InfiniBand Support in Xen Virtual Machine Environment

WEI HUANG, JIUXING LIU†, BULENT ABALI† AND DHABALESWAR K. PANDA

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

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

Technical Report
OSU-CISRC-10/05-TR63

InfiniBand Support in Xen Virtual Machine Environment *

Wei Huang Jiuxing Liu† Bulent Abali† Dhabaleswar K. Panda

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

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

Abstract

Virtual machine (VM) technologies provide the benefit of running multiple OS instances in a single physical machine, as well as other advantages such as performance isolation and the ease of management. Meanwhile, InfiniBand is becoming a strong player in the area of data center and High Performance Computing (HPC) due to its high performance and the features such as Remote Direct Memory Access (RDMA) and user space communication. Thus, supporting InfiniBand in a virtualized environment has become increasingly important. However, current device virtualization models require the involvement of a hypervisor and/or a privileged virtual machine (device domain), which may turn out to be a bottleneck for high performance communications.

In this paper, we present a novel design of Xen-IB, an InfiniBand device driver for a popular virtual machine monitor, Xen. We use a device domain bypass *approach to avoid the involvement of the hypervisor and the device domain for time critical operations. Xen-IB has been implemented for Xeno-Linux 2.6.12 in Xen 3.0. Our performance measurements show that the Xen-IB was able to achieve nearly the same raw performance as OpenIB Gen2 driver running in native Linux.*

*This research was initiated during Wei Huang's visit to IBM T. J. Watson Research Center as a summer intern. It is supported in part by Department of Energy's Grant #DE-FC02-01ER25506 and and National Science Foundation grants #CNS-0403342 and #CCR-0509452.

1 Introduction

With modern computers becoming increasingly powerful, virtual machine (VM) technologies are becoming more and more attractive to both the industry and the research communities. VM technologies allow many different virtual machines running in a single physical box, with each virtual machine possibly running a different operating system. VMs can also provide secure and portable environments to meet the demanding requirements of computing resources in modern computing systems [2].

Device I/O is handled in different ways in various VM technologies. For instance, in VMware Workstation [14], which provides full virtualization of the x86 architecture, device I/O relies on user level emulation or switching back to the host operating system. VMware ESX Server makes direct accesses to high performance I/O devices from the hypervisor [15]. In Xen [4], a high performance virtual machine monitor originally developed at the University of Cambridge, 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 domains) need to pass the I/O re-

quests to the IDD to access those devices. This control transfer between domains needs the involvement of the hypervisor and therefore has a performance penalty compared to native device operations.

The InfiniBand Architecture (IBA) [6] defines a System Area Network (SAN) for interconnecting processing nodes and I/O nodes. It provides memory semantics (RDMA) as well as channel semantics (send/receive). InfiniBand provides OS-bypass communication schemes. Many time critical communication operations can be performed directly from the user space without the overhead of going through operating systems. InfiniBand provides very low latency and high bandwidth, making it a strong player in the field of high performance cluster computing and I/O.

Supporting high performance communication devices such as InfiniBand in VMs is important in many aspects. First, VMs like Xen are becoming increasingly popular on modern servers and workstations. Many of these powerful computers are equipped with high speed interconnects like InfiniBand. As a result, providing efficient VM communication and I/O using these high speed interconnects is critical to fully achieve the benefit of virtualization. Further, InfiniBand is widely used in High Performance Computing (HPC) [13]. VM technologies may provide many valuable features for HPC, such as better checkpointing support, QoS and better cluster management. Although VMs currently are not widely used in HPC because of the overhead caused by virtualization, recently introduced high performance VMs like Xen, combined with a highly efficient InfiniBand support as demonstrated in this paper, can make VM based HPC a viable solution.

In this paper, we present a novel design of Xen-IB, an InfiniBand driver in the Xen virtual machine environment. Virtualizing an InfiniBand device poses unique challenges compared to other devices. Current device virtualization models in VMs require the involvement of either the hypervisor (VMware ESX) or the privileged domain (Xen and VMware workstation), making it difficult to support OS-bypass features of InfiniBand. To address this issue, we expanded the split device driver model used in Xen by adding IDD-bypass features, which allow time critical operations to be carried out directly within guest domains. With this optimization, we remove the bottleneck of going through the device domain and greatly improve the performance.

The main contributions of our work are:

- We designed and implemented an InfiniBand driver for Xen guest operating systems based on OpenIB Gen2 stack [10]. We keep the same interfaces as provided by Gen2, so existing programs written using the Gen2 verbs interface are transparently supported.
- We enhanced our driver with the device domain (IDD) bypass approach. For time critical operations, Xen guest applications are able to bypass the device domain and directly communicate with the IB adapters without violating the Xen security model.
- We evaluated the performance of our solution with a set of micro-benchmarks. The evaluation showed that our Xen-IB implementation provides similar performance compared to OpenIB driver running on native Linux systems.

