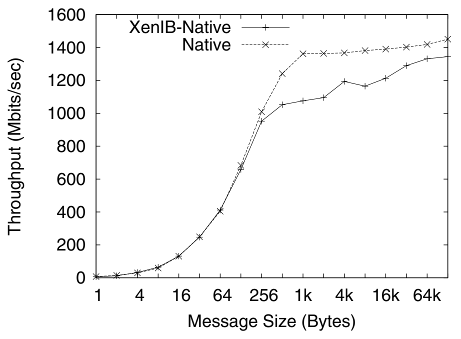

Figure 16: IPoIB Netperf Throughput

if the registration is a time critical operation of an application, we need to use techniques such as an efficient implementation of registration cache [38] to reduce costs.

6.5 IPoIB Performance

IPoIB allows one to run TCP/IP protocol suites over InfiniBand. In this subsection, we compared IPoIB performance between Xen-IB and native InfiniBand using Netperf [2]. For Xen-IB performance, the netperf server is hosted in a guest domain with Xen-IB while the client process is running with native InfiniBand.

Figure 16 illustrates the bulk data transfer rates over TCP stream using the following commands:

```
netperf -H $host -l 60 -- -s$size -S$size
```

Due to the increased cost of interrupt/event processing, we cannot achieve the same throughput while the server is hosted with Xen-IB compared with native InfiniBand. However, Xen-IB is still able to reach more than 90% of the native InfiniBand performance for large messages.

We notice that IPoIB achieved much less bandwidth compared with raw InfiniBand. This is because of two reasons. First, IPoIB uses InfiniBand unreliable datagram service, which has significantly lower bandwidth than the more frequently used reliable connection service due to the current implementation of Mellanox HCAs. Second, in IPoIB, due to the limit of MTU, large messages are divided into small packets, which can cause a large number of interrupts and degrade performance.

Figure 17 shows the request/response performance measured by Netperf (transactions/second) using:

```
netperf -l 60 -H $host -tTCP_RR -- -r $size,$size
```

Again, Xen-IB performs worse than native InfiniBand, especially for small messages where interrupt/event cost plays a dominant role for performance. Xen-IB performs more comparable to native InfiniBand for large messages.

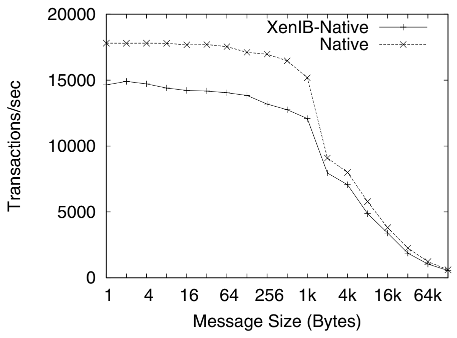

Figure 17: Netperf Transaction Test

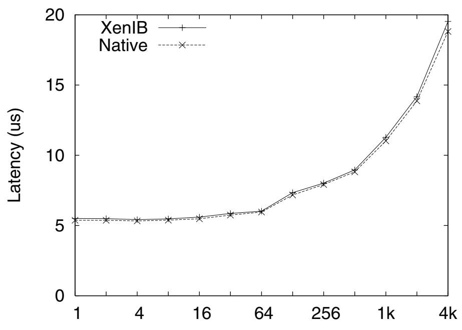

Figure 18: MPI Latency

6.6 MPI Performance

MPI is a communication protocol used in high performance computing. For tests in this subsection, we have used MVAPICH [27, 20], which is a popular MPI implementation over InfiniBand.

Figures 18 and 19 compare Xen-IB and native InfiniBand in terms of MPI one-way latency and bandwidth. The tests were run between two physical machines in the cluster. Since MVAPICH uses polling for all underlying InfiniBand communication, Xen-IB was able achieve the same performance as native InfiniBand by using VMM-bypass. The smallest latency achieved by MPI with Xen-IB was 5.4 μs. The peak bandwidth was 870 MBytes/s.

