High Performance VMM-Bypass I/O in Virtual Machines

1. Introduction

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. In VMware ESX Server, guest VM I/O operations trap into the VMM, which makes direct access to I/O devices. In Xen, 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 5 us) and very high bandwidth (multiple Gbps) are emerging. Examples of these high speed interconnects include Virtual Interface Architecture (VIA) , InfiniBand, Quadrics, and Myrinet . 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.

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 , 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 and Xen 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, 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.

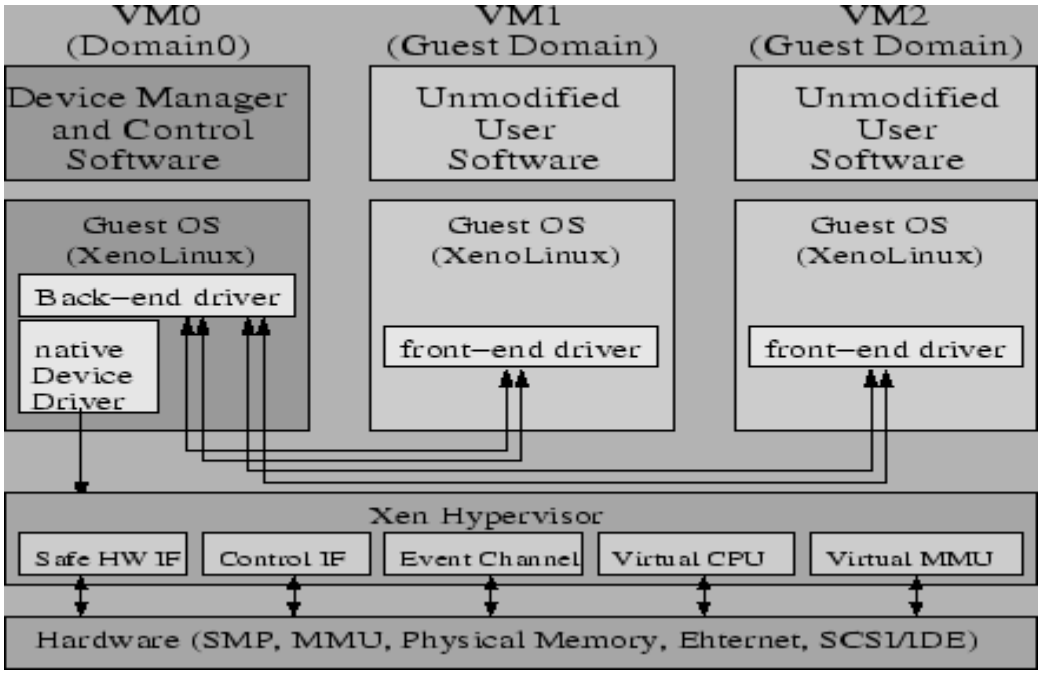

Figure 1: The structure of the Xen hypervisor, hosting three xenoLinux operating systems

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. 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. It can also result in *QoS crosstalk* 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

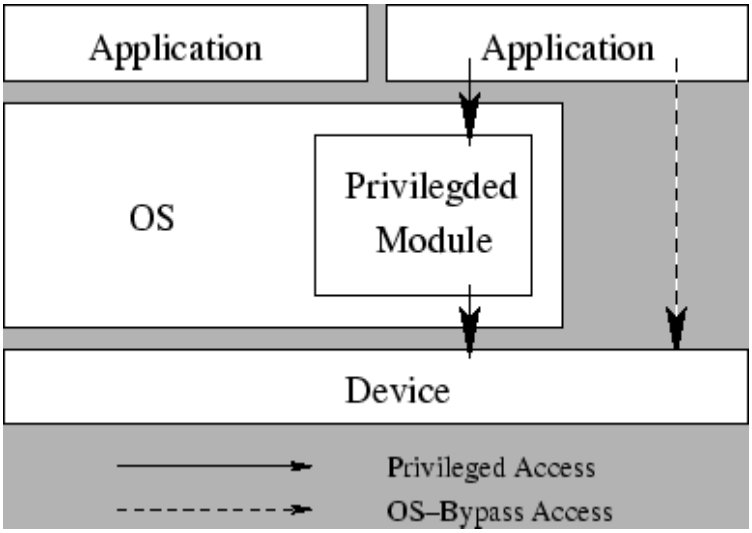

Figure 2: OS-Bypass Communication and I/O

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.