The rest of the paper is organized as follows: In Section 2, we present background information of the Xen virtual machine monitor and the InfiniBand architecture. In Section 3, we discuss detailed design and implementation issues for the Xen-IB driver. Performance evaluation results are given in Section 4. We discuss related work in Section 5 and conclude the paper in Section 6.

2 Background

2.1 Overview of the Xen Virtual Machine Monitor

In this section we give a brief overview of the Xen virtual machine monitor (hypervisor). We

also describe Xen inter-domain communication and its split device driver model.

Xen uses paravirtualization [18], in which host operating systems need to be explicitly ported to the Xen architecture. This architecture is similar to native hardware such as x86 architecture, with only slight modifications to support efficient virtualization. Since Xen does not require changes to the application binary interface (ABI), existing applications can run without any modifications.

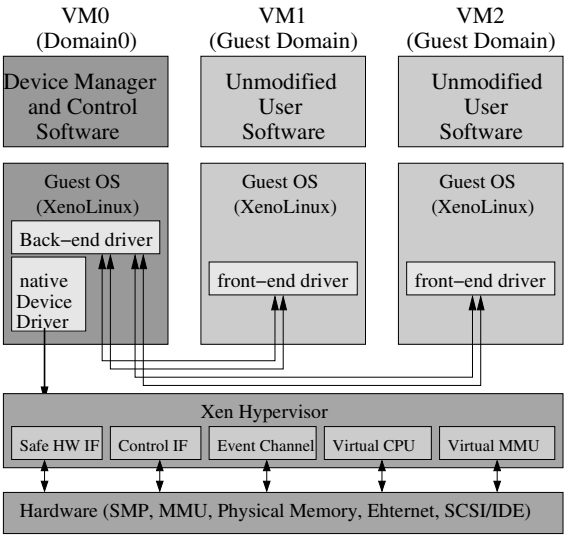

Figure 1. The structure of the Xen hypervisor, hosting three xenoLinux operating systems (courtesy [11])

Figure 1 illustrates the architecture of a physical machine running the Xen hypervisor. The Xen hypervisor is at the lowest level and has direct access to the hardware. Xen needs to ensure that the hypervisor, instead of the guest operating systems, is running in the most privileged processor-level. The x86 privilege levels are described by "rings", from ring 0 (the most privileged) to ring 3 (the least privileged). The hypervisor is running in ring 0. It provides basic control interfaces needed to perform complex policy decisions in Xen architecture. Above the hypervisor are the Xen domains. There can be many domains running simultaneously. Each domain hosts a guest operating system. Instead of running in ring 0, Guest OSes are modified to run in ring 1, which prevents them from directly executing the privileged processor instructions. Guest applications, like on normal x86 machines, run in ring 3 (the least privileged level). Only domain0, which is created at the boot time, is allowed to access the control interface provided by the hypervisor. The guest OS on domain0 hosts the application-level management software and perform the tasks to create, terminate or migrate other domains through the control interface provided by hypervisor.

There is no guarantee that a domain will get a continuous stretch of physical memory to run a guest OS. So Xen makes a distinction between *machine memory* and *pseudo-physical memory*. Machine memory refers to the physical memory installed in the machine. On the other hand, pseudo-physical memory is a per-domain abstraction, allowing a guest OS to treat its memory as consisting of a contiguous range of physical pages. Xen maintains the mapping between the machine memory and the pseudo physical memory. Only a certain specialized 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 communicate with each other through *event channels*. Event channels provide an asynchronous notification mechanism between Xen domains. Each domain has a set of end-points (or ports) which may be bounded to an event source (e.g. a physical IRQ, a virtual IRQ, or a port in another domain) [17]. When a pair of end-points in two different domains are bound together, a "send" operation on one side will cause an event to be received by the destination domain. 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 mem-

ory 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 [5]. Each device driver is expected to run in an *isolated device domain (IDD)*, which also hosts a *back-end* driver, running as a daemon and serving the access requests from guest domains. Each guest OS uses a *front-end* driver to communicate with the back-end. The front-end and back-end drivers communicate through the shared memory page and the event channel mechanism described above. The split driver organization provides security: misbehaving code in a guest domain will not result in failure of other guest domains. To improve the throughput for devices that need large data transfer such as network I/O, only the request descriptors can be passed to the IDD. The actual data payload may be directly DMAed from the guest domain to the device. Since the split device driver model requires the development of front-end and back-end drivers for each individual device, not all devices are supported in guest domains right now.

