HIGH PERFORMANCE NETWORK I/O IN VIRTUAL MACHINES OVER MODERN INTERCONNECTS

DISSERTATION

Presented in Partial Fulfillment of the Requirements for

the Degree Doctor of Philosophy in the

Graduate School of The Ohio State University

By

Wei Huang, M.Sc (Tech)

* * * * *

The Ohio State University

2008

Dissertation Committee:

Prof. D. K. Panda, Adviser

Prof. P. Sadayappan

Prof. F. Qin

Approved by

Adviser
Graduate Program in
Computer Science and
Engineering

ABSTRACT

With the increasing size and complexity of modern computing systems, a balance between performance and manageability is becoming critical to achieve high performance as well as high productivity computing. Virtual Machine (VM) technology provides various features that help management issues on large-scale computing systems; however, performance concerns have largely blocked the deployment of VM-based computing, especially in the High-Performance Computing (HPC) area.

This dissertation aims at reducing the virtualization overhead and achieving the co-existence of performance and manageability through VM technologies. We focus on I/O virtualization, designing an experimental VM-based computing framework, and addressing performance issues at different levels of the system software stack. We design VMM-bypass I/O, which achieves native I/O performance in VMs by taking advantage of the OS-bypass interconnects. Also with the OS-bypass interconnects, we propose high performance VM migration with Remote Direct Memory Access (RDMA), which drastically reduces the VM management cost. To further improve the communication efficiency on multi-core systems, we design Inter-VM Communication (IVC), a VM-aware communication library to allow efficient shared memory communication among VMs on the same physical host. Finally, we design MVAPICH2-ivc, an MPI library that is aware of VM-based environments and can transparently benefit HPC applications with our proposed designs.

The dissertation concludes that performance should no longer be a barrier to deploying VM-based computing, which enhances productivity by achieving much improved manageability with very little sacrifice in performance.

Dedicated to my parents, Zhijun and Shiqian;

To my wife, Na

ACKNOWLEDGMENTS

I have been lucky to spend my PhD years, which could be the most important five years in my life, with so many great people. While the few words below cannot fully express my gratitude, I write these paragraphs to thank them for their guidance, support and friendliness.

I would like to thank my adviser, Prof. D. K. Panda for his support, patience and guidance throughout the duration of my Ph.D. study. I'm greatly indebted to him for the time and efforts which he spent to lead me into the world of computer science research. I have been learning a lot from his hard work and dedication as an academic researcher.

I am really grateful to Dr. Jiuxing Liu and Dr. Bulent Abali from IBM T. J. Watson Research Center. Topics in this dissertation started as intern projects there and continued at OSU, which would not have been possible without their support and guidance. Especially thanks to Dr. Liu for the pleasant and enlightening discussions over the years.

I would also like to thank my committee members, Prof. P. Sadayappan and Prof. F. Qin for their valuable comments and suggestions.

I'm lucky to have collaborated closely with my colleagues: Matthew, Gopal, Dr. Weikuan Yu, Dr. Hyun-wook Jin, Dr. Karthikeyan Vaidyanathan, Lei, Qi, and Tejus. I would also like to thank them for innumerable discussions and collaborations. I also deeply appreciate the help and support from all my current and past colleagues: Dr. Jiuxing Liu, Dr. Jiesheng Wu, Dr. Pavan Balaji, Dr. Amith Mamidala, Dr. Sayantan Sur, Dr. Abhinav Vishnu, Weihang, Sundeep, Ranjit, Shuang, Ping, Savitha, Hari, Ouyang, KGK, and Jaidev, for their discussions on technical and non-technical issues, as well as their friendship.

Finally, I thank my family: my parents, Zhijun Huang and Shiqian Lou, and my wife, Na Li. My accomplishments are impossible without their love, support, and encouragement throughout the years.

VITA

November 25th, 1980	Born - Hangzhou, China
September 1999 - June 2003	Bachelor in Computer Science and Engineering, Zhejiang University, Hangzhou, China
September 2003 - August 2004	University Fellow, The Ohio State University
September 2003 - Present	Graduate Research Associate, The Ohio State University
June 2005 - September 2005	Summer Intern, IBM T. J. Watson Research Center, Hawthorne, NY
June 2006 - September 2006	Summer Intern, IBM T. J. Watson Research Center, Hawthorne, NY
June 2007 - September 2007	Summer Intern, IBM T. J. Watson Research Center, Hawthorne, NY

PUBLICATIONS

W. Huang, M. Koop, and D. K. Panda, "Efficient One-Copy MPI Shared Memory Communication in Virtual Machines", In *Proceedings of IEEE Conference on Cluster Computing (Cluster'08)*, Tsukuba, Japan, September, 2008

M. Koop, W. Huang, K. Gopalakrishnan, and D. K. Panda, "Performance Analysis and Evaluation of PCIe 2.0 and Quad-Data Rate InfiniBand", In *Proceedings of the 16th Annual IEEE Symposium on High-Performance Interconnects (Hoti'08)*, Palo Alto, CA, August, 2008

W. Huang, M. Koop, Q. Gao, and D. K. Panda, "Virtual Machine Aware Communication Libraries for High Performance Computing", In *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC'07)*, Reno, NV. November, 2007 (*Best Student Paper Finalist*)

W. Huang, Q. Gao, J. Liu, and D. K. Panda, "High Performance Virtual Machine Migration with RDMA over Modern Interconnects", In *Proceedings of IEEE Conference on Cluster Computing (Cluster'07)*, Austin, Texas. September, 2007 (*Selected as a Best Technical Paper*)

K. Vaidyanathan, L. Chai, W. Huang, and D. K. Panda, "Efficient Asynchronous Memory Copy Operations on Multi-Core Systems and I/OAT", In *Proceedings of IEEE Conference on Cluster Computing (Cluster'07)*, Austin, Texas, September, 2007

Q. Gao, W. Huang, M. Koop, and D. K. Panda, "Group-based Coordinated Checkpointing for MPI: A Case Study on InfiniBand", In *Proceedings of Int'l Conference on Parallel Processing (ICPP'07)*, XiAn, China, September 2007

W. Huang, J. Liu, M. Koop, B. Abali, and D. K. Panda, "Nomad: Migrating OS-bypass Networks in Virtual Machines", In *Proceedings of the 3rd ACM/USENIX Conference on Virtual Execution Environment (VEE'07)*, San Diego, CA. June, 2007

K. Vaidyanathan, W. Huang, L. Chai, and D. K. Panda, "Designing Efficient Asynchronous Memory Operations Using Hardware Copy Engine: A Case Study with I/OAT", In *Proceedings of International Workshop on Communication Architecture for Clusters (CAC)*, held in conjunction with IPDPS '07, March 2007

Q. Gao, W. Yu, W. Huang, and D. K. Panda, "Application-Transparent Checkpoint/Restart for MPI Programs over InfiniBand", In *Proceedings of International Conference on Parallel Processing (ICPP'06)*, Columbus, OH, August, 2006

M. Koop, W. Huang, A. Vishnu, and D. K. Panda, "Memory Scalability Evaluation of the Next-Generation Intel Bensley Platform with InfiniBand", In *Proceedings of the14th IEEE Int'l Symposium on Hot Interconnects (HotI14)*, Palo Alto, CA, August 2006

W. Huang, J. Liu, B. Abali, and D. K. Panda, "A Case of High Performance Computing with Virtual Machines", In *Proceedings of the 20th ACM International Conference on Supercomputing (ICS'06)*, Cairns, Queensland, Australia. June, 2006

J. Liu, W. Huang, B. Abali, and D. K. Panda, "High Performance VMM-Bypass I/O in Virtual Machines", In *Proceedings of USENIX Annual Technical Conference 2006 (USENIX'06)*, Boston, MA. May, 2006

W. Huang, G. Santhanaraman, H. -W. Jin, and D. K. Panda, "Design of High Performance MVAPICH2: MPI-2 over InfiniBand", In *Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGrid'06)*, Singapore, May, 2006

A. Vishnu, G. Santhanaraman, W. Huang, H. -W. Jin, and D. K. Panda, "Supporting MPI-2 One Sided Communication on Multi-Rail InfiniBand Clusters: Design Challenges and Performance Benefits", In *Proceedings of the International Conference on High Performance Computing (HiPC'05)*, Goa, India, December, 2005

W. Huang, G. Santhanaraman, H. -W. Jin, and D. K. Panda, "Design Alternatives and Performance Trade-offs for Implementing MPI-2 over InfiniBand", In *Proceedings of EuroPVM/MPI*, Sorrento, Italy, September 2005

S. Sur, A. Vishnu, H. -W. Jin, W. Huang, and D. K. Panda, "Can Memory-Less Network Adapters Benefit Next-Generation InfiniBand Systems?", In *Proceedings of the 13th Annual IEEE Symposium on High Performance Interconnects (HOTI'05)*, Palo Alto, CA, August, 2005

G. Santhanaraman, J. Wu, W. Huang, D. K. Panda, "Designing Zero-copy MPI Derived Datatype Communication over InfiniBand: Alternative Approaches and Performance Evaluation", The *Special Issue of the International Journal of High Performance Computing Applications (IJHPCA) on the Best Papers of EuroPVMMPI 2004*

W. Huang, G. Santhanaraman, H. -W. Jin, and D. K. Panda, "Scheduling of MPI-2 One Sided Operations over InfiniBand", In *Proceedings of Workshop on Communication Architecture on Clusters (CAC)* in conjunction with International Parallel and Distributed Processing Symposium (IPDPS), April, 2005

FIELDS OF STUDY

Major Field: Computer Science and Engineering

Studies in:

Computer Architecture	Prof. D. K. Panda
Software Systems	Prof. G. Agrawal
Computer Networks	Prof. D. Xuan

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

With ever-increasing computing power being demanded by modern applications, today's ultra-scale data centers and High Performance Computing (HPC) systems are being deployed with increasing size and complexity. These large scale systems require significant system management effort, including maintenance, reconfiguration, fault tolerance, and administration, which are not necessarily the concerns of earlier small scale systems. As a result, system manageability is widely considered a critical requirement for modern computing environments.

Recent Virtual Machine (VM) technology emphasizes ease of system management and administration. Through a Virtual Machine Monitor (VMM or hypervisor), which is implemented directly on hardware, this technology allows running multiple guest VMs on a single physical node, with each guest VM possibly running a different Operating System (OS) instance. VM technology separates hardware and software management and provides useful features including performance isolation, server consolidation and live migration [10]. VMs provide secure and portable environments to meet the demanding resource requirements of modern computing systems and have been widely adopted in industry computing environments, especially data centers.

The benefits of VMs are at the penalty of performance degradation, however, by adding a virtualization layer between the OS instances and the native hardware resources. To reduce performance overhead, modern VMMs such as Xen [12] and VMware [67] allow guest VMs to access hardware resources directly whenever possible. For example, a guest VM can execute all non-privileged instructions natively in hardware without intervention of the VMM. Since many CPU intensive workloads seldom use privileged operations, the CPU virtualization overhead can be extremely low. Unfortunately, such direct access optimizations are not always possible. I/O virtualization is an example. Because I/O devices are usually shared among all VMs in a physical machine, the VMM has to make sure that accesses to them are valid and consistent. Currently, this requires VMM intervention on every I/O access from guest VMs, which leads to longer I/O latency and higher CPU overhead due to the context switches between the guest VMs and the VMM. In some cases, this I/O virtualization overhead weighs heavily against the wider adoption of VM technology, especially in the area of High Performance Computing (HPC).

Modern computing systems, featured with multi-core architecture and high performance networks, bring both opportunities and challenges to VM technology. On one side, the intelligent modern high speed interconnects connecting parallel systems, such as InfiniBand [21], typically allow OS-bypass [5, 44, 49, 61, 62] accesses. OS-bypass carries time-critical operations directly from user space applications to hardware while still maintaining system integrity and isolation. Based on OS-bypass, we can design VMM-bypass I/O virtualization, removing the bottleneck of going through the VMM or a separate VM for many I/O operations, and significantly improving communication and I/O performance. This potentially eliminates the performance barrier that prevents performance-critical applications from taking advantage of various features available through VM technology. On the other hand, OS-bypass networks pose additional challenges to VM migration because not all I/O operations can be intercepted by the VMM. Multi-core architecture also increases the level of server consolidation, requiring much more efficient inter-VM communication.

This dissertation aims at designing an efficient I/O virtualization framework for current and next-generation computing systems with high speed interconnects and multi-core architecture. We mainly target HPC environments, where VM deployments are severely limited because of the performance concerns while the features of VMs are welcomed to help the management issues on increasingly larger systems. We focus on network I/O operations, and propose a high performance VMM-bypass I/O virtualization technique that achieves native-level performance in VM-based environments. We design efficient middleware to benefit applications and VM management through our proposed techniques. More specifically, we have addressed the following questions:

- How can we reduce the network I/O virtualization overhead in modern computing systems featuring high speed interconnects and multi-core architecture?
- How can we reduce the cost of inter-VM communication on the same physical host, which is especially important for multi-core architecture?
- How can we design efficient middleware so that applications running in VM-based environments can benefit transparently from our research?
- How can we further reduce the management overhead for VM-based computing environments by taking advantage of modern high speed interconnects?

The rest of this dissertation is organized as follows. In Chapter 2, we discuss the existing technologies which provide background information of our work, including VM technology, InfiniBand architecture and MPI. In Chapter 3, we define the problems that we are addressing in this dissertation. Chapters 4 - 8 discuss our detailed approaches and results. We introduce the open source software packages derived from our work in Chapter 9. Finally in Chapter 10, we conclude and discuss some future research directions.

CHAPTER 2

OVERVIEW OF VIRTUALIZATION AND HIGH PERFORMANCE NETWORKING TECHNOLOGIES

In this chapter, we discuss several important technologies that are related to our research, including VMs, OS-bypass I/O, InfiniBand and iWARP/10GibE architecture, and Message Passing Interface (MPI). This will provide appropriate background information for this dissertation.

2.1 Virtual Machines

First we introduce VM technology using Xen as an example. Xen [12] is a popular high performance virtual machine monitor originally developed at the University of Cambridge. We start with a brief overview of the Xen architecture. We then describe how I/O is virtualized through the Xen split device driver model. Finally we discuss how Xen performs VM migration.

2.1.1 Xen Architecture

Xen adopts para-virtualization [70], 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.