Figure 20 shows performance of IS, FT, SP and BT applications from the NAS Parallel Benchmarks suite [26]

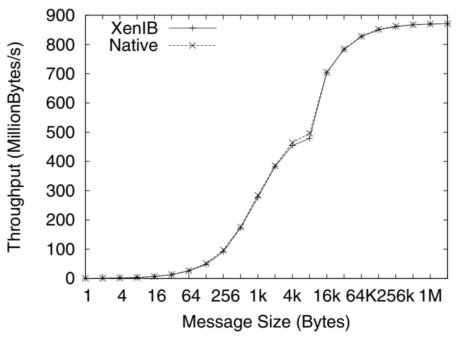

Figure 19: MPI Bandwidth

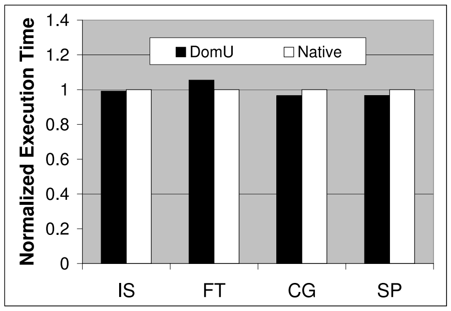

Figure 20: MPI NAS Benchmarks

(class A), which is frequently used by researchers in the area of high performance computing. We show normalized execution time based on native InfiniBand. In these tests, two physical nodes were used with two guest domains per node for Xen-IB. For native InfiniBand, two MPI processes were launched for each node. We can see that Xen-IB performs comparably with native InfiniBand, even for communication intensive applications such as IS. IB-Xen performs about 4% worse for FT and around 2–3% better for SP and BT. We believe the difference is due to the fact that MVAPICH uses shared memory communication for processes in a single node. Although MVAPICH with Xen-IB currently does not have this feature, it can be added by taking advantage of the page sharing mechanism provided by Xen.

7 Related Work

In Section 2.1, we have discussed current I/O device virtualization approaches such as those in VMware Workstation [37], VMware ESX Server [42], and Xen [13]. All of them require the involvement of the VMM or a privileged VM to handle every I/O operation. In our VMM-bypass approach, many time-critical I/O operations can be executed directly by guest VMs. Since this method makes use of intelligence in modern high speed network interfaces, it is limited to a relatively small range of devices which are used mostly in high-end systems. The traditional approaches can be applied to a much wider ranges of devices.

OS-bypass is a feature found in user-level communication protocols such as active messages [41], U-Net [40], FM [29], VMMC [6], and Arsenic [33]. Later, it was adopted by the industry [12, 19] and found its way into commercial products [25, 34]. Our work extends the idea of OS-bypass to VM environments. With VMM-bypass, I/O and communication operations can be initiated directly by user space applications, bypassing the guest OS, the VMM, and the device driver VM. VMM-bypass also allows an OS in a guest VM to carry out many I/O operations directly, although virtualizing interrupts still needs the involvement of the VMM.

The idea of direct device access from a VM has been proposed earlier. For example, [7] describes a method to implement direct I/O access from a VM for IBM mainframes. However, it requires an I/O device to be dedicated to a specific VM. The VMM-bypass approach not only enables direct device access, but allows for safe device sharing among many different VMs. Recently, the industry has started working on standardization of I/O virtualization by extending the PCI Express standard [30] to allow a physical device to present itself as multiple virtual devices to the system [31]. This approach can potentially allow a VM to directly interact with a virtual device. However, it requires building new hardware support into PCI devices while our VMM-bypass approach is based on existing hardware. At about the same time when we were working on our virtualization support for InfiniBand in Xen, others in the InfiniBand community proposed similar ideas [39, 22]. However, details regarding their implementations are currently not available.