2.2 InfiniBand Architecture

The InfiniBand communication stack consists of many layers. The interface presented by Channel adapters to consumers belongs to the transport layer. A queue-based model is used in this interface. A *Queue Pair (QP)* in InfiniBand Architecture 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 are submitted to the work queue. Once submitted, a Work Queue Request becomes a *Work Queue Element (WQE)*, and are executed by Channel Adapters. The completion of work queue elements is reported through *Completion Queues (CQ)*. Once a work queue element is finished, a *Completion Queue Entry (CQE)* is placed in the associated completion queue. A kernel application can subscribe for notifications from HCA and register a callback handler with CQ. Completion queue can also be accessed through polling to reduce latency.

Initiating data transfer (posting work requests) and completion of work requests notification (poll for completion) are time-critical tasks that need to be performed by the application. In the Mellanox [8] approach, which represents a typical implementation of InfiniBand specification, these operations are done by ringing a doorbell. Doorbells are rung by writing to the registers that form the *User Access Region (UAR)*. UAR is memory-mapped directly from a physical address space that is provided by HCA. It allows access to HCA resources from privileged as well as unprivileged mode. Each UAR is a 4k page. Mellanox HCAs replicate the UARs up to 16M times. Each process using the HCA, no matter in kernel or user space, is allocated one UAR, which is remapped to the process's virtual address space. Posting a work request includes putting the descriptors (WQR) to QP buffer and writing the doorbell to the UAR. This process can be completed without the involvement of the operating system. Therefore it is very fast. 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.

Mellanox HCAs require all buffers involved in communication be registered before they can be used in data transfers. 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. always target to the correct position. Upon the success of registration, a local key and a remote key are returned. They will be used later for local and remote (RDMA) accesses. QP and CQ buffers described above are just normal buffers that are directly allocated from process virtual memory space and registered with HCA.

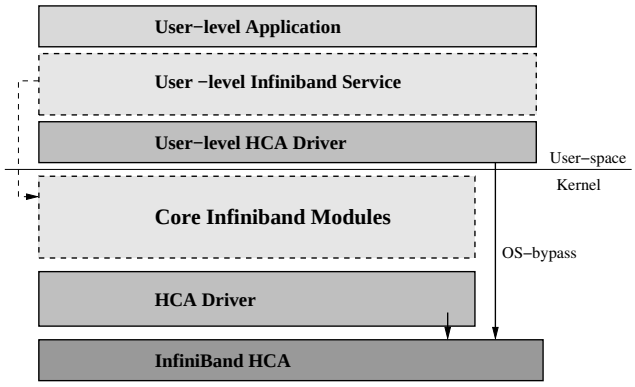

Figure 2. Architectural overview of OpenIB Gen2 stack

There are two popular stacks for InfiniBand drivers. VAPI [9] is the Mellanox implementation and OpenIB Gen2 [10] recently comes out as a new generation of IB stack provided by the OpenIB community. In this paper we implement InfiniBand drivers based on Gen2. Figure 2 represents the architecture of Gen2 stack.

3 Design and Implementation

In this section we present the design and implementation of Xen-IB, our InfiniBand driver for Xen. First we introduce a design that follows the traditional Xen split device driver model and the details of its major functional modules. Then we optimize the driver by directly accessing the HCA from guest domains for time critical tasks.

3.1 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 3 illustrates a basic design of our Xen-IB driver. The back-end runs as a daemon on top of the native InfiniBand driver in the isolated device domain (IDD). It waits for incoming connections from the front-end drivers in the guest domains. The front-end driver replaces the kernel HCA driver in normal OpenIB Gen2 stack. Thus the higher layer InfiniBand services can be supported smoothly in this design. Once the front-end driver is loaded, it establishes two event channels with the back-end daemon. The first channel, together with the shared memory page scheme described in Section 2.1, forms a device channel [5] which is used to process the requests initiated from the guest domain. The second channel is used for sending InfiniBand CQ and asynchronous events to the guest domain and will be discussed in detail later.

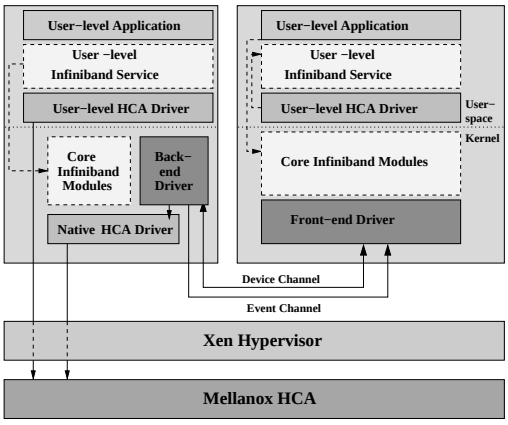

Figure 3. The Xen-IB driver structure with the split driver model

The Xen-IB front-end driver provides the same set of interfaces as normal Gen2 stack for kernel modules. It is a relatively thin layer whose tasks

include packing the command together with necessary parameters and sending it to the back-end through the device channel. The back-end driver re-constructs the commands, performs the operations with the native kernel HCA driver on behalf of the guest domain, and returns the result to the front-end driver. The back-end manages all InfiniBand resources on behalf of the guest domains. So all resources including QP/CQ buffers are actually allocated within the device domain. Only the handles to those resources are passed back to the front-end driver for later references.