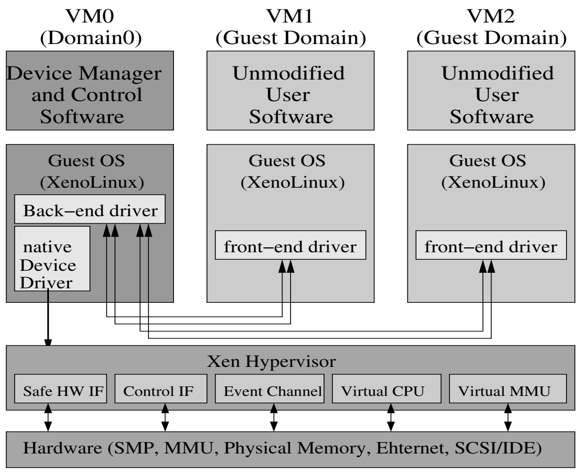


Figure 2.1: The Xen architecture (courtesy [48])

Figure 2.1 illustrates 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 (VMs). 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 (dom0), which is created at boot time, is allowed to access the control interface provided by the hypervisor. The guest OS on dom0 hosts the application-level management software and performs the tasks to create, terminate or migrate other domains through the control interface provided by the 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 (*machine frame number* or *mfn*) and the pseudo physical memory (*physical frame number* or *pfn*). Only certain specialized parts of the operating system need 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. To achieve efficient memory accesses in each domain, guest OSes allocate and manage their own hardware page tables. A read from a page table does not need to involve the Xen hypervisor, and thus introduces no overhead. However, all updates to the page tables must be verified by Xen to ensure safety and isolation. There are several restrictions for updating the page tables: a guest OS may only map pages that it owns, and page-table pages may only be mapped as read only, etc.

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 Interrupt Request (IRQ), a virtual IRQ, or a port in another domain) [58]. 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 one domain wants to send data to another one, 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.

2.1.2 Xen Device Access Models

VMs in Xen usually do not have direct access to hardware. Because 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 [14]. 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. This split driver organization provides security: misbehaving code in a guest domain will not result in failures 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 (Direct Memory Access) 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.

2.1.3 Virtual Machine Migration

Migration is one of the most important features provided by modern VM technology. It allows system administrators to move an OS instance to another physical node without interrupting any hosted services on the migrating OS. It is an extremely powerful cluster administration tool and serves as a basis for many modern administration frameworks which aim to provide efficient online system maintenance, reconfiguration, load balancing and proactive fault tolerance in clusters and data centers [7, 38, 55, 74]. In such management frameworks, it is desirable that VM migration be carried out in a very efficient manner, with both short migration times and low impacts on hosted applications.

When migrating a guest OS[1], Xen first enters the *pre-copy* stage, where all the memory pages used by the guest OS are transferred from the source to pre-allocated memory regions on the destination host. This is done by user level *migration helper processes* in Dom0s of both hosts. All memory pages of the migrating OS instance (VM) are mapped to the address space of the *helper processes*. After that the memory pages are sent to the destination host over TCP/IP sockets. Memory pages containing page tables need special attention that all machine dependent addresses (*mfn*) are translated to machine independent addresses (*pfn*) before the pages are sent. The addresses will be translated back to *mfn* at the destination host. This ensures transparency since guest OSes reference memory by *pfn*. Once all memory pages are

[1]For Xen, each domain (VM) hosts only one operating system. Thus, we do not distinguish among migrating a VM, a domain, and an OS instance in this dissertation.

transferred, the guest VM on the source machine is discarded, and the execution will resume on the destination host.

Xen also adopts live migration [10], where the pre-copy stage involves multiple iterations. The first iteration sends all the memory pages, and the subsequent iterations copy only those pages dirtied during the previous transfer phase. The pre-copy stage terminates when the page dirty rate exceeds the page transfer rate or when the number of iterations exceeds a pre-defined value. One of the benefits of live migration is that the guest OS is still active during the *pre-copy* stage, thus the actual *downtime* will be short. The only observed downtime is at the last iteration of pre-copy, when the VM is shut down to prevent any further modification to the memory pages.

2.2 OS-bypass I/O and InfiniBand Architecture

In this section we describe OS-bypass networks, InfiniBand, and iWARP/10GibE architecture.

2.2.1 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 [4]. 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 involving 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 2.2.

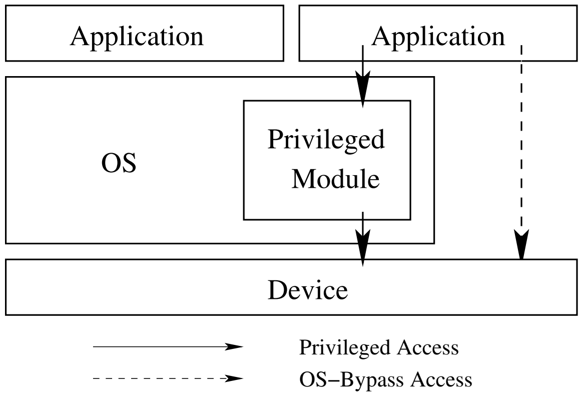


Figure 2.2: OS-Bypass Communication and I/O

The key challenge in implementing 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 land themselves nicely to a virtualized environment, as we will see later.

2.2.2 InfiniBand

InfiniBand [21] is a typical OS-bypass interconnect offering high performance. Figure 2.3 illustrates the 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 WQR becomes a *Work Queue Element (WQE)*, and are executed by Channel Adapters. The completion of WQEs is reported through *Completion Queues (CQs).* Once a WQE is finished, a *Completion Queue Entry (CQE)* is placed in the associated CQ. A kernel application can subscribe for notifications from InfiniBand Network Interface Card (NIC, or Host Channel Adapter, HCA) and register a callback handler with a CQ. A CQ 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

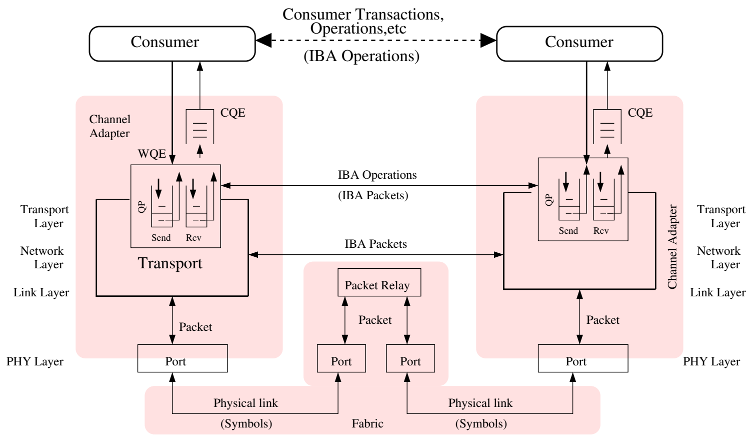


Figure 2.3: InfiniBand Architecture (Courtesy InfiniBand Specifications)

the application. In the Mellanox [30] 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 Host Channel Adapter (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, whether in kernel space 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) in QP buffer and writing the doorbell to the UAR. This process can be completed without the involvement of the operating system, so 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. 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. The QP and CQ buffers described above are just normal buffers that are directly allocated from process virtual memory space and registered with HCA.

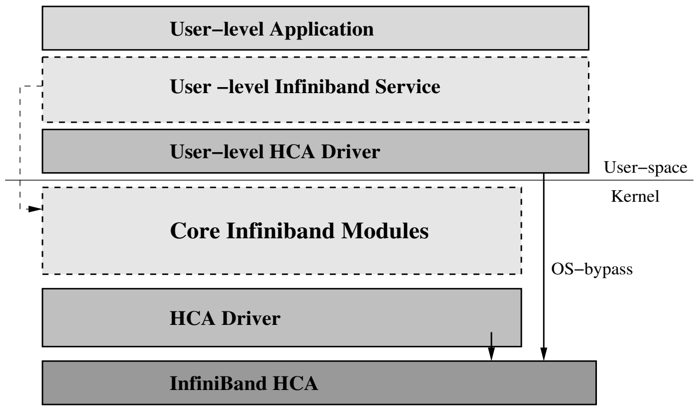


Figure 2.4: Architectural overview of OpenFabrics driver stack

OpenFabrics [42] is the most popularly used InfiniBand driver stack currently. Figure 2.4 presents its architecture.

2.2.3 iWARP/10GibE

iWARP stands for Internet Wide Area RDMA Protocol. It defines Remote Direct Memory Access (RDMA) operations over Ethernet networks [51]. Basically, iWARP allows for zero-copy data transfer over the legacy TCP/IP communication stacks. It bringing the benefits of OS-bypass to the Internet community, including significantly higher communication performance and low CPU overhead. Applications use the verbs API which provides RDMA support instead of traditional Socket API. As specified in [51], the basic message transport for iWARP is undertaken by the TCP layer. Since TCP itself is a stream protocol and does not respect message boundaries, an additional MPA layer is introduced to enforce this, as illustrated in Figure 2.5. The actual zero-copy capability is enabled by the Direct Data Placement (DDP) Layer. The RDMA features provided by DDP are exported to the upper level protocol by the RDMAP layer. Though iWARP can be implemented with either software [11] or hardware, it is intended to be implemented in hardware to achieve a significant performance improvement over the traditional TCP/IP stacks.

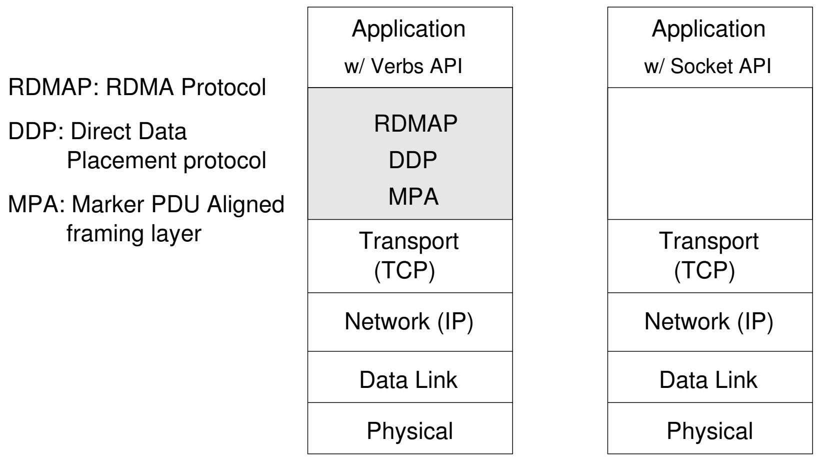


Figure 2.5: Compare iWARP with traditional TCP/IP layers (Courtesy [11])

The Chelsio [9] 10GibE adapter supports a hardware implementation of iWARP protocol. It supports both a private programming interface as well as the aforementioned OpenFabrics interface. We focus on the OpenFabrics interface in our research. From the iWARP perspective, the exposed software interface by OpenFabrics stack (verbs) to interact with hardware is very similar to the interface for Mellanox HCA, except for different layouts in the UARs or QP/CQ buffers etc. Thus, we will not provide additional details here.

2.3 Message Passing Interface (MPI), MPICH2 and MVAPICH2

MPI is currently the de facto standard for parallel programming. MPI-2 is the latest MPI standard and supports both point to point and one sided operations which are also referred to as Remote Memory Access (RMA) operations. MPICH2 is a portable implementation of MPI-2 from Argonne National Lab[2]. Figure 2.6 describes the layered approach provided by MPICH2 for designing MPI-2 over RDMA capable networks like InfiniBand.

Implementation of MPICH2 on InfiniBand can be done at one of the three layers in the current MPICH2 stack: RDMA channel, CH3 and ADI3. The decision is usually based on the trade-offs between communication performance and ease of porting.

RDMA channel is at the lowest position in the hierarchical structure of MPICH2. All the communication operations that MPICH2 supports are mapped to just five functions at this layer. In the five-function-interface only two (*put* and *read*) are communication functions, thus providing the least porting overhead. The interface needs to conform to stream semantics, which deliver data in FIFO order, and inform the upper layer of the number of bytes successfully sent. It is especially designed for

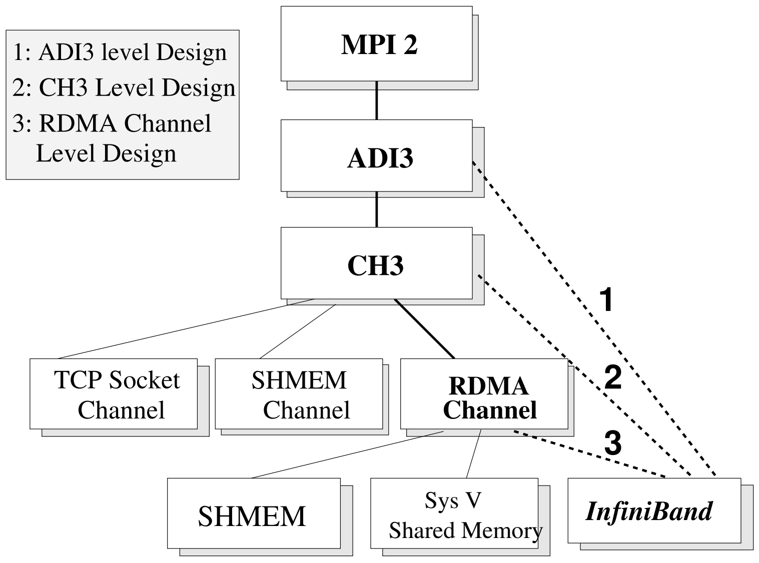


Figure 2.6: Layered design of MPICH2

architectures with RDMA capabilities, which directly fit with InfiniBand's RDMA semantics.

The CH3 layer provides a *channel* device that consists of a dozen functions. It accepts the communication requests from the upper layer and informs the upper layer once the communication has completed. The CH3 layer is responsible for making communication progress, which is the added complexity associated with implementing MPI-2 at this layer. But meanwhile, it can also access more performance oriented features than the RDMA channel layer. Thus, at this layer, we have more flexibility to optimize the performance.

The ADI3 is a full-featured, abstract device interface used in MPICH2. It is the highest layer in MPICH2 hierarchy. A large number of functions must be implemented to bring out a ADI3 design, but it can also provide flexibility for many optimizations, which are hidden from lower layers.

MVAPICH2 is a high performance implementation of MPI-2 over InfiniBand[69] from the Network Based Computing Laboratory at the Ohio State University. This software was originally designed based on the RDMA channel in MPICH2 [36], which is reflected in an earlier public released version MVAPICH2-0.6.5. In that design, we use eager and rendezvous schemes to support the communication interfaces of the RDMA channel. For small messages, we use the eager protocol. It copies messages to pre-registered buffers and sends them through RDMA write operations, which achieves good latency. And for large messages, a zero-copy rendezvous protocol is used, because using pre-registered buffers introduces high copy overhead. The user buffer is registered on the fly and sent directly through RDMA.

CHAPTER 3

MOTIVATION AND PROBLEM STATEMENT

In this chapter, we first motivate our research by discussing both the benefits and the performance limitations with current VM technology. Then we present the general research framework conducted in this dissertation.