Our InfiniBand virtualization support for Xen uses a para-virtualization approach. As a technique to improve VM performance by introducing small changes in guest OSes, para-virtualization has been used in many VM environments [8, 16, 44, 11]. Essentially, para-virtualization presents a different abstraction to the guest OSes than native hardware, which lends itself to easier and faster virtualization. The same idea can be applied to the virtualization of both CPU and I/O devices. Para-virtualization usually trades compatibility for enhanced performance. However, our InfiniBand virtualization support achieves both high performance and good compatibility by maintaining the same interface as native InfiniBand drivers at a higher level than hardware. As a result, our implementation is able to support existing kernel drivers and user applications. Virtualization at higher levels than native hardware is used in a number of other systems. For example, novel operating systems such as Mach [15], K42 [4], and L4 [18] use OS level API or ABI emulation to support traditional OSes such as Unix and Linux. Several popular VM projects also use this approach [10, 3].

8 Conclusions and Future Work

In this paper, we presented the idea of VMM-bypass, which allows time-critical I/O commands to be processed directly in guest VMs without involvement of a VMM or a privileged VM. VMM-bypass can significantly improve I/O performance in VMs by eliminating context switching overhead between a VM and the VMM or two different VMs caused by current I/O virtualization approaches. To demonstrate the idea of VMM-bypass, we described the design and implementation of Xen-IB, an VMM-bypass capable InfiniBand driver for the Xen VM environment. Xen-IB runs with current InfiniBand hardware and does not require modification to applications or kernel drivers which use InfiniBand. Our performance evaluations showed that Xen-IB can provide performance close to native hardware under most circumstances, with expected degradation on event/interrupt handling and memory registration.

Currently, we are working on providing check-pointing and migration support for our Xen-IB prototype. We are also investigating how to provide performance isolation by implementing QoS support in Xen-IB. In future, we plan to study the possibility to introduce VMs into high performance computing area. We will explore how to take advantages of Xen to provide better support of check-pointing, QoS and cluster management with minimum loss of computing power.

Acknowledgments

We would like to thank Charles Schulz, Orran Krieger, Muli Ben-Yehuda, Dan Poff, Mohammad Banikazemi, and Scott Guthridge of IBM Research for valuable discussions and their support for this project. We thank Ryan Harper and Nivedita Singhvi of IBM Linux Technology Center for extending help to improve the Xen-IB implementation. We also thank the anonymous reviewers for their insightful comments.

This research is supported in part by the following grants and equipment donations to the Ohio State University: Department of Energy's Grant #DE-FC02-01ER25506; National Science Foundation grants #CNS-0403342 and #CCR-0509452; grants from Intel, Mellanox, Sun, Cisco, and Linux Networx; and equipment donations from Apple, AMD, IBM, Intel, Microway, Pathscale, Silverstorm and Sun.

References

[1] IP over InfiniBand Working Group. http://www.ietf.org/-html.charters/ipoib-charter.html.

[2] Netperf. http://www.netperf.org.

[3] D. Aloni. Cooperative Linux. http://www.colinux.org.

[4] J. Appavoo, M. Auslander, M. Burtico, D. D. Silva, O. Krieger, M. Mergen, M. Ostrowski, B. Rosenburg, R. W. Wisniewski, and J. Xenidis. K42: an Open-Source Linux-Compatible Scalable Operating System Kernel. *IBM Sysmtems Journal*, 44(2):427–440, 2005.

[5] R. A. F. Bhoedjang, T. Ruhl, and H. E. Bal. User-Level Network Interface Protocols. *IEEE Computer*, pages 53–60, November 1998.

[6] M. Blumrich, C. Dubnicki, E. W. Felten, K. Li, and M. R. Mesarina. Virtual-Memory-Mapped Network Interfaces. In *IEEE Micro*, pages 21–28, Feb. 1995.

[7] T. L. Borden, J. P. Hennessy, and J. W. Rymarczyk. Multiple Operating Systems on One Processor Complex. *IBM System Journal*, 28(1):104–123, 1989.

[8] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems*, 15(4):412–447, 1997.

[9] P. M. Chen and B. D. Noble. When virtual is better than real. *Hot Topics in Operating Systems*, pages 133–138, 2001.