The split device driver model in Xen poses difficulties for user-level direct HCA access in Xen guest domains. Since all resources are managed inside the device domain, the user-level HCA driver does not have direct access to the UARs as well as the QP/CQ buffers. It has to be changed so that the original OS-bypass operations should now also go through the kernel. To achieve this, we need to modify the user-level InfiniBand service module and the user access control part of the InfiniBand core modules.

3.2 Design Details

In the following, we first discuss in general how we support all InfiniBand operations. Then we explain in detail the implementation of major functional modules, including queue pair access, completion polling, memory registration and event handling.

Before applications can send messages 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 of the applications and can be implemented in a straightforward way. Basically, the guest domains forward these commands to the device domain (IDD) and wait for the acknowledgments after the operations are completed in IDD. All the resources are managed in IDD and the front-ends refer to these resources by handles. Validation checks must be conducted in IDD to ensure that all the references are legal.

Next we look at some key functional modules of our Xen-IB driver:

- **QP Access:** QP accesses (posting descriptors) include writing the WQEs to the QP buffer and ringing the doorbell to notify the HCA. Then the HCA will use DMA to transfer the WQEs to internal HCA memory and perform the send/receive or RDMA operations. Since all resources are located in the back-end driver, including the QP buffers and the UARs, it is required that the doorbell is rung at the device domain, and the WQEs also have to be accessible by the back-end only. Thus, once a post operation occurs, the front-end driver will pack and send to the back-end driver the content of work request and the lists of the buffers that need to be transferred. The back-end takes the responsibility to reconstruct the work requests and post them through the native HCA driver. Note that in this process, we only send the DMA addresses of the buffers. The actual data in the guest domains will be directly accessed by the HCA using DMA.

  The biggest problem for this approach, however, is its inefficiency. Involving the device domain can be very costly in terms of performance. And the need to change user-level access modules to support user-level verbs also makes implementation quite complicated.

  Meanwhile there are advantages of this approach. In Xen, all the guest domains share the same HCA in the physical box. So it will be beneficial if we can have global information of how the HCA is being used. By forcing every post to go into the device domain, the back-end driver may put the work requests from the guest domains into different request queues, which can be used to

build different kinds of scheduling schemes for better QoS.

- **CQ Polling:** Once the work request is completed, HCA will put a completion entry (CQE) in the CQ buffer. However, since CQ buffers are allocated in the device domain, the front-end driver cannot directly poll the CQ buffer in the guest domains. Therefore, every time the application tries to poll a CQ, The front-end driver has to send a request to the back-end driver. The back-end driver will poll the CQ. And if there is any CQE newly generated, it sends the CQE content back to the the front-end driver.

  Polling CQ is a quite frequently used verbs operation and usually lies in the critical path of the communication. Since it is not efficient to go to the back-end in every polling, we should optimize it as much as possible. One optimization to improve the efficiency is CQE buffering. Even though the front-end driver may only ask for a few number of WQEs in a poll CQ request, the back-end driver returns all the CQEs currently in the CQ buffer. The front-end then returns to the application the needed CQEs and buffers the rest of them. The next time the application polls CQ, the front-end driver does not need to communicate with the back-end if the needed CQEs have already been buffered locally. In this way we can amortize the cost of visiting the back-end driver.

- **Memory Registration:** The InfiniBand specification requires that all the memory regions involved in data transfers be registered with HCA. With Xen's paravirtualization approach, all domains see the same DMA address as the real machine address. So there will be no extra address translation needed here. 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 buffer. Again, the registration happens at the device domain. The front-end driver simply needs to send above information to the back-end driver and get back the local and remote keys. Note that since the Translation and Protection Table (TPT) on HCA is indexed by keys, we don't need to worry if multiple guest domains try to register with the same virtual address.

  For security reasons, the back-end driver can verify if the front-end driver offers valid DMA addresses belonging to the specific domain that it is running in. This check will make 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, which is described in section 2.2. Event handlers are associated with CQs or QPs when they are created. The application can subscribe for event notification by writing the appropriate command to the UAR page. When those subscribed events happen, the HCA driver will first get notified from HCA and then dispatch the event to different CQs or QPs according to the event type. Then the application that owns the CQ/QP will get a callback on their event handlers.