3.1 Benefits of VM-based Computing Environments

System management capabilities are increased for a VM-based environment – including allowing special configurations and online maintenance. To demonstrate this, Figure 3.1 illustrates a possible VM-based deployment. Each computing node in the physical server pool is running a hypervisor that hosts one or more VMs[2]. A set of VMs across the cluster form a virtual cluster, on which users can host their multi-tier data centers or parallel applications. There can be a management console which performs all management activities, including launching, checkpointing, or migrating VMs across the cluster. To prepare a virtual cluster with a special configuration, all that is required is to choose a set of physical servers and launch VMs with the desired configurations on those servers. This avoids the lengthy procedure of resetting

[2]For performance-critical deployments such as HPC, each active VM runs on one or more dedicated cores, while more VMs can be consolidated on a single core for date centers.

all the physical hosts. More importantly, this does not affect other users using the same physical servers. Online maintenance is also simplified. To apply patches or update a physical node, system administrators no longer need to wait for application termination before taking them off-line. VMs on those servers can be migrated to other available physical nodes, reducing the administrative burden.

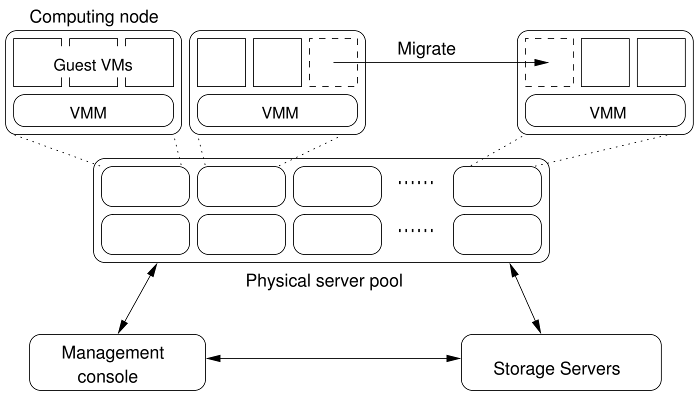


Figure 3.1: A possible deployment of HPC with virtual machines

Besides manageability, under certain circumstances VM environments can also lead to performance benefits. For example, with easy re-configuration it becomes much more practical to host HPC applications using customized, light-weight OSes. This concept of a customized OS has proven to be desirable for achieving increased performance in HPC [3, 13, 26, 34, 35]. VM migration can also help improve communication performance. For example, a parallel application may initially be scheduled onto several physical nodes far from each other in network topology due to node availability. But whenever possible, it is desirable to schedule the processes onto physical nodes near each other, which improves communication efficiency. Through VM migration, it is also possible to alleviate resource contention and thus achieve better resource utilization. For instance, VMs hosting computing jobs with a high volume of network I/O can be relocated with other VMs hosting more CPU intensive applications with less I/O requirements.

VM technology is also beneficial in many other aspects, including security, productivity, or debugging. Mergen et al. [34] have discussed in more detail the value of a VM-based HPC environment.

3.2 Performance Limitations of Current I/O Virtualization

While VM technology has shown its potential to relieve the manageability issues on large scale computing systems, concerns about performance block the community from embracing VM technologies in certain environments such as HPC. As we have discussed in Section 2.1, network I/O can incur high latency and high CPU utilization because of the VMM intervention. We find such overhead clearly observable in some cases.

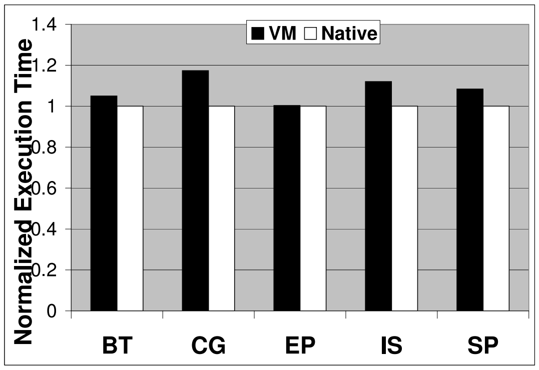


Figure 3.2: NAS Parallel Benchmarks

Figure 3.2 shows the performance of the NAS [39] Parallel Benchmark suite on 4 processes with MPICH [2] over TCP/IP. The processes are on 4 Xen DomUs, which are hosted on 4 physical machines. We compare the relative execution time with the same test on 4 nodes with native environment. For communication intensive benchmarks such as IS and CG, applications running in virtualized environments perform 12% and 17% worse, respectively. The EP benchmark, however, which is computation intensive with only a few barrier synchronizations, maintains the native level of performance. Table 3.1 illustrates the distribution of execution times collected by Xenoprof [33]. We find that for CG and IS, around 30% of the execution time is consumed by the isolated device domain (Dom0 in our case) and the Xen VMM for processing the network I/O operations. On the other hand, for the EP benchmark, 99% of the execution time is spent natively in the guest domain (DomU) due to a very low volume of communication. It should be noted that in this example, each physical node hosts only one DomU with one processor, even though the node is equipped with dual CPUs. If we run two DomUs per node to utilize both the CPUs, Dom0 starts competing for CPU resources with user domains when processing I/O requests, which leads to an even larger performance degradation as compared to the native case.

This example identifies I/O virtualization as the main bottleneck for virtualization, which leads to the observation that I/O virtualization with near-native performance would allow us to achieve application performance in VMs that rivals native environments.

Table 3.1: Distribution of execution time for NAS

	Dom0	Xen	DomU
CG	16.6%	10.7%	72.7%
IS	18.1%	13.1%	68.8%
EP	00.6%	00.3%	99.0%
BT	06.1%	04.0%	89.9%
SP	09.7%	06.5%	83.8%

3.3 Proposed High Performance I/O Virtualization Framework

The main objective of our research is to reduce the overhead of network I/O in VM environments. We mainly focus on HPC systems, where performance is one of the most critical demands.

Figure 3.3 shows the various components of the proposed research framework. In general, we work on the shaded boxes. The un-shaded ones represents the system components that need not be changed (End-user Application), or are being studied by other related research and industrial efforts (VM System Management [38, 55, 60]), which will not be the focus of our study. As can be seen, we work on multiple aspects of the I/O subsystems. At the device virtualization layer, we propose VMM-bypass I/O, which leverages the OS-bypass nature of modern high speed interconnects and achieves close-to-native level network I/O performance. The migration support for VMM-bypass I/O is carried out at the system level. Also at system level, we propose high performance inter-VM communication, which is especially important since multi-core systems are becoming ubiquitous. We also optimize the VM migration through RDMA, significantly reducing the management cost. On top of the system level, we work on high level services so that High End Computing (HEC) applications can transparently benefit from our research. We design a VM-aware MPI, which hides all the complexities of our research work from the end applications.

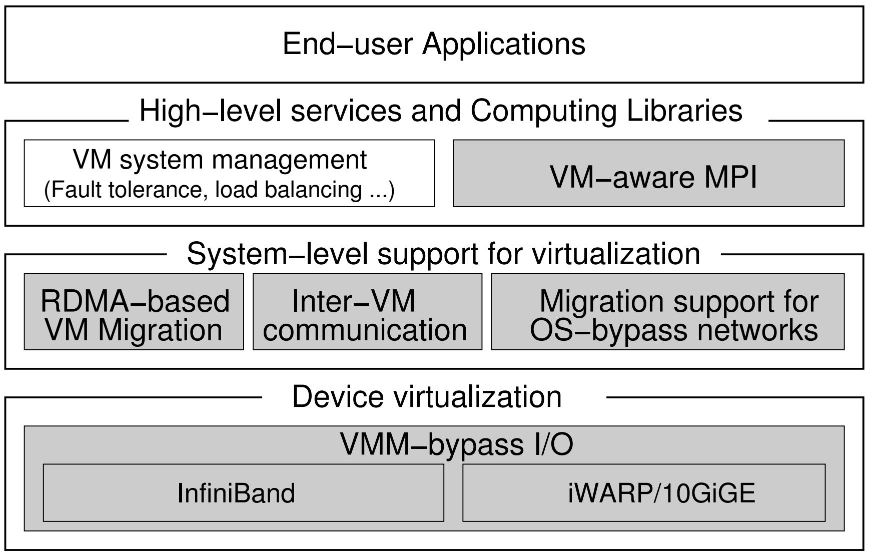

End−user Applications
High−level services and Computing Libraries
VM system management
(Fault tolerance, load balancing ...)
VM−aware MPI
System−level support for virtualization
RDMA−based VM Migration
Inter−VM communication
Migration support for OS−bypass networks
Device virtualization
VMM−bypass I/O
InfiniBand
iWARP/10GiGE

Figure 3.3: Proposed research framework

More specifically, we aim to address the following questions with this dissertation:

- **How can we reduce the network I/O virtualization overhead in modern computing systems featuring high speed interconnects?**

  To reduce the network I/O performance, we use a technique dubbed Virtual Machine Monitor bypass (VMM-bypass) I/O. VMM-bypass I/O extends the idea of OS-bypass I/O (as described in Section 2.2) in the context of VM environments. The key idea is that a guest module residing in the VM takes care of all the privileged accesses, such as creating the virtual access points (e.g. UAR for InfiniBand) and mapping them into the address space of the user processes. Thus a backend module provides such accesses for guest modules. This backend module can either reside in the device domain (such as Xen) or in the virtual machine monitor for other VM environments like VMware ESX server. The communication between the guest modules and the backend module is achieved through the inter-VM communication schemes provided by the VM environment. After the initial setup process, communication operations can be initiated directly from the user process. By removing the VMM from the critical path of communication, VMM-bypass I/O is able to achieve near-native I/O performance in VM environments. We will discuss VMM-bypass I/O in more details in Chapter 4.

- **How can we migrate network resources with VMM-bypass I/O capability?**

  The VMM-bypass I/O technique itself brings additional challenges. For example, migration is becoming difficult in such scenarios compared with VMs using traditional network devices such as Ethernet. First, the intelligent OS-bypass capable NICs manage large amounts of location-dependent resources which are kept transparent to both applications and operating systems and cannot be migrated with the VMs. This makes the existing solutions for current VM technologies, which target TCP/IP networks, less applicable. Further, applications in cluster environments using specialized interfaces of high speed interconnects typically expect reliable services at the NIC level. Thus, it is important to make migration transparent to applications and to avoid packet drops or out-of-order delivery during migration. We achieve migration of OS-bypass NICs by namespace virtualization and peer coordination, which will be described in detail in Chapter 5.

- **How can we reduce the cost of inter-VM communication on the same physical host, which is especially important for multi-core architectures?**

  Efficient inter-VM communication is another key issue for reducing the I/O virtualization overhead especially when multi-core architecture is leading to a high degree of server consolidation. In VM environments, management activities such as migration occur at the level of entire VMs. Thus, all processes running in the same VM must be managed together. To achieve fine-grained process-level control, each computing process has to be hosted in a separate VM, and therefore, in a separate OS. This presents a problem, because processes in distinct OSes can no longer communicate via shared memory [8], even if they share a physical host. This restriction is undesirable because communication via shared memory is typically considered to be more efficient than network loopback and is being used in most MPI implementations [8, 43]. This inefficiency is magnified with the emergence of modern multi-core systems. We present efficient inter-VM communication, which allows shared memory communication even for computing processes in different VMs, in Chapter 6. A further enhancement to inter-VM communication via one-copy shared memory will be presented in Chapter 7.

- **How can we design efficient middleware so that applications running in VM-based environments can benefit transparently from our research?**

  One philosophy for today's high performance virtualization techniques is para-virtualization, which allows certain changes to systems running in VM. Para-virtualization sacrifices transparency for performance. While it is always desirable to keep all design complexities away from end-user applications, application transparency is not always easy to achieve. For example, we propose our own APIs for the above-mentioned inter-VM communication. But in practice it is almost impossible to ask end applications to be modified to take advantage of it. As a result, our proposed techniques can be severely limited if we cannot make them transparent to end applications. Fortunately, most of today's HPC applications are written with a certain programming model. For parallel programming, MPI is the de facto standard. Most of the parallel HPC applications are written using MPI for communication among peer computing processes. As a result, by delivering an efficient MPI library which integrates our research, most applications can benefit while remaining unmodified. To achieve this goal, we design MVAPICH2, a high performance MPI2 library based by MPICH2 by extending the ADI3 layer. The goal of our design is to take advantage of the available communication methods on a cluster to achieve the best performance, such as shared memory for intra-node communication and InfiniBand for inter-node communication. We describe our design in Section 6.2.1.

  MVAPICH2 can also run in VM environments, however, its default shared memory communication channel can no longer be used if computing processes are hosted on different VMs. Thus, we design MVAPICH2-ivc, which is able to use inter-VM communication in VM-based environments. We present our design in detail in Section 6.2.2.

- **How can we further reduce the management overhead for VM-based computing environments by taking advantage of modern high speed interconnects?**

  To address this issue, we focus on VM migration in this dissertation, the basis for many management tasks in VM-based environments. We propose to use RDMA as the transfer mechanism for VM migration. RDMA can benefit VM migration in various aspects. First, with the extremely high throughput offered by high speed interconnects and RDMA, the time needed to transfer the memory pages can be reduced significantly, which leads to an immediate save on total VM migration time. Further, data communication over OS-bypass and RDMA does not need to involve CPUs, caches, or context switches. This allows migration to be carried out with minimum impact on guest operating systems and hosted applications. We discuss RDMA-based VM migration in Chapter 8.

CHAPTER 4

LOW OVERHEAD NETWORK I/O IN VIRTUAL MACHINES

In this chapter, we focus on reducing the network I/O virtualization overhead in VM environments, as well as its migration support. More specifically, we work on the highlighted part in Figure 4.1 of our proposed research framework.

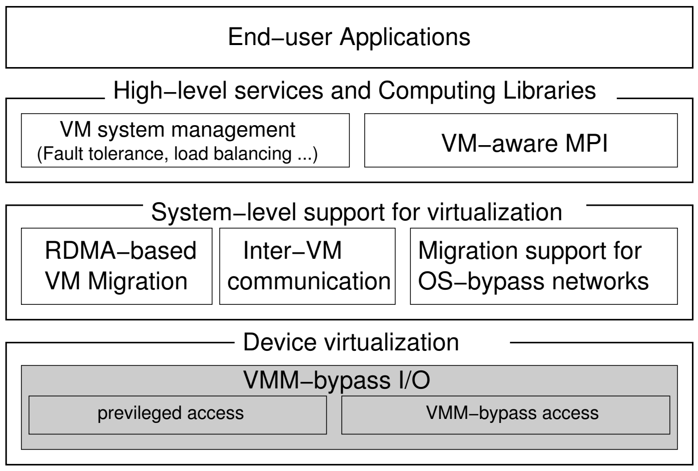


Figure 4.1: Network I/O in proposed research framework

We propose VMM-bypass I/O in Section 4.1, which achieves native-level network I/O by taking advantage of the OS-bypass features available on most modern high speed interconnects. We then discuss XenIB in Sections 4.2 and 4.3, which implements VMM-bypass I/O in Xen over InfiniBand. Finally, we carry out a performance evaluation in Section 4.4.