[10] J. Dike. User Mode Linux. http://user-mode-linux.sourceforge.net.

[11] 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.

[12] D. Dunning, G. Regnier, G. McAlpine, D. Cameron, B. Shubert, F. Berry, A. Merritt, E. Gronke, and C. Dodd. The Virtual Interface Architecture. *IEEE Micro*, pages 66–76, March/April 1998.

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

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

[15] D. B. Golub, R. W. Dean, A. Forin, and R. F. Rashid. UNIX as an application program. In *USENIX Summer*, pages 87–95, 1990.

[16] K. Govil, D. Teodosiu, Y. Huang, and M. Rosenblum. Cellular disco: resource management using virtual clusters on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 18(3):229–262, 2000.

[17] S. M. Hand. Self-paging in the nemesis operating system. In *Operating Systems Design and Implementation, USENIX*, pages 73–86, 1999.

[18] H. Hartig, M. Hohmuth, J. Liedtke, and S. Schonberg. The performance of micro-kernel-based systems. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 66–77, December 1997.

[19] InfiniBand Trade Association. InfiniBand Architecture Specification, Release 1.2.

[20] J. Liu, J. Wu, S. P. Kini, P. Wyckoff, and D. K. Panda. High Performance RDMA-Based MPI Implementation over InfiniBand. In *Proceedings of 17th Annual ACM International Conference on Supercomputing (ICS '03)*, June 2003.

[21] Mellanox Technologies. http://www.mellanox.com.

[22] Mellanox Technologies. I/O Virtualization with InfiniBand. http://www.xensource.com/company/xensummit.html-/Xen_Virtualization_InfiniBand_Mellanox_MKagan.pdf.

[23] Mellanox Technologies. Mellanox IB-Verbs API (VAPI), Rev. 1.00.

[24] Mellanox Technologies. Mellanox InfiniBand InfiniHost MT23108 Adapters. http://www.mellanox.com, July 2002.

[25] Myricom, Inc. Myrinet. http://www.myri.com.

[26] NASA. NAS Parallel Benchmarks. http://www.nas.nasa.gov-/Software/NPB/.

[27] Network-Based Computing Laboratory. MVAPICH: MPI for InfiniBand on VAPI Layer. http://nowlab.cse.ohio-state.edu/projects/mpi-iba/index.html.

[28] Open InfiniBand Alliance. http://www.openib.org.

[29] S. Pakin, M. Lauria, and A. Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM). In *Proceedings of the Supercomputing*, 1995.

[30] PCI-SIG. PCI Express Architecture. http://www.pcisig.com.

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

[32] I. Pratt. Xen Virtualization. Linux World 2005 Virtualization BOF Presentation.

[33] I. Pratt and K. Fraser. Arsenic: A User-Accessible Gigabit Ethernet Interface. In *INFOCOM*, pages 67–76, 2001.

[34] Quadrics, Ltd. QsNet. http://www.quadrics.com.

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

[36] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI–The Complete Reference. Volume 1 - The MPI-1 Core, 2nd edition.* The MIT Press, 1998.

[37] 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.

[38] H. Tezuka, F. O'Carroll, A. Hori, and Y. Ishikawa. Pin-down cache: A virtual memory management technique for zero-copy communication. In *Proceedings of the 12th International Parallel Processing Symposium*, 1998.

[39] Voltaire. Fast I/O for Xen using RDMA Technologies. http://www.xensource.com/company/xensummit.html-/Xen_RDMA_Voltaire_YHaviv.pdf.

[40] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-level Network Interface for Parallel and Distributed Computing. In *ACM Symposium on Operating Systems Principles*, 1995.

[41] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *International Symposium on Computer Architecture*, pages 256–266, 1992.

[42] C. Waldspurger. Memory resource management in vmware esx server. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, 2002.

[43] A. Whitaker, M. Shaw, and S. Gribble. Denali: Lightweight virtual machines for distributed and networked applications. In *Proceedings of the USENIX Annual Technical Conference, Monterey, CA*, June 2002.

[44] 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.