  For Xen-IB driver, event handling is a bit more complicated than other operations. Events are generated for the device domain, since all QPs and CQs are actually created there. 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 the front-end and the back-end driver. The back-end 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 the information such as the event type and the CQ/QP identifier to the guest domain through the event channel.

The front-end 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. Once the event dispatcher gets an event notification from the back-end driver, it checks the identifier and gives the corresponding CQ/QP a callback on the associated handler. In this process, the front-end driver has to maintain a translation table between the identifiers and the actual CQ/QPs.

3.3 An IDD-Bypass Approach for Improving Performance

The Xen-IB driver follows a traditional Xen split device driver model. Every HCA resource including QP buffers, CQ buffers, and User Access Regions (UAR), is allocated in the isolated device domain (IDD). So every operation including time critical ones have to go to IDD. As a result, there can be significant performance loss due to the context switch and inter-domain communication. For example, in the native IB drivers, applications can directly post or poll completion even from the user space. But now as shown in Fig. 4, since all access to UARs have to go to the device domain, all those operations have to go through the kernel of the guest domain, the kernel of the device domain, and comes all the way back, which adds large overhead to the latency.

Similar to the OS-bypass concept in InfiniBand user level access, we enhanced our driver by an IDD-bypass scheme that allows the time critical operations to be directly issued from guest domains. This poses two requirements on our design

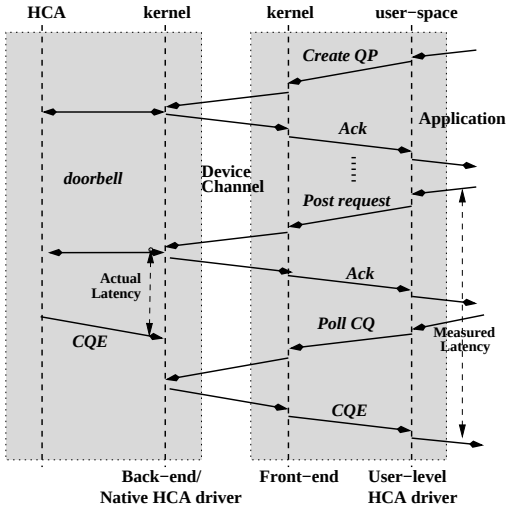

Figure 4. Working flow of the Basic Xen-IB driver

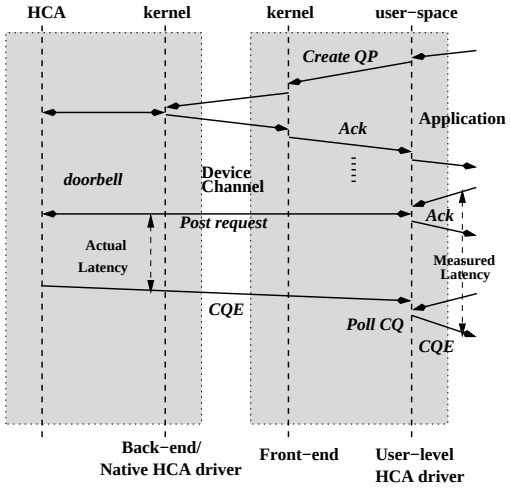

Figure 5. Working flow of the IDD-bypass Xen-IB driver

of Xen-IB driver: first, the UAR page must be accessible from the guest domain; second, both the QP and CQ buffers should be directly visible in the guest domain. Thus we changed the design of the front-end and the back-end drivers as follows:

- When a front-end driver is loaded, the back-end driver allocates a UAR and returns its page frame number (machine address) to the front-end. The front-end 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 applications. In the same way, when a user application starts, the front-end driver also applies a UAR page from the back-end and remaps the page to the application's virtual memory 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.

- Creating CQs/QPs is no longer simply sending the commands to the back-end. Instead, it is divided into 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 front-end 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 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. As shown in Fig. 5, the IDD-bypass scheme greatly reduces the communication latency.

The strategies for memory registration and event handling still fit in this IDD-bypass model. All other operations, such as creating and destroying QPs, still have to go through the device domain. But since usually these are not in critical path of communication, it has almost no impact on performance. We show the new stack of our Xen-IB driver in Figure 6.

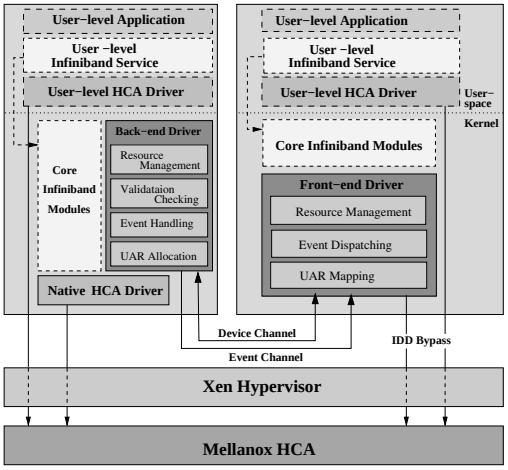

Figure 6. IDD bypass design of Xen-IB driver

Note that allowing the front-end driver to directly post requests does not mean that we are violating Xen rules to make validation checks. All QP/CQ creations and buffer registrations go through the back-end driver, which basically means all parts involved in the communication can be verified with the device domain.