4.1 VMM-bypass I/O

VMM-bypass I/O can be viewed as an extension of 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 hardware level [57]. Each I/O instruction to access a device is virtualized by the VMM. With this approach, existing device drivers can be used in guest VMs without any modification. However, it significantly increases the complexity of virtualizing devices. For example, one popular InfiniBand HCA (MT23108 from Mellanox [32]) presents itself as a PCI-X device to the system. After initialization, it can be accessed by OS using memory mapped I/O. Virtualizing this device at hardware level would require us not only understand all hardware commands issued through memory mapped I/O, but also implement virtual PCI-X bus in guest VMs. 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, similar to [12] and [71]. We do not preserve hardware interfaces of existing devices. To virtualize a device in a guest VM, we implement a device driver called a *guest module* in guest OS. The guest module is responsible for handling all 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 4.2.

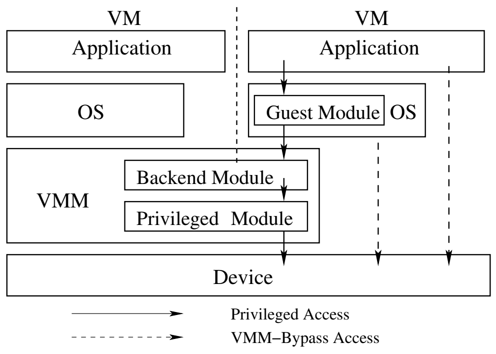


Figure 4.2: VMM-bypass (I/O Handled by VMM Directly)

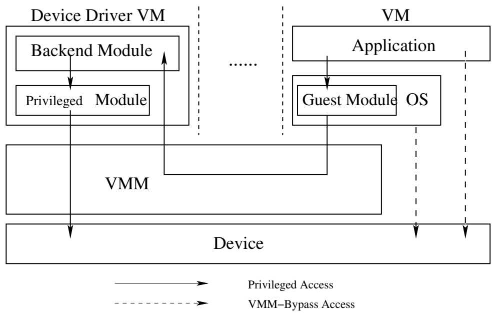


Figure 4.3: VMM-bypass (I/O Handled by Another VM)

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 4.3.

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.

4.2 XenIB: Prototype Design and Implementation

Now we introduce the design and implementation of Xen-IB, our InfiniBand virtualization driver for Xen. We describe details of the design and how we enable accessing InfiniBand HCAs from guest domains directly for time-critical tasks.

4.2.1 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 4.4 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 the last section, 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 [15] 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.

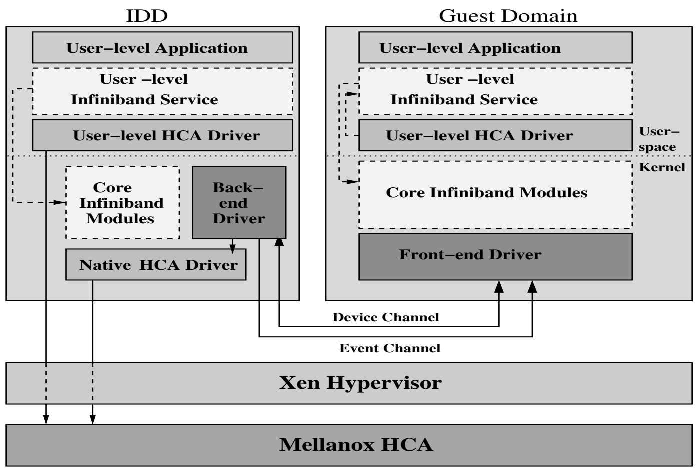


Figure 4.4: The Xen-IB driver structure with the split driver model

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.

4.2.2 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 resources are managed in backend and 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 para-virtualization 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 the above information to the backend driver and gets back the local and remote keys.

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 to 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. The 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.

4.2.3 VMM-Bypass Accesses

In InfiniBand, QP accesses (posting descriptors) include writing WQEs to the QP buffers and ringing the 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, the HCA will put a completion entry (CQE) in the CQ buffer. In InfiniBand, QP access functions are used for initiating communication. To detect the 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 this VMM-bypass design is shown in Figure 4.5.

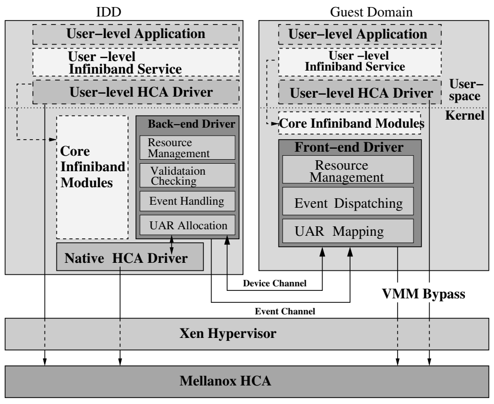


Figure 4.5: VMM-Bypass design of Xen-IB driver

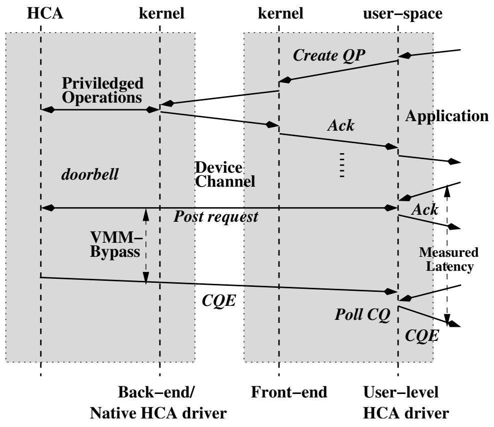


Figure 4.6: Working flow of the VMM-bypass Xen-IB driver

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 4.6.

It should be noted that since VMM-bypass accesses directly interact with HCAs, they are usually hardware dependent and 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.

4.2.4 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 an approach dubbed 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.

4.3 XenIB over iWARP/10GibE

To further demonstrate the concept of VMM-bypass I/O, we develop XenIB over iWARP/10GibE as well. Our implementation is based on the Chelsio 10GibE adapter [9]. Since the Chelsio adapter can run on the OpenFabrics stack, the stack on which we develop the InfiniBand support, most concepts that we used for InfiniBand are the same here, including QPs, CQs, UAR pages, etc. As a result, we use very similar design as for InfiniBand virtualization.

The additional complexity lies in management operations. The Chelsio 10GibE adapter does not use special QPs to send MADs. Instead, it sends raw packets over hardware to support a higher level OpenFabrics connection management service, called RDMA Connection Management Agent (RDMA-CMA). We use high-level virtualization, similar to the description in Section 4.2.4, to handle all the RDMA-CMA management requests to the Chelsio hardware in dom0.

4.4 Evaluation of VMM-bypass I/O

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. Finally, we look at the performance of Xen-IB over iWARP/10GibE.

4.4.1 Experimental Setup

Our experiments were conducted on two separate systems for InfiniBand and iWARP /10GibE. For InfiniBand related experiments, our testbed is a cluster with each node 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 the Linux 2.6.12 kernel. Xen 3.0 is used for all our experiments, with each guest domain running with single virtual CPU and 512 MB memory. We use MVAPICH-0.9.9 [37] to conduct the MPI level performance. For iWARP/10GibE experiments, we use two systems, each equipped with quad-core Intel Xeon 2.33GHz CPUs, 4GB memory, and a Chelsio cxgb3 10GibE adapter. Xen 3.1 is installed on these systems. And MVAPICH2-1.0 [37], which supports Chelsio Adapters through the iWARP interface, is used to get the MPI level performance.

4.4.2 InfiniBand Latency and Bandwidth

We first compare 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 4.7 and 4.8 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 4.9 and 4.10, we again see virtually no difference between Xen-IB and native InfiniBand.

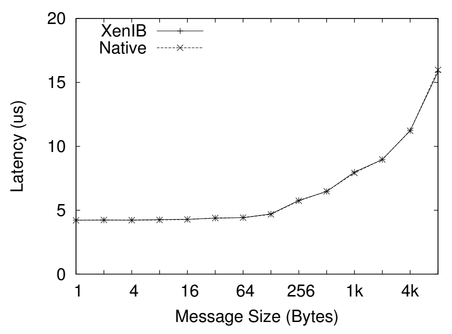


Figure 4.7: InfiniBand RDMA write latency

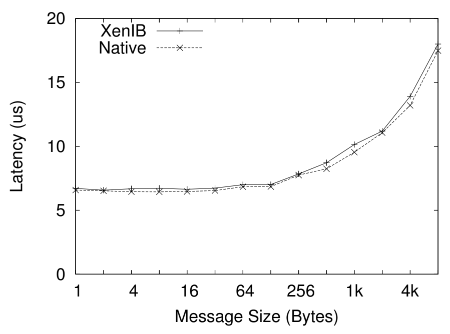


Figure 4.8: InfiniBand send/receive latency

Both of them were able to achieve bandwidth up to 880 MByte/s, which was limited by the bandwidth of the PCI-X bus.

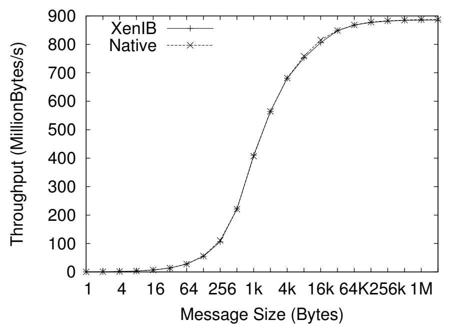


Figure 4.9: InfiniBand RDMA write bandwidth

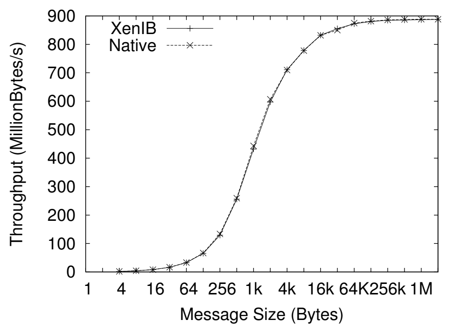


Figure 4.10: InfiniBand send/receive bandwidth

4.4.3 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 in

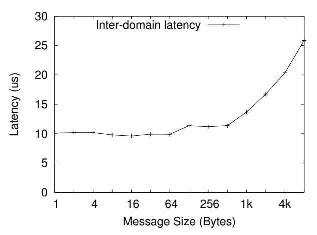


Figure 4.11: Inter-domain Communication One Way Latency

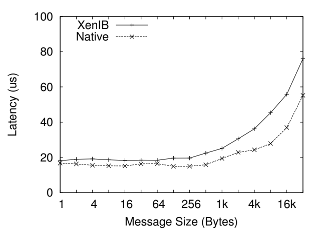


Figure 4.12: Send/Receive Latency Using Blocking VERBS Functions

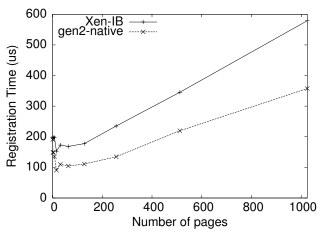


Figure 4.13: Memory Registration Time

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 4.11, 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 4.12 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 being 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.

4.4.4 Memory Registration

Memory registration is generally a costly operation in InfiniBand. Figure 4.13 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 if the registration is a time critical operation of an application, we need to use techniques such as an efficient implementation of registration cache [16] to reduce costs.

4.4.5 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 [1]. 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 4.14 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 for 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 4.15 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 comparably to native InfiniBand for large messages.

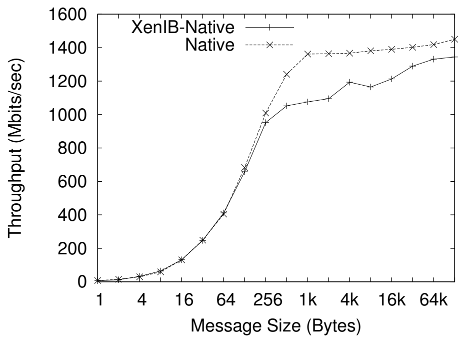


Figure 4.14: IPoIB Netperf Throughput

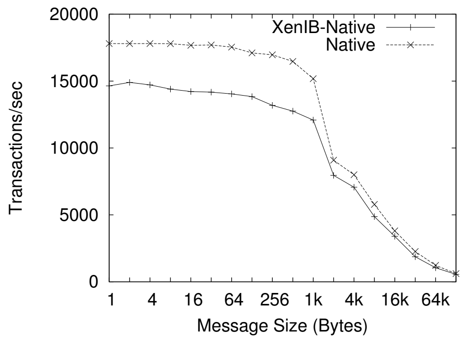


Figure 4.15: Netperf Transaction Test

4.4.6 MPI Performance

MPI is a communication protocol used in high performance computing. For tests in this subsection, we have used MVAPICH [37], which is a popular MPI implementation over InfiniBand developed by our research group.

Here we compare MPI-level micro-benchmark performance between the Xen-based cluster and the native InfiniBand cluster. Tests were conducted between two user domains running on two different physical machines with Xen or between two different nodes in native environments.

The latency test repeated ping-pong communication many times and the average half round-trip time is reported as one-way latency. As shown in Figure 4.16, there is very little difference between the Xen and the native environment, with both achieving 4.9μs for 1 byte messages. This shows the benefit of VMM-bypass I/O, where all communication operations in Xen are completed by directly accessing the HCAs from the user domains.

In the bandwidth tests, a sender sent a number of messages to a receiver using MPI non-blocking communication primitives. It then waited for an acknowledgment

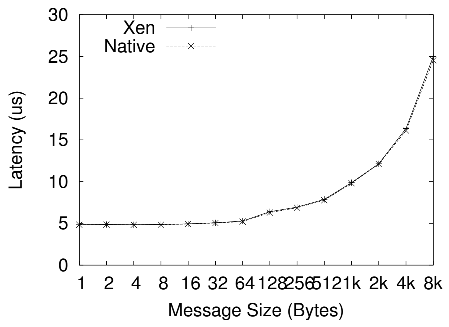


Figure 4.16: MPI latency test

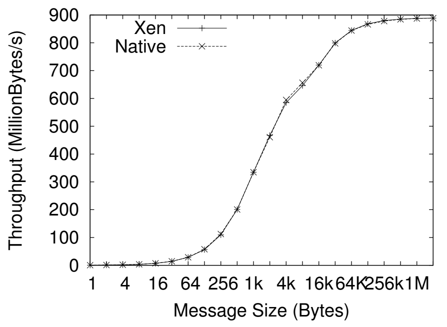


Figure 4.17: MPI bandwidth test

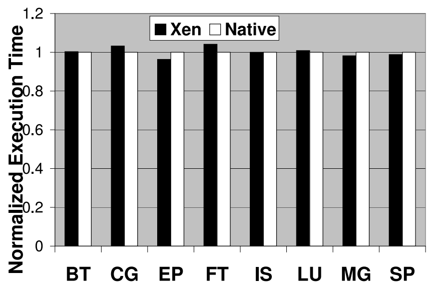
and reported bandwidth by dividing number of bytes transferred by the elapsed time. We again see almost no difference between Xen and native environments. As shown in Figure 4.17, in both cases we achieved 880MB/sec. [3]

We now evaluate the Xen-based computing environment with actual HPC applications. We instantiate two DomUs on each physical machine and run one process per DomU for the Xen case. (Each DomU runs a uni-processor kernel.) And for native environment, we run two processes per physical node.