2.4 InfiniBand Architecture

InfiniBand 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.

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

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. 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) 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.

Our VMM-bypass I/O virtualization design is based on the idea of para-virtualization. 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 3

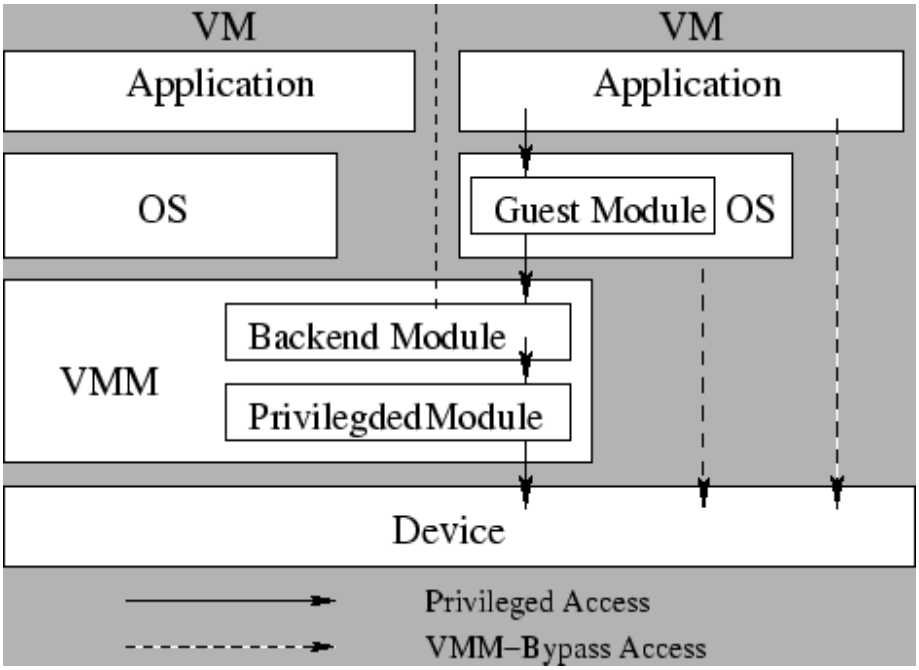

Figure 3: VM-Bypass I/O (I/O Handled by VMM Directly)

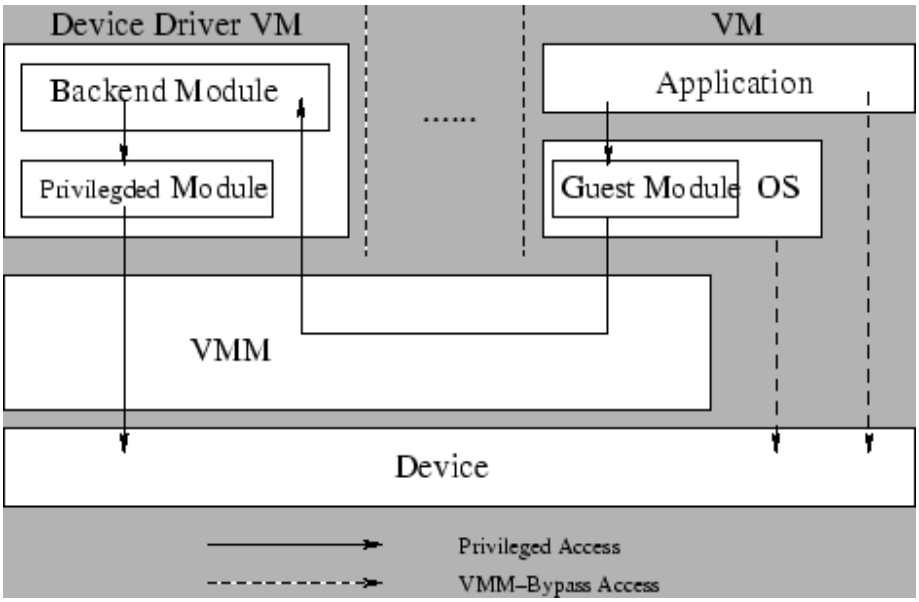

Figure 4: VM-Bypass I/O (I/O Handled by Another VM)

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 4

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.

Conclusions

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, people are working on Xen-IB, a 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.