4 Performance Evaluation

In this section, we evaluate the performance of our Xen-IB driver using a set of OpenIB Gen2 micro-benchmarks. We illustrate that the IDD-bypass scheme shows significant performance improvements over the basic design by using kernel-level RDMA latency and throughput tests. And with user-level micro-benchmarks, we show that the improved Xen-IB driver provides performance very close to native hardware, with only slight performance degradation for event handling and memory registration.

4.1 Testing Setup

We use two IBM xSeries 360 servers as our testbed. Each of them is equipped with an Intel Xeon 3.2GHz CPU, 1 GB memory and a Mellanox MHXL-CF128-T PCI-X InfiniBand HCA. The servers are connected with no switches in between. One of the servers (server A) is running Xen hypervisor with 2.6 kernel. RedHat AS4 are running both in domain0 and guest domains. OpenIB Gen2 stack is installed in domain0 as the native IB driver. The other server (server B) is running RedHat AS4 with 2.6.13 kernel. The same version of OpenIB Gen2 is installed on server B. Test environment settings on server B is fixed. Server A may have different settings to evaluate different schemes.

This asymmetric testing setup represents a scenario where servers in Xen guest domains use InfiniBand as an I/O transport to access remote devices which are hosted on a remote node that runs an unhypervised OS.

4.2 Benefi ts of the IDD-Bypass Driver

In this subsection, we compare the performance of the basic split driver design with the IDD-bypass design using kernel Gen2 micro-benchmarks. We use kernel level RDMA latency and throughput tests to evaluate our designs.

In RDMA latency test, server A writes to server B using RDMA, while server B polls for the arrival of the data and replies back with a RDMA write of the same size. Since our test setup is asymmetric, we measure the round trip time of this rendezvous process instead of the one way latency. In RDMA throughput test, server A, which is running Xen, issues RDMA writes back to back to server B. And we measure the maximum throughput we can achieve. However, we do not measure the throughput from server B to server A because there is only DMA operations at the target side of RDMA, in which our Xen-IB driver is not involved.

Fig. 7 and Fig. 8 show the results for the

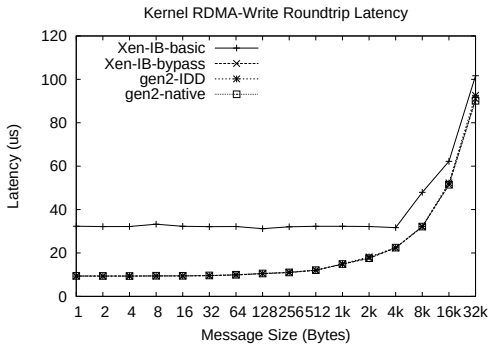

Figure 7. Kernel-level round-trip time for RDMA write operation

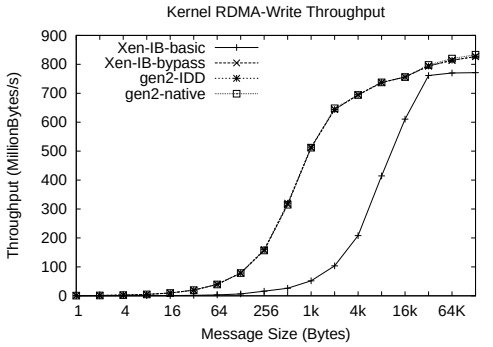

Figure 8. Kernel-level RDMA write Unidirectional throughput from server A to server B

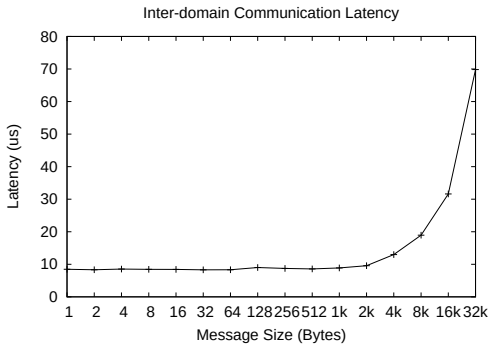

Figure 9. Inter-domain device channel one-way latency

RDMA latency and throughput tests. While the server B is always running native Linux, we compare the performance of kernel level RDMA write latency/throughput between the following four environments in server A: the basic Xen-IB driver in the guest domain (Xen-IB-basic), the IDD-bypass Xen-IB driver in the guest domain (Xen-IB-bypass), the native Gen2 driver in the device domain (gen2-IDD), and the native Gen2 driver on native Linux (gen2-native).