We evaluate both NAS parallel benchmarks and High Performance Linpack (HPL). HPL is the parallel implementation of Linpack [45] and the performance measure for ranking the computer systems of the Top 500 supercomputer list.

The execution time for NAS has been normalized based on the native environment in Figure 4.18. We observed that the Xen-based environment performs comparably with the native environment. For NAS applications CG and FT, where the native environment performs around 4% better, the gap is due to the ability of MVAPICH to utilize shared memory communication for processes on the same node in the native

[3]Please note that all bandwidth numbers in this dissertation are reported in millions of bytes per second.



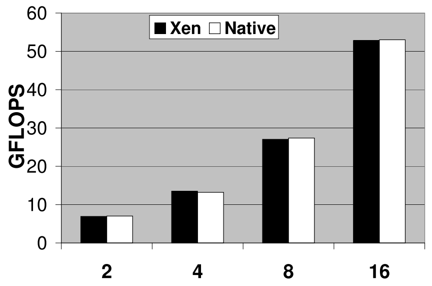


Figure 4.18: NAS Parallel Benchmarks (16 processes, class B)

Figure 4.19: HPL on 2, 4, 8 and 16 processes

Table 4.1: Distribution of execution time for NAS

	Dom0	Xen	DomUs
IS	3.6	1.9	94.5
SP	0.3	0.1	99.6
BT	0.4	0.2	99.4
EP	0.6	0.3	99.3
CG	0.6	0.3	99.0
LU	0.6	0.3	99.0
FT	1.6	0.5	97.9
MG	1.8	1.0	97.3

environment. Our prototype implementation of VM-based computing currently does not have this feature. However, high speed communication between Xen domains on the same physical node can be added by taking advantage of the page-sharing mechanism provided by Xen.

We also report the Gflops achieved in HPL on 2, 4, 8 and 16 processes in Figure 4.19. We observe very close performance here with the native environment outperforming by at most 1%.

Table 4.1 shows the distribution of the execution times among Dom0, Xen hypervisor, and two DomUs (sum) for NAS benchmarks. As we can see, due to VMM-bypass approach, most of the instructions are executed locally in user domains, which achieves highly efficient computing. For IS, the time taken by Dom0 and the Xen hypervisor is slightly higher. This is because IS has a very short running time and the Dom0/hypervisor is mainly involved during application startup/finalize, where all the connections need to be set up and communication buffers need to be pre-registered.

4.4.7 iWARP/10GibE Performance

Here we take a look at iWARP/10GibE performance with VMM-bypass virtualization.

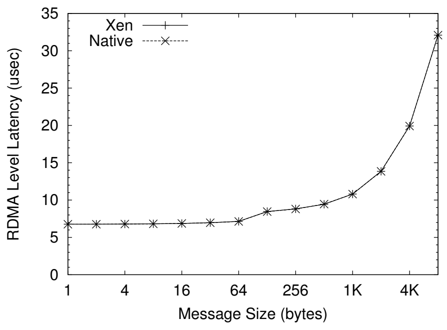


Figure 4.20: RDMA level latency test

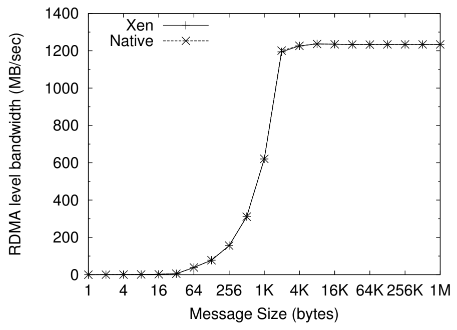


Figure 4.21: RDMA level bandwidth test

Figures 4.20 and 4.21 present the latency and bandwidth at the RDMA level. We compare the results of hosting testing processes on two Xen domUs on different physical hosts and between two native OSes. As on InfiniBand, the differences here are also negligible. In both cases, we are able to achieve 6.8μs latency and 1234MB/s bandwidth. Similarly, as shown in Figures 4.22 and 4.23, MPI level latency and bandwidth are practically the same at 7.3μs and 1233MB/s.

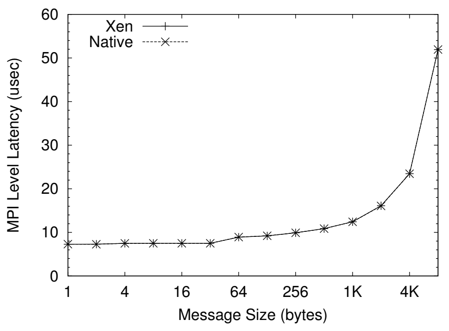


Figure 4.22: MPI level latency test

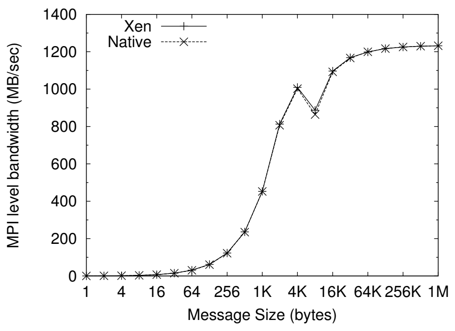


Figure 4.23: MPI level bandwidth test

Figure 4.24 shows the performance of typical MPI level collective operations using Intel MPI Benchmarks. These tests are conducted using 2 nodes with 4 cores each (in total 8 processes). We show performance is for small (16 bytes), medium (16KB), and large (256KB) size operations. Since the current Chelsio 10GibE adapter has limitations in setting up connections within one adapter, the processes on the same physical node are hosted in one domU for the Xen case. Communication among processes in the same domU is through shared memory instead of through network. This comparison is fair since shared memory communication is also used in native case when the processes are in the same OS. We observe that in most cases the performance numbers are close except for collectives involving reduce operations. We believe this is because reduce operations require allocating and access temporary buffers, where Xen memory management could induce some overhead.

We finally show the evaluation with NAS Parallel Benchmarks (class A) on 4 MPI processes in Figure 4.25. The results are normalized based on the performance

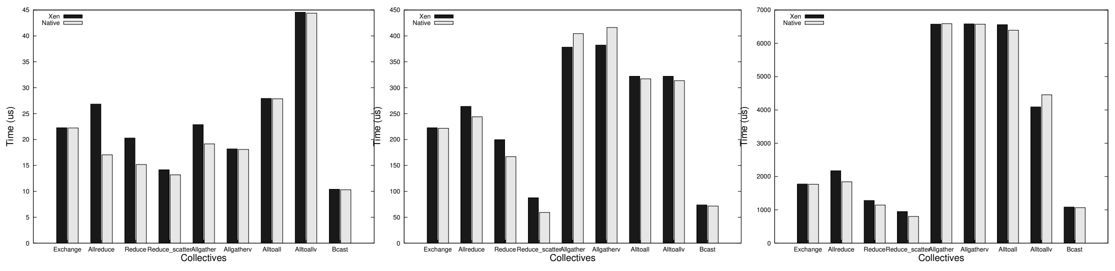

(a) 16 Bytes (b) 16K Bytes (c) 256K Bytes

Figure 4.24: MPI collective operations

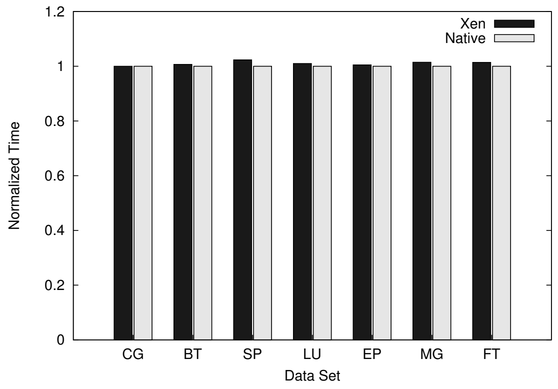


Figure 4.25: NAS Parallel Benchmarks (class A)

achieved in the native environment. Again, as shown, the performances are very close here.

In summary, our experiments with iWARP/10GibE together with the experiments on InfiniBand conclude that VMM-bypass I/O is an effective way to virtualize network interconnects with OS-bypass capabilities.

CHAPTER 5

NOMAD: MIGRATION OF VMM-BYPASS I/O NETWORKS

In this chapter we work on the highlighted part in Figure 5.1. VMM-bypass I/O, as discussed in the last chapter, is able to significantly reduce the I/O virtualization overhead. However, it also poses more challenges on VM migration. In the rest of this chapter, we first take a closer look at the challenges of migrating OS-bypass interconnects. Then we propose *Nomad*, a software framework to migrate OS-bypass networks and its implementation on XenIB. Finally we present the performance evaluation of Nomad.

5.1 Challenges for Migrating VMM-bypass Networks

Compared with traditional network devices such as Ethernet, migration of modern OS-bypass interconnects (thus virtualized through VMM-bypass I/O) have been studied less. There are multiple challenges in migrating VMM-bypass I/O in VMs:

5.1.1 Location-Dependent Resources

Many network resources associated with OS-bypass interconnects are location dependent, making them very difficult to migrate with the migrating OS instance with application transparency.

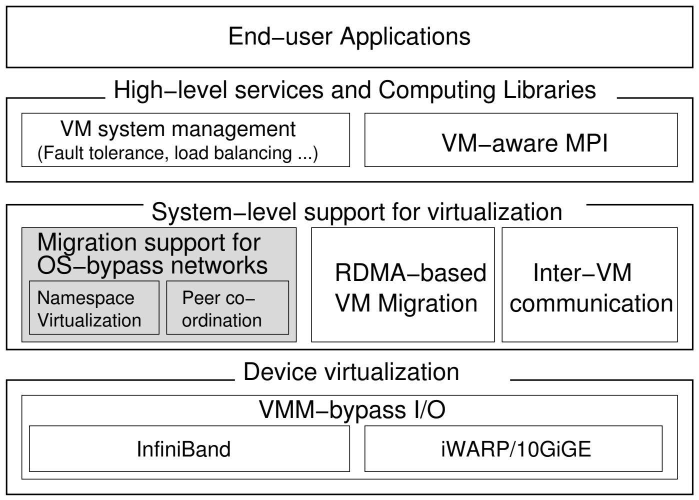


Figure 5.1: Network I/O in proposed research framework

First, as mentioned in Section 2.2 of Chapter 2, to allow user-level communication and remote memory access, the HCAs of modern interconnects will often manage some data structures in hardware. Software can use opaque handles to access HCA resources but cannot manage them directly. Once a VM is migrated to another physical machine with a different HCA, the opaque handles are no longer valid. One approach is to attempt to reallocate the resources on the new host using the same handle values. This method requires changes to the device firmware which assigns the handles. Even with that additional complexity, we will have a problem if multiple VMs are sharing the HCA, because the handles may have already been assigned to other VMs.

Further, InfiniBand port addresses (local ID or LID) are associated with each port, and only one LID can be associated with each port. The mapping between the LIDs and the physical ports is managed by external subnet management tools, making it difficult to make changes during migration. Also, since the LIDs can be used by other VMs sharing the same HCA, in many cases it is not feasible to change them during migration. In contrast, both IP and MAC addresses used in Ethernet are device-transparent and can be associated with any Ethernet device. Multiple MAC and IP addresses can also be associated with one device, which offers flexibility in migrating and sharing the network devices in the VM environment. For example, each Xen domain has its own IP and MAC address, which can be easily migrated with domains, and can be re-associated to the new hardware.

5.1.2 User-Level Communication

User level communication makes migration more difficult from at least two aspects:

First, besides kernel drivers, applications can also cache multiple opaque handles to reference the HCA resources. If those handles are changed after migration we cannot update those cached copies at the user-level. Also, RDMA needs some handles (remote memory keys) to be cached at remote peers, which makes the problem even more difficult. In contrast, applications for traditional networks generally use the sockets interface, where all complexities are hidden inside the kernel and can be changed transparently after migration.

Second, with direct access to the hardware device from the user-level it is difficult to suspend the communication during migration. For traditional networks with socket programming, the kernel intercepts every I/O request, making it much easier to suspend and resume the communication during migration.

5.1.3 Hardware Managed Connection State Information

To achieve high performance and RDMA, OS-bypass interconnects typically store connection state information in hardware. This information is also used to provide hardware-level reliable service, e.g., Reliable Connection (RC) service, which automatically performs packet ordering, re-transmission, etc. With hardware managed connection states, the operating system avoids the stack processing overhead and can devote more CPU resources to computation. This presents a problem for migration, however, since there is no easy way to migrate connection states between the network devices. Given this, the hardware cannot recover any dropped packets during migration. Meanwhile, any dropped or out-of-order packets may cause a fatal error to be returned to an application since there is no software recovery mechanism.

In contrast, migration is easier for a traditional TCP stack on Ethernet. The connection states are managed by the operating system. Thus, it is usually sufficient to save the main memory during migration and all connection states will be migrated with the virtual OS. For example, Xen does not make a special effort to recover any lost or out-of-order IP packet during migration; such IP level errors are recovered by the OS at the TCP layer.

5.2 Detailed Design Issues of Nomad

We now present some of the design choices made for Nomad to address the aforementioned challenges. We use namespace virtualization to virtualize the location-dependent resources and a handshake protocol to coordinate among VMs to make the connection state deterministic during migration.

We use Xen and InfiniBand throughout our discussion. However, most of the issues discussed are common to other VMMs and OS-bypass network devices.

5.2.1 Location-Dependent Resources

As we have discussed, software can only use opaque handles to refer to HCA resources. All details of connection-level states are managed by HCAs and cannot be directly accessed or modified. Thus, we need to free the location-dependent resources before the migration starts and re-allocate them after the migration. The major complexity comes from the location-dependent opaque handles. These handles are assigned by the firmware and can only be used to access local HCA resources. It means that they must be changed after the resources are re-allocated on the new host. However, they can be cached by user applications to access the HCA resources. How to approach these cached opaque handles is a challenging task. For example, a typical parallel application using InfiniBand caches the following opaque handles:

- LIDs (port addresses) - These are exchanged among the processes involved after the application starts to address the remote peers.
- Queue Pair numbers (QPN) after the QPs are created - To establish a reliable connection, each QP has to know both the LID and the QP number of the remote side.
- Local and remote memory keys after registration - The same keys must be used to reference the communication buffer for either local communication operations or remote access (RDMA).

All above handles may be changed upon migration. In order to maintain application transparency, we must ensure that the application can still use the re-allocated resources with the old handles.