Note that the latency we show is the round-trip time. We can see that the basic Xen-IB driver has significantly higher latency than other drivers. The IDD-bypass driver, native driver in the device domain, and native driver on native Linux perform almost the same. We also show the one way latency of inter-domain communication on server A in Fig. 9, which explains the difference between the performance numbers. Going through the device domain adversely affect the throughput we can achieve. As shown in Fig. 8, the peak value with the basic Xen-IB driver is only 770 Million Bytes/s at 128 KB while we can achieve around 830 MB/s with all other drivers. With each post costing at least 10 μs (one way latency of inter-domain communication) to go to the device domain, it is almost impossible to post back-to-back sends to saturate the link unless the messages are sufficiently large.

4.3 Latency and Throughput

With kernel level benchmarks, we have already shown that involving the device domain in time critical operation adversely affects performance. So in the rest of this section, we focus on our IDD-bypass approach.

Fig. 10 and Fig. 11 show the latency and throughput for user-level micro-benchmarks. As in the kernel-level tests, the throughput test measures the uni-directional bandwidth from server A to server B.

As clearly shown in the figures, in both RDMA (indicated as rdma in figures) and send/receive (sr) tests, the IDD-bypass driver shows nearly the same performance as the native Gen2 driver in the device domain and the native Linux. This is not surprising because the critical path of these tests, posting descriptors and polling completions, is executed entirely in the guest domain, which can achieve performance close to native hardware. In the tests, we achieved 7.8 μs round-trip latency (3.9 μs one way) for 1 byte message and 830 MB/s throughput for 256k bytes message.

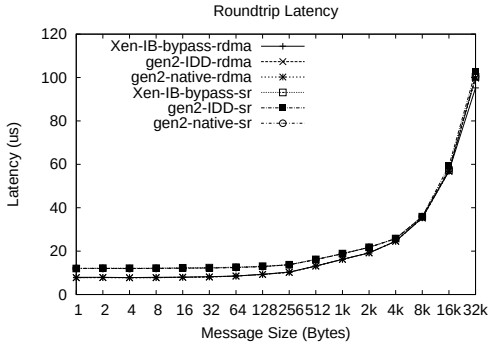

Figure 10. User-level round-trip communication time

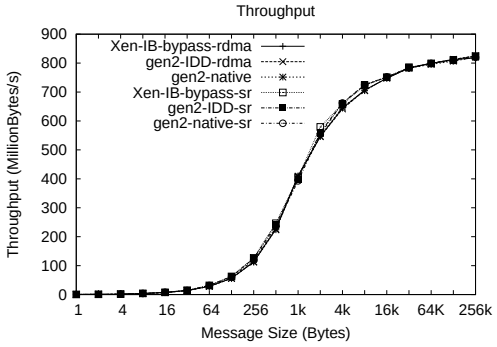

Figure 11. User-level Uni-directional throughput from server A to server B

4.4 Event Handling

The latency numbers we showed above are all based on polling schemes. In this section we show

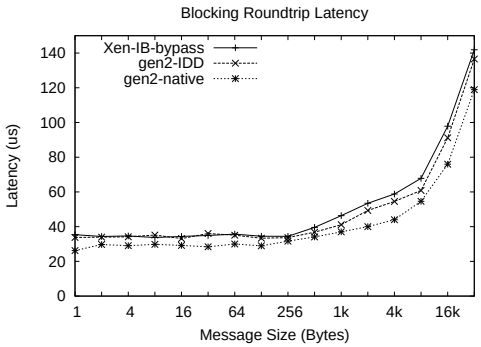

Figure 12. User-level send/recv round-trip time using blocking verbs

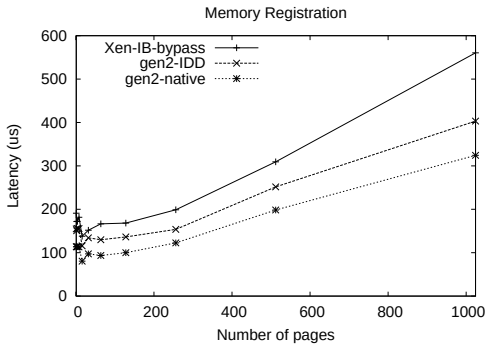

Figure 13. Memory registration time with respect to number of pages

the send/receive latency using blocking IB user-level verbs functions. This will reflect the efficiency of our event handling schemes.

Fig. 12 shows the send/receive round-trip latency using blocking verbs. The test is almost the same as send/receive ping pong test using polling. The only difference is that the process will block and wait for a completion event instead of busy polling on the completion queue. We show the results for server A running the IDD-bypass Xen-IB driver in the guest domain, the native IB driver in the device domain, and the native IB driver on native Linux. (Server B is running native Linux all the time.) Since Xen hypervisor is involved in the handling of device interrupts, round-trip latency is higher on Xen than native Linux, even with the native Gen2 driver. The extra cost of passing the event from the device domain to the guest domain adds another 2 to 10 μs to latency.

4.5 Memory Registration

Memory registration is generally a costly operation. Fig. 13 shows the registration time of the IDD-bypass driver, the native Gen2 driver in the device domain and the native Gen2 driver on native Linux. The benchmark allocates a trunk of user buffers, then registers and unregisters the buffers multiple times and measures the average time for each registration.

As we can see from the graph, Xen adds consistently around 25%-35% overhead to the registration cost no matter how many pages we register. The Xen-IB driver in the guest domain adds more overhead, which increases with the number of pages involved in registration. This is because the more pages we register, the bigger the size of scatter gather list we need to send to the device domain through the inter-domain device channel. The performance drop is hard to avoid because every single page registered with the HCA should be verified by the device domain.

This observation indicates that if the registration is a time critical operation of an application, an efficient implementation of registration cache [16] becomes important.

5 Related Work

The concept of virtual machines was first introduced by IBM in 1960s to provide a time-sharing access environment to mainframe hardware [3]. It has been applied to operating systems for more than 30 years. Today there are numbers of solutions to virtualize x86 architecture. Denali [18] [19] uses para-virtualization as Xen does. VMware and Microsoft Virtual PC provide full x86 virtualization, which supports unmodified guest operating systems. Device accesses on full virtualization usually require that the hypervi-

sor intercept all I/O operations issued by the guest OS.

Recently both Intel and AMD provide hardware assistance to ease the full virtualization of x86 processor architecture. Intel VT-x technology [7] consists of a set of virtual-machine extensions (VMX) that support virtualization of processor hardware for multiple software environments used by virtual machines. It will be supported in the next release of Xen [12]. Pacifica [1] is the AMD equivalent, which is claimed to be a functional superset of Intel VT. Though those hardware extensions make virtualization easier, the hypervisors still bear the responsibility for device I/O management.

There has been consistent effort to support high performance I/O devices in virtualized environment. Approaches used in Xen are discussed in Section 2. VMware workstation understands the semantics of I/O operations. Any accesses that actually interact with the physical hardware must go to the the VMApp that runs in the host domain. Otherwise, the operations can be handled directly in the hypervisor. In VMware ESX Server, I/O devices are accessed from the hypervisor with the help of a global component that recognizes each individual virtual machine [15]. In either models, the I/O accesses need to involve the hypervisor, thus make it hard to support the OS-bypass features of InfiniBand.

6 Conclusions and Future Work

In this paper, we presented the design and implementation of Xen-IB, an InfiniBand driver for Xen virtual machine environment. We illustrated the general design of the driver and explained in detail all the key functional components. Our design bypasses the device domain for time critical tasks but still keeps every resource under the control of Xen. Our performance evaluations showed that the Xen-IB driver can provide performance close to native hardware under most circumstances, with expected degradation on event handling and memory registration.

In the future, we plan to support the full set of InfiniBand services in Xen-IB, including management datagram (MAD) services. Meanwhile, since Xen is a fast developing open source project, we are continuously following Xen's newest features and integrate them into our work. We also 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, Dan Poff, Mohammad Banikazemi, and Scott Guthridge of IBM Research for valuable discussions and their support.

References

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

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

[3] R. J. Creasy. The Origin of the VM/370 Time-sharing System. *IBM Journal of Research and Development*, 25(5):483–490, 1981.

[4] 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.

[5] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M̃. Williamson. Reconstructing I/O. Technical Report UCAM-CL-TR-596, University of Cambridge, UK, August 2004.

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

[7] Intel Corperation. White paper: Enhanced Virtualization on Intel Architecture-based Servers.

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

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

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

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

[12] I. Pratt, K. Fraser, S. Hand, C. Limpach, and A. Warfield. Xen 3.0 and the Art of Virtualization. In *Proceedings of Linux Symposium*, 2005.

[13] Top 500 Supercomputer Site. http://www.top500.com.

[14] VMware Virtual Infrastructure Software. http://www.vmware.com.

[15] 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.

[16] 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.

[17] University of Cambridge. Xen Interface Manual.

[18] A. Whitaker, M. Shaw, and S. D. Gribble. Denali: Lightweight Virtual Machines for Distributed and Networked Applications. Technical report, University of Washington, February 2002.

[19] 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.