To achieve application transparency, we introduce a virtualization layer between the opaque handles that applications may use and the real handles associated with HCA resources. To virtualize the LID, we assign a VM identification (VLID), which is unique within a cluster, to each VM once it is instantiated. The VLID is returned to the application as LID. Similarly, once a QP is created, a virtual QP number (VQPN) is returned to the application instead of the actual QPN.

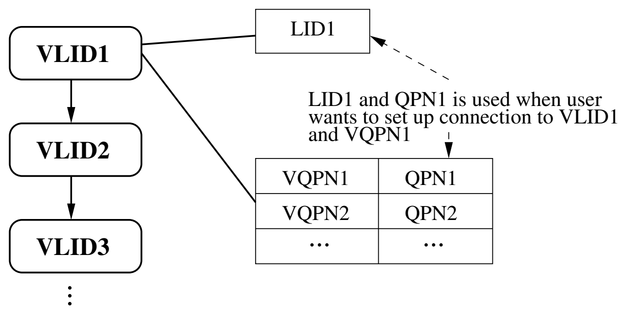

VLID1
VLID2
VLID3
LID1
LID1 and QPN1 is used when user wants to set up connection to VLID1 and VQPN1
VQPN1
QPN1
VQPN2
QPN2
...
...

Figure 5.2: Destination mapping table

In order to determine the real LID and QPN when the application tries to set up a connection, Nomad maintains a *destination mapping table* similar to Figure 5.2 at the front-end driver. When an application tries to set up a connection to a remote peer represented by VLID and VQPN, the front-end driver intercepts the connection request and replaces the VLID and VQPN according to the content in the mapping table. The driver may not be able to locate the entry in the table, which happens the first time the connection is established. In that case it will send a lookup request to the front-end driver on the VM denoted by VLID to “pull” the real LID and QPN that should be used and create an entry in the mapping table.

Once a connection is set up between two processes on different VMs we consider those two VMs connected. When a VM is migrated, the same VQPN and VLID then may correspond to a different QP number and LID. Nomad must make sure that any changes are reflected in the *destination mapping table* on each of the connected VMs. To achieve that, each VM maintains a *registered list* at the front-end driver to keep track of the connected VMs. Once a VM receives a lookup request, it puts the remote VM into the *registered list.* After migration, updates of new handles will be sent to all the connected VMs in the *registered list* to "push" the updates, which will be reflected in their mapping table. The connections can then be re-established automatically between the VMs without notifying the application, using the latest handles.

Once an application completes, the driver will determine all remote VMs to which it no longer has connections. It then sends an "unregister" request to those VMs to remove itself from their *registered list.* In this way, we avoid unnecessary updates being sent among VMs.

5.2.2 User-Level Communication

With namespace virtualization, the applications can use the same handle to access the HCA resources after migration. Even with this, we still need to suspend the network activity during the migration. Unlike the traditional TCP/IP stack where all communication goes through the kernel, the user-level communication leaves no central control point from where we can suspend the communication. Fortunately, practically no application accesses the hardware directly. The user-level communication is always carried out from a user communication library, which is maintained synchronously with the kernel driver. This allows us to intercept all send operations in this communication library. Taking InfiniBand as an example, we generate an event to the communication library to mark the QP suspended if we want to suspend the communication on that specific QP. If the application attempts to send a message to a suspended QP, we buffer the descriptors in the QP buffer, but do not ring the doorbell so that the requests are not issued to HCA. When resuming the communication, we update every buffered descriptor with the latest memory keys and ring the doorbell; the descriptors then will be processed on the new HCA. This delays communication without compromising application transparency. Note that this scheme does not require extra resources to buffer the descriptors, because the QP buffers are already allocated.

5.2.3 Connection State Information

Since there is no easy way to manage the connection state information stored on the HCA, we work around this problem by bringing the connection (QP) states to a deterministic state. When the VM starts migrating, we not only mark all QPs as suspended, but also wait for all the outstanding send operations to finish during the suspension of communication. In this way, there will be no in-flight packets originating from the migrating VM.

We must also avoid packets being sent to the migrating VM. Nomad achieves this by sending a suspend request to all the connected VMs. Upon receiving a suspend request, the connected VM will notify the user communication library to mark the corresponding QP as suspended and wait for all outstanding send operations on that

QP to finish. Note that communication on QPs to other VMs will not be affected. The *registered list* can be used to identify all the connected VMs.

After all communication on all the migrating and the connected VMs is suspended and all the outstanding sends finish, the connection states are deterministic and thus need not be migrated. We simply need to resume the communication after the migration is done.

5.2.4 Unreliable Datagram (UD) Services

Besides Reliable Connection (RC) service, most modern interconnects also provide Unreliable Datagram (UD) service. UD service is easier to manage since we do not need to suspend the remote communication, and lost packets during migration will be recovered by the application itself. Only the UD address handles need to be updated after migration; for InfiniBand these are the LID and QP number.

Updating the UD address can be done in a similar way as for RC services. For example, InfiniBand requires a UD address handle to be created before any UD communication takes place. Nomad checks with the destination VM, denoted by VLID, for updates when creating the address handle. It is then registered with that VM. When a VM is migrated, it will update all VMs in the *registered list* with the new QP numbers and LIDs so the address handles will be re-created.

5.3 Nomad Architecture for XenIB

Figure 5.3 illustrates the overall architecture of Nomad, which consists of the following major components:

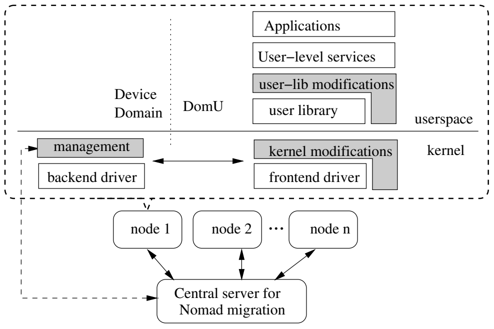


Figure 5.3: Architecture of Nomad

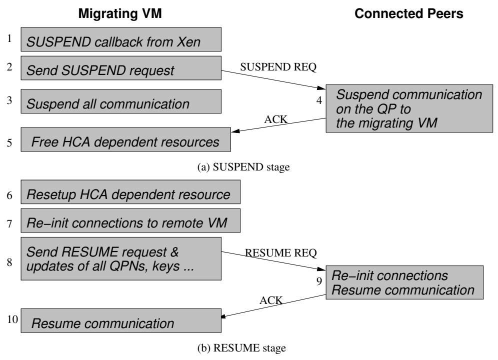

Migrating VM
Connected Peers
1 SUSPEND callback from Xen
2 Send SUSPEND request
SUSPEND REQ
3 Suspend all communication
4 Suspend communication on the QP to the migrating VM
ACK
5 Free HCA dependent resources
(a) SUSPEND stage
6 Resetup HCA dependent resource
7 Re–init connections to remote VM
8 Send RESUME request & updates of all QPNs, keys ...
RESUME REQ
9 Re–init connections Resume communication
ACK
10 Resume communication
(b) RESUME stage

Figure 5.4: Protocol for migrating one VM

- Modified user communication library: major modifications compared with the driver stack shown in Figure 2.4 include code to suspend/resume communication on QPs, and a lookup list for memory keys. Note that all changes are transparent to the higher level InfiniBand services and applications.

- Modified InfiniBand driver in kernels of guest operating system: Major code changes include the suspend/resume callback interfaces interacting with XenBus interfaces[73]; the interaction with the user library notifying it to suspend/resume communication as necessary; the destination mapping tables as described in Figure 5.2; re-allocation of opaque handles after migration; and memory key mapping tables for all kernel communication.

- Management network: This includes a central server and management module plug-ins at the privileged domain. All control messages (i.e. suspend or resume requests) are forwarded by a management module in the privileged domain. The central server keeps track of the physical host of each virtual machine so that control messages addressed by VID can be sent to the correct management module. Though this forwarding is not absolutely necessary, this design has its advantages. First, we can verify the correctness/validity of the control messages, so a malicious guest domain (which may not belong to the same parallel job) will not break the migration protocols. Further, the privileged domain will not be migrated, so the management framework itself can be built on high speed interconnects like InfiniBand. If the management network involves the VMs that could be migrated, using InfiniBand may cause additional complexity. Note that the central server does not necessarily affect system scalability on large node clusters. It is only accessed to resolve the actual location of VMs. All communication steps do not go through this central server.

Figure 5.4 illustrates the protocol of Nomad to migrate a VM. We use a two stage protocol, following the model of migrating para-virtualized devices in Xen. The front-end drivers go into a *suspend* stage after receiving a suspend callback from the hypervisor to get ready for migration. It goes into a *resume* stage after receiving the resume callback to restart communication on the new host. At the *suspend* stage, the driver sends suspend requests to the connected VMs to suspend their communication. Local communication is then suspended in parallel. Once acknowledgments have been received from all connected VMs, location dependent resources are freed and the suspend stage is finished. In the resume stage, the driver will first re-allocate all location dependent resources and then send update messages to all VMs in the *registered list*. Upon receiving the update, connected VMs can re-establish the connection and resume the communication. After all connected VMs have acknowledged, the communication on the migrating VM will be resumed.

In some cases users need to migrate a group of virtual machines to new hosts. In this case, because of the existence of the central server as a coordinator, we simplify the control message exchange among the migrating VMs. We assume that the central server will know the set of VMs that the user wants to migrate simultaneously. Then during the suspend stage, each migrating VM will query the server to get the list of connected VMs which are also migrating. Suspension requests are not sent to those VMs, because they will suspend their communication regardless. Instead, all migrating VMs will send the suspend acknowledgments directly to each other. During the resume stage, however, extra steps are needed to exchange the updated resource handles among the migrating VMs before the connections between the QPs can be re-established (before step 7 in Figure 5.4) with the correct resource handles.

5.4 Evaluation of Nomad

In this section, we evaluate the performance of our prototype implementation of Nomad. We first evaluate the impact of VM migration on InfiniBand verbs layer micro-benchmarks, then we move to application-level HPC benchmarks. We focus on HPC benchmarks since they are typically more sensitive to the network communication performance and allow us to evaluate the performance of Nomad better. Since there are few HPC benchmarks directly written with InfiniBand verbs, we use benchmarks on top of MVAPICH for this evaluation.

5.4.1 Experimental Setup

The experiments are carried out on an InfiniBand cluster. Each system in the cluster is equipped with dual Intel Xeon 2.66 GHz CPUs, 2 GB memory and a Mellanox MT23108 PCI-X InfiniBand HCA. The systems are connected with an InfiniScale

InfiniBand switch. Besides InfiniBand, the cluster is also connected with Gigabit Ethernet as the control network. Xen-3.0 with the 2.6.16 kernel is used on all computing nodes. Domain 0 (the device domain) is configured to use 512 MB and each guest domain runs with a single virtual CPU and 256 MB memory. Because Xen migration transfers memory pages over TCP/IP networks, which requires heavy CPU resources, we host one VM on each physical node. In this way, there is one spare CPU to handle the memory page transfer, which separates the overhead of Nomad from other migration costs. This allows us to better evaluate our implementation.

5.4.2 Micro-benchmark Evaluation

In this subsection, we evaluate the impact of Nomad on micro-benchmarks. We use *Perftest* benchmarks provided with the OpenFabrics stack. They consist of a set of InfiniBand verb layer benchmarks to evaluate the basic communication performance between two processes. We ran the tests on two VMs, with each of them hosting one process. We measure the performance reported by the benchmarks while migrating the VMs, one at a time.

We first measure performance numbers with all the *Perftest* benchmarks without migration. We observe no noticeable overhead caused by Nomad as compared to our original VMM-bypass I/O in [28], on either latency or bandwidth. This means the overhead of the namespace virtualization is negligibly small.

Next we measure the migration downtime using the RDMA latency test. It is a ping-pong test that a process RDMA writes to the peer and the peer acknowledges with another RDMA write. This process repeats one thousand times and the worst and median half round-trip time are reported. We modify the benchmark to keep measuring the latency in loops. Figure 5.5 shows the RDMA latency reported in each iteration. The worst latency is always higher than the typical latency due to process skews at the first few ping-pongs. We also observe that during iterations that we migrate the VMs, the worst latency increases to around 90 ms from under few hundred micro-seconds. This approximates the migration cost when migrating simple programs.

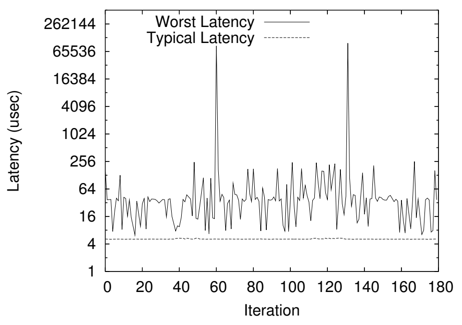


Figure 5.5: Impact of migration on RDMA latency

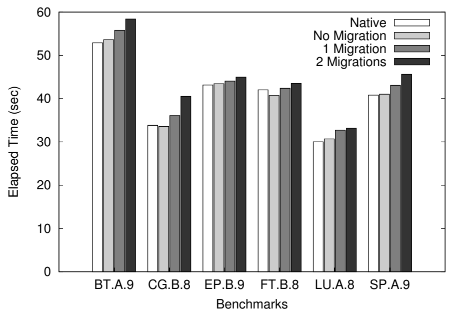


Figure 5.6: Impact of migration on NAS benchmarks (migrating one VM out of eight)

5.4.3 HPC Benchmarks

In this subsection we examine the impact of migration on HPC benchmarks. We use the NAS Parallel Benchmarks (NPB) [39] for evaluation, which are a set of computing kernels widely used by various classes of scientific applications. Also, the benchmarks have different communication patterns, from the ones hardly communicating (EP) to communication intensive ones like CG and FT. This allows us to better evaluate the overhead of Nomad.

We run the benchmarks on 8 processes with each VM hosting one computing process. We then migrate VMs one at a time during the process to see the impact of migration. Figure 5.6 compares the performance between running NAS on native systems, with Nomad but no migration, migrating a VM once, and migrating a VM twice. As we can see from the graph, Nomad causes only slight overhead if there is no migration, which conforms to our earlier evaluation on XenIB [65]. Each migration causes 0.5 to 3 seconds increase of total execution time, depending on the benchmark. This overhead will be marginal for longer running applications.

We now take a closer look at the migration cost caused by Nomad. As we have discussed, the migration process can be divided into suspend and resume stages. We analyze the cost of both of these stages.

The overhead of the suspend stage can be broken down into two parts, time to wait for local and remote peers to suspend communication and the time to free local resources. Suspending local communication occurs in parallel with suspending remote communication. Suspension of remote communication typically takes a relatively larger amount of time since there is extra overhead to synchronize through the management network. To estimate the overhead of synchronization, we also measure the cost of suspending communication on the remote peers (*remote suspension time*). Please note that the results are based on multiple runs.

We profile each of these stages, as in Figure 5.7. We observe that the remote suspension times vary largely depending on the communication patterns. Table 5.1 characterizes the communication patterns observed on the MPI process hosted in the migrating VM. As we can see, CG has the longest *remote suspension time*, because it communicates frequently with relatively large messages, thus likely takes a long

Table 5.1: NAS Communication patterns: number of total messages to the most frequently communicated peer and the average message size to that peer

	Num. of Msg.	Avg. Size (KBytes)
BT	615	100.4
CG	6006	49.4
FT	48	3844.8
EP	5	0.024
LU	15763	3.8
SP	1214	74.9

time waiting for the outstanding communications. Following CG are LU and FT, which have a large number of small messages and a small number of large messages, respectively. EP has extremely low communication volume, and its remote suspension time is almost unobservable in the figure. With additional synchronization time, the migrating VM takes typically a few to tens of milliseconds waiting for communication suspension.

Another major cost observed is the time to free local resources, which takes 22 to 57 ms based on the resources allocated. Because we fix the number of peers in the job, we see a strong correlation between the time to free the resources and the amount of memory registered. For instance, for NAS-BT, the VM has registered 7,540 pages of memory by the time it is migrated, and it takes 57 ms to free the local resources. For NAS-EP, where only 2,138 pages are registered by the time the VM is migrated, it takes only around 22 ms to free all the resources. This suggests that a scheme which delays the freeing of resources will potentially reduce the migration cost further: the VM can be suspended without freeing HCA resources; and the privileged domain can track the resources used by the VM and free them after VM migration.

The cost at the resume stage mainly includes the time to re-allocate the HCA resources and the time to resume the communication. As with our analysis of the suspend stage, we also profile the time taken on the remote peers to resume the communication. The time is measured from the resume request arrival to the sending of the acknowledgment; this time includes updating the resource handle lookup list, re-establishing the connections, and reposting the unposted descriptors during the migration period. The time to resume local communication on the migrating VM has very low overhead because there are no unposted descriptors.

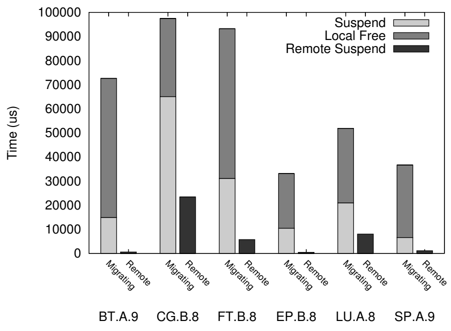


Figure 5.7: Suspend time running NAS benchmarks

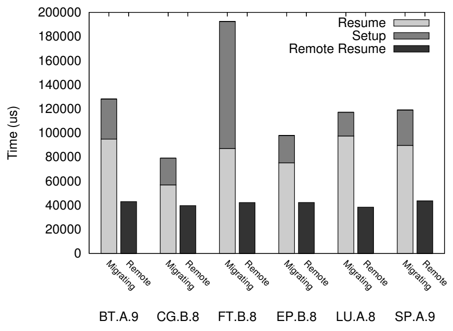


Figure 5.8: Resume time running NAS benchmarks

As shown in Figure 5.8, re-allocating the HCA resources is still a major cost of the resume period. We see the same correlation between the amount of registered memory and the time to re-allocate resources. The time varies from around 105ms for NAS-FT to 22ms for NAS-EP in our studies. This suggests pre-registration of memory pages can help reduce the migration cost too.

Our evaluation shows slightly more time to resume than to suspend the communication on remote peers. This difference is largely due to the process to update the lookup list and to re-establish the connections. Also, the total time spent in the suspend and resume stages is smaller than the overhead we observed in Figure 5.6. We believe that the extra overhead is the cost of live migration, e.g., the migrating OS is running in a shadow mode, with extra cost to dirty a page.

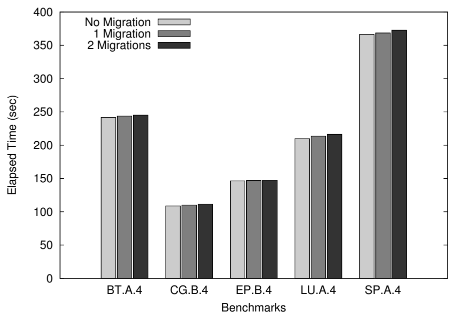
5.4.4 Migrating Multiple VMs

We also measure the overhead of migrating multiple VMs simultaneously while running the applications. We run the NAS benchmarks on 4 processes located on 4 different VMs. During the execution we migrate all 4 VMs simultaneously to distinct nodes.

Figure 5.9 shows the comparison of the total execution time. We observe from the graph that the average per checkpoint cost is not increased much as compared to the case of migrating one VM. Since the applications have longer execution time with four processes, the impact of migration looks much smaller. Despite this, we observe larger variation of the results we collected. We believe that the skew between processes is the major cause for the variation. The skew can be mainly due to two reasons:

- Though we start migration of all 4 VMs at the same time, Xen may take a varied amount of time to pre-copy the memory pages, thus the time each process enters the Nomad suspend stage is different.

- Each VM may not take the same amount of time to suspend the local communication and to free the local resources. Similarly, the time to re-allocate the resources on the new host and resume communication can also be different.



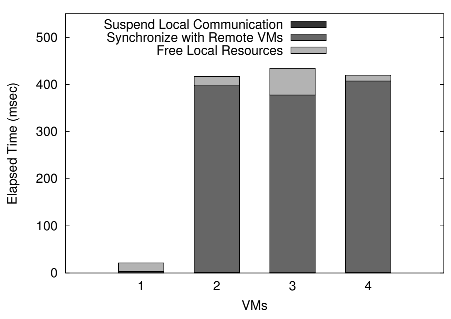


Figure 5.9: Impact of migration on NAS benchmarks (migrating all four VMs)

Figure 5.10: Suspend time running NAS-CG

Figure 5.10 shows a breakdown of suspend time spent on each of the migrating VM. As we can see, VM1 takes a significantly shorter time to wait for remote communication suspension than other three VMs. It clearly indicates that VM1 enters the suspend stage later than other three: all the other three VMs have already suspended their traffic and are waiting to synchronize with VM1.

CHAPTER 6

EFFICIENT VIRTUAL MACHINE AWARE COMMUNICATION LIBRARIES

In this chapter we propose efficient VM-aware communication libraries. We design IVC, an Inter-VM Communication library that allows efficient shared memory communication between VMs located on the same physical hosts. We also design MPI library, which benefits unmodified MPI applications from our work. Finally, we evaluate the designs in Section 6.3. More specifically, this chapter focuses on the highlighted regions in our research framework shown as in Figure 6.1. We will discuss further enhancement to inter-VM communication with a one-copy shared memory mechanism in the next chapter.

6.1 IVC: Efficient Inter-VM Communication through Shared Memory

In this section we present our inter-VM communication (IVC) library design, which provides efficient shared memory communication between VMs on the same physical host. It offers the flexibility to host computing processes on separate VMs for fine-grained scheduling without sacrificing the efficiency of shared memory communication.

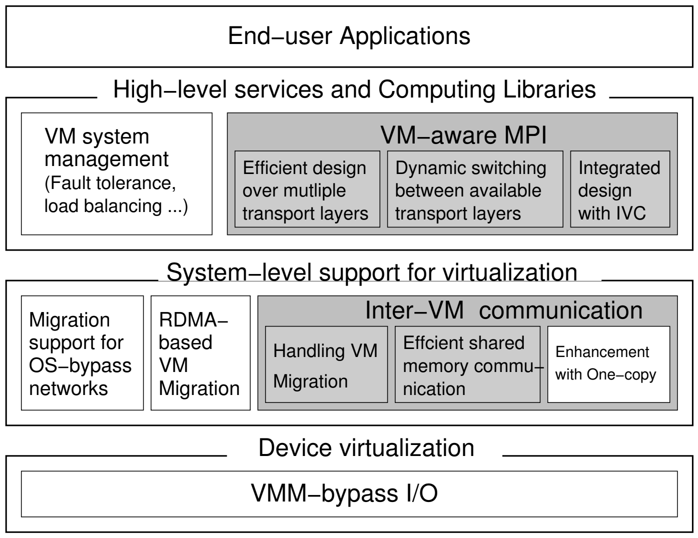

End−user Applications
High−level services and Computing Libraries
VM system management (Fault tolerance, load balancing ...)
VM−aware MPI
Efficient design over mutliple transport layers
Dynamic switching between available transport layers
Integrated design with IVC
System−level support for virtualization
Migration support for OS−bypass networks
RDMA−based VM Migration
Inter−VM communication
Handling VM Migration
Effcient shared memory commu−nication
Enhancement with One−copy
Device virtualization
VMM−bypass I/O

Figure 6.1: VM-aware communication libraries in the proposed research framework

Our design is based on page-sharing mechanisms through grant tables provided by Xen, as described in Section 2.1. There are still many challenges that need to be addressed, however. First, the initial purpose of a grant-table mechanism is for sharing data between device drivers in kernel space. Thus, we need to carefully design protocols to set up shared memory regions for user space processes and provide communication services through these shared regions. Second, we need to design an easy-to-use API to allow applications to use IVC. Further, IVC only allows communication between VMs on the same host, which leads to additional concerns for VM migration. We address these challenges in the following.

6.1.1 Setting up Shared Memory Regions

In this section we describe the key component of IVC: How to set up shared memory regions between two user processes when they are not hosted in the same OS?

IVC sets up shared memory regions through Xen *grant table* mechanisms. Figure 6.2 provides a high level architectural overview of IVC and illustrates how IVC sets up shared memory regions between two VMs. IVC consists of two parts: a user space communication library and a kernel driver. We use a client-server model to set up an IVC connection between two computing processes. One process (the client) calls the IVC user library to initiate the connection setup (step 1 in Figure 6.2). Internally, the IVC user library allocates a communication buffer, which consists of several memory pages and will be used later as the shared region. The user library then calls the kernel driver to grant the remote VM the right to access those pages and returns the grant table reference handles from the Xen hypervisor (steps 2 and 3). The reference handles are sent to the IVC library on the destination VM (step 4). The IVC library on the destination VM then maps the communication buffer to its own address space through the kernel driver (steps 5 and 6). Finally, IVC notifies both computing processes that the IVC connection setup is finished, as in step 7 of Figure 6.2.

6.1.2 Communication through Shared Memory Regions

After creating a shared buffer between two IVC libraries, we design primitives for shared memory communication between computing processes. IVC provides a socket

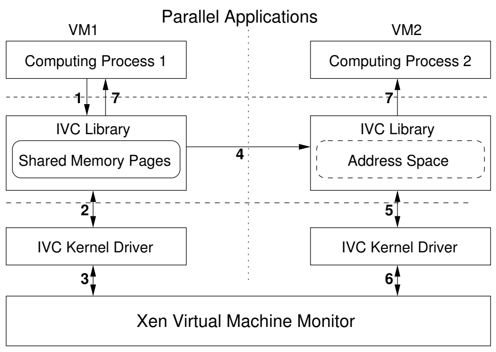


Figure 6.2: Mapping shared memory pages

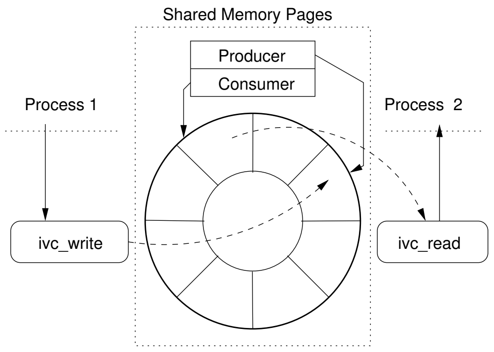


Figure 6.3: Communication through mapped shared memory regions

style read/write interface and conducts shared memory communication through classic producer-consumer algorithms. As shown in Figure 6.3, the buffer is divided into send/receive rings containing multiple data segments and a pair of producer/consumer pointers. A call to `ivc_write` places the data on the producer segments and advances the producer pointer. And a call to `ivc_read` reads the data from the consumer segment and advances the consumer pointer. Both the sender and receiver check the producer and consumer pointers to determine if the buffer is full or for data arrival. Note that Figure 6.3 only shows the send ring for process 1 (receive ring for process 2). The total buffer size and the data segment size are tunable. In our implementation, each send/receive ring is 64KB and each data segment is 64 bytes. Thus, an `ivc_write` call can consume multiple segments. If there is not enough free data segments, `ivc_write` will simply return the number of bytes successfully sent, and the computing process is responsible to retry later. Similarly, `ivc_read` is also non-blocking and returns the number of bytes received.

```
/* Register with IVC */
ctx = ivc_register(MAGIC_ID, CLIENT_ID);
/* Get all Peers on the same physical node */
peers = ivc_get_hosts(ctx);
/* We assume only one peer now (the server).
 * Connect to server (peers[0]).
 * channel.id will be SERVER_ID upon success. */
channel = ivc_connect(ctx, peers[0]);
/* Send data to server, bytes will be @size upon
 * success. Otherwise, we will have to retry. */
bytes = ivc_write(channel, buffer, size);
/* Close connection */
ivc_close(channel);
```

(a) Communication client

```
/* Register with IVC */
ctx = ivc_register(MAGIC_ID, SERVER_ID);
/* Block until an incomming connection.
 * Upon success, channel.id will be CLIENT_ID */
channel = ivc_accept(ctx);
/* Receive data from client. */
 * Upon success, bytes will be equal to @size.
 * We will have to retry other wise. */
bytes = ivc_read(channel, buffer, size);
/* Close connection */
ivc_close(channel);
```

(b) Communication server

Figure 6.4: A very brief client-server example for using IVC

Initialization of IVC also requires careful design. IVC sets up connections for shared memory communication between VMs only after receiving a connection request from each of the hosted processes. Computing processes, however, cannot be expected to know which peers share the same physical host. Such information is hidden to the VMs and thus has to be provided by IVC. To address this issue, an IVC backend driver is run in the privileged domain (Dom0). Each parallel job that intends to use IVC should have a unique *magic id* across the cluster. When a computing process initializes, it notifies the IVC library of the *magic id* for the parallel job through an `ivc_init` call. This process is then registered to the backend driver. All registered processes with the same *magic id* form a local communication group, in which processes can communicate through IVC. A computing process can obtain all peers in the local communication group through `ivc_get_hosts`. It can then connect to those peers through multiple `ivc_connect` calls, which initialize the IVC connection as mentioned above. Using a *magic id* allows multiple communication groups for different parallel jobs to co-exist within one physical node. Assignment of this unique *magic id* across the cluster could be provided by batch schedulers such as PBS [47], or other process manager such as MPD [2] or SLURM [54].

Figure 6.4 shows a brief example of how two processes communicate through IVC. As we can see, IVC provides socket-style interfaces, and communication establishment follows a client-server model.

6.1.3 Virtual Machine Migration

As a VM-aware communication library, an additional challenge faced by IVC is handling VM migration. As a shared memory library, IVC can only be used to communicate between processes on the same physical host. Subsequently, once a VM migrates to another physical host, processes running on the migrating VM can no longer communicate with the same peers through IVC. They may instead be able to communicate through IVC with a new set of peers on the new physical host after migration.

IVC provides callback interfaces to assist applications in handling VM migration. Figure 6.5 illustrates the main flow of IVC in case of VM migration. First, the IVC kernel driver gets a callback from the Xen hypervisor once the VM is about to migrate (step 1). It notifies the IVC user library to stop writing data into the send ring to prevent data loss during migration; this is achieved by returning 0 on every attempt of `ivc_write`. After that, the IVC kernel notifies all other VMs on the same host through event channels that the VM is migrating. Correspondingly, the IVC libraries running in other VMs will stop their send operations and acknowledge the migration event (steps 2 and 3). IVC then gives the user programs a callback, indicating that the IVC channel is no longer available due to migration. Meanwhile, there may be data remaining in the receive ring that has not been passed to the application. Thus, IVC also provides applications a data buffer containing all data in the receive ring that has not been read by the application through `ivc_read`(step 4). Finally, IVC unmaps the shared memory pages, frees local resources, and notifies the hypervisor that the device has been successfully suspended and can be migrated safely (step 5). After the VM is migrated to the remote host, the application will receive another callback indicating arrival on a new host, allowing connections to be set up to a new set of peers through similar steps as in Figure 6.4(a) (step 6).

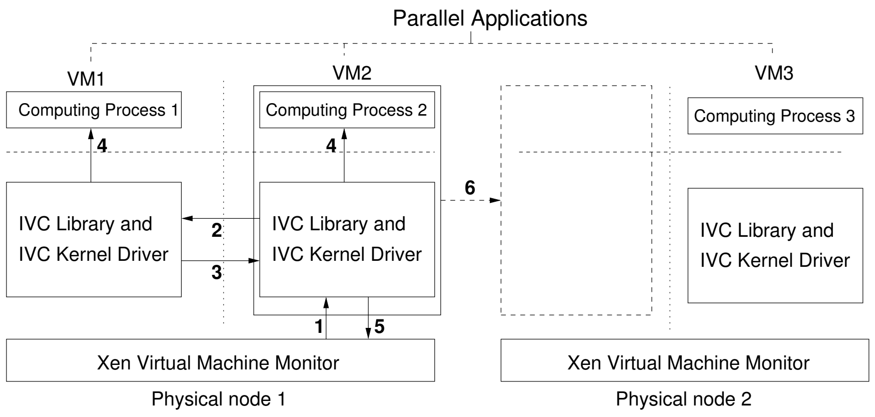

Parallel Applications
VM1
VM2
VM3
Computing Process 1
Computing Process 2
Computing Process 3
4
4
6
IVC Library and IVC Kernel Driver
2
3
IVC Library and IVC Kernel Driver
IVC Library and IVC Kernel Driver
1
5
Xen Virtual Machine Monitor
Xen Virtual Machine Monitor
Physical node 1
Physical node 2

Figure 6.5: Migrating one VM (VM2) of a three-process parallel job hosted on three VMs

6.2 Virtual Machine Aware MPI-2 Library

In this section we present VM-aware MPI, which allows user applications benefit from the aforementioned I/O virtualization techniques. We introduce our design in two steps. First in Section 6.2.1 we design MVAPICH2, a high performance

MPI-2 library over InfiniBand. And based on that, we present MVAPICH2-ivc in Section 6.2.2, which is VM-aware and is able to transparently take advantage of inter-VM communication and VMM-bypass I/O.

6.2.1 MVAPICH2: High Performance MPI-2 Library over InfiniBand

In this section we present the design of MVAPICH2. We are trying to achieve two main objectives through this design:

- Modern clusters provide multiple low-level methods for communication, such as intra-node communication through shared memory, and different programming libraries for various interconnects etc. Our new design should be able to take advantage of the available communication methods on a cluster to achieve the best performance.
- For portability reasons, it is desirable to have a concise device abstraction for each of these communication methods. And this abstraction should be well designed so that we can have enough information to perform most of the hardware-specific optimizations and enable the MPI library to achieve maximum performance and scalability.

As illustrated in Figure 6.6, we follow the basic idea of the layering structure from MPICH2. We start from an extension of the ADI3 layer in MPICH2. And below that we have our own design of the multi-communication method (MCM) layer and the device layer. The device layer provides abstractions of the communication methods on the cluster, which can be either an intra-node communication device or an inter-node communication device. And the MCM layer aims to exploit the performance benefits provided by these abstract devices.

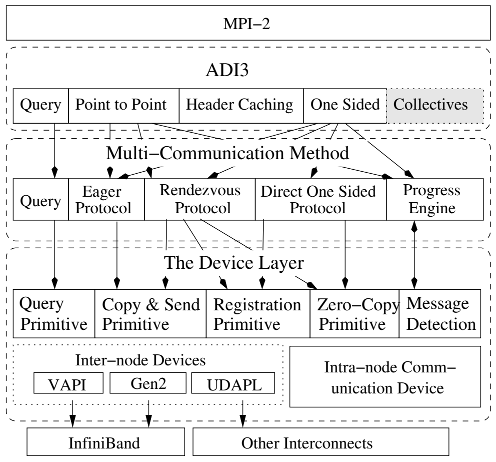


Figure 6.6: Overall design of MVAPICH2

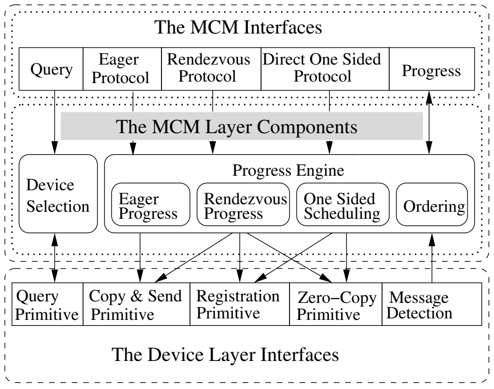


Figure 6.7: Design details of the MCM layer

The ADI3 layer, which extends from the original ADI3 layer in MPICH2, is the highest level in our design. We inherit the other MPI functionality from MPICH2 implementation. The main responsibilities of our layer include selecting an appropriate internal communication protocol, eager or rendezvous, for each point-to-point or one-sided operation from the user application. Our extensions for the ADI3 layer are mainly for high level optimizations along the following fronts:

- Query: Instead of choosing the rendezvous or eager protocols solely based on the message size, as most of the current MPI implementations do, the ADI3 layer makes decisions based on the preference of the MCM layer. Because the MCM layer dynamically chooses from the available communication devices to send out the message and the optimal point to switch from the eager to rendezvous protocol might be different for each device, this query process helps the ADI3 layer to select the most efficient communication protocol for a specific message.

- Point-to-Point operations: Point to point communication can take advantage of header caching. The header caching feature caches the content of internal MPI headers at the receiver side. As a result, if the next message to the receiver contains the same cached fields in header, we reduce the message size, thus reducing the small message communication latency. As discussed in more detail in [63], header caching can only be put into this layer since only the ADI3 layer is able to understand the content of the message.

- One Sided Operations: In previous work we have extended the ADI3 layer to directly implement one sided operations based on InfiniBand RDMA operations to achieve higher performance and less CPU utilization [66]. This piece of work is also incorporated into the added functionality of the ADI3 layer based on the direct one sided interface provided by the MCM layer.

The multi-communication method (MCM) layer implements the communication protocols selected by the ADI3 layer using the communication primitives supported by the device layer. It understands the performance features of each communication device provided at the device layer and chooses the most suitable one to complete the communication.

The device-selection component collects the performance characteristics from the device layer through the device-query interfaces. It knows the message size and the destination from the ADI3, so it is able to decide the most suitable device to complete this message and the preferred communication protocol. The information is passed back to the ADI3 layer through the query interface.

Once the ADI3 layer decides the communication protocol, the actual communication is taken care of by the progress engine at the MCM layer. There are several important components of the progress engine. The eager and rendezvous progress components implement the eager and rendezvous communication protocols. The communication requests will be processed through the communication device chosen by the device-selection components. Typically for eager protocols, the messages will be sent through the copy-and-send primitives provided by the device layer. And for rendezvous protocol, the user buffers involved in the communication are first sent to the device through the registration primitives for registration and then the data is sent through the zero-copy primitives. In our design the progress engines are supposed to mask the recoverable failures of the device level. For example, the ADI3 layer may send an eager message too large for the device to send out at one time, so that the message needs to be broken down into smaller sizes and sent out in multiple packets. Also the device may fail to register the user buffer for zero-copy rendezvous protocols, then large messages will also need to be broken down into pieces to be sent through copy-and-send primitives.

The direct one sided progress component implements the direct one sided communication support for one sided operations. The detailed design and implementation details are described in [66, 64].

The ordering component helps to keep the correct ordering of the messages sent from different devices at the receiver side. The incoming message from a specific device is detected through the message detection interface provided by the device layer.

The lowest device layer implements the basic network transfer primitives. The paradigm of this layer allows us to hide the difference between various communication schemes provided by the system while exposing the maximum number of features (such as zero-copy communication) to the MCM layer. The interfaces provided by the device layer include copy-and-send primitives, zero-copy primitives, and the registration primitives which prepare for zero-copy communication.

Since there can be multiple devices existing at the same time, each device also implements the query primitive to provide the device-selection component of the MCM layer with the basic performance features. This enables the MCM layer to select at runtime the most suitable device for a particular message.

The message-detection primitive is queried by the progress engine at the MCM layer to detect the next incoming packet. The communication primitives at this layer are directly implemented based on the programming libraries provided by the communication systems.

6.2.2 MVAPICH2-ivc: Virtual Machine Aware MPI-2 Library

Now we present MVAPICH2-ivc, a VM-aware MPI library modified from MVAPICH2. MVAPICH2-ivc is able to communicate through IVC with peers on the same physical host and over InfiniBand when communication is inter-node. MVAPICH2-ivc is also able to intelligently switch between IVC and network communication as VMs migrate. By using MVAPICH2-ivc in VM environments, MPI applications can benefit from IVC without modification.

Design Overview

MVAPICH2-ivc is modified from MVAPICH2. A simplified architecture for MVAPICH2 is shown Figure 6.8. There are two device level communication channels available in MVAPICH2: a shared memory communication channel for peers hosted in the same operating system and a network channel over InfiniBand user verbs for other peers.

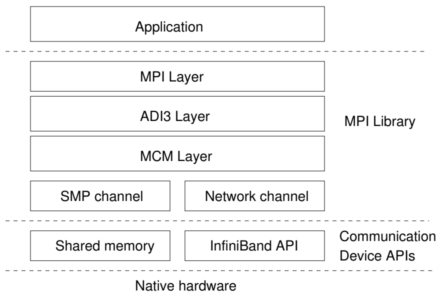


Figure 6.8: MVAPICH2 running in native environment

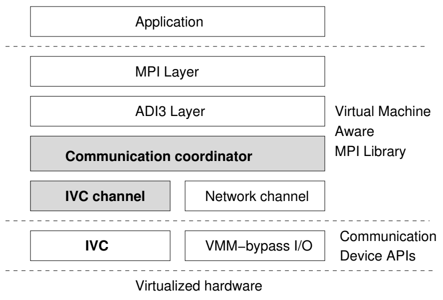


Figure 6.9: MVAPICH2-ivc running in VM environment

An unmodified MVAPICH2 can also run in VM environments, however, its default shared memory communication channel can no longer be used if computing processes are hosted on different VMs. By using IVC, MVAPICH2-ivc is able to communicate via shared memory between processes on the same physical node, regardless of whether they are in the same VM or not. Figure 6.9 illustrates the overall design of MVAPICH2-ivc running in a VM environment. Compared with MVAPICH2 in a native environment, there are three important changes. First, we replace the original shared memory communication channel with an IVC channel, which performs shared memory communication through IVC primitives. Second, the network channel is running on top of VMM-bypass I/O, which provides InfiniBand service in VM environments (because VMM-bypass I/O provides the same InfiniBand verbs, no changes to MVAPICH2 are needed). Third, we design a communication coordinator, which extends the original MCM layer by the ability to dynamically create and and tear down IVC connections as the VM migrates.

The communication coordinator keeps track of all the peers to which IVC communication is available. To achieve this, it takes advantage of a data structure called a *Virtual Connection (VC)*. In MVAPICH2, there is a single VC between each pair of computing processes, which encapsulates details about the available communication methods between that specific pair of processes as well as other state information. As shown in Figure 6.10, the communication coordinator maintains an *IVC-active* list, which contains all VCs for peer processes on the same physical host. During the initialization stage, this *IVC-active* list is generated by the communication coordinator according to the peer list returned by `ivc_get_hosts`. VCs on the list can be removed or added to this list when the VM migrates.

Once the application issues an MPI send to a peer process, the data is sent through IVC if the VC to that specific peer is on the *IVC-active* list. As described in Section 6.1.2, IVC cannot guarantee all data will be sent (or received) by one `ivc_write` (or `ivc_read`) call. To ensure in-order delivery, we must maintain queues of all outstanding send and receive operations for each VC in the *IVC-active* list. Operations on these queues are retried when possible.

Virtual Machine Migration

As discussed in Section 6.1.3, IVC issues application callbacks upon migration. Correspondingly, applications are expected to stop communicating through IVC to

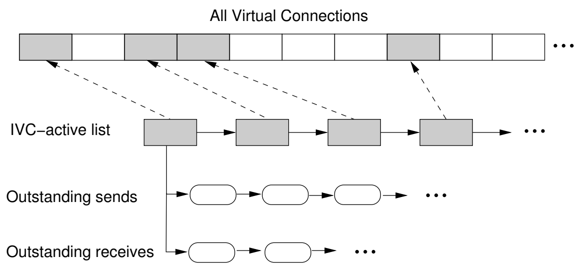


Figure 6.10: Organization of IVC-active list in MVAPICH2-ivc

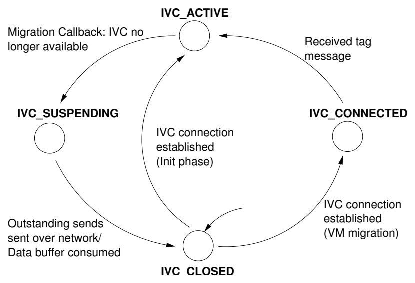

IVC_ACTIVE
Migration Callback: IVC no longer available
Received tag message
IVC_SUSPENDING
IVC connection established (Init phase)
IVC_CONNECTED
Outstanding sends sent over network/ Data buffer consumed
IVC connection established (VM migration)
IVC_CLOSED

Figure 6.11: State transition graph of *ivc state*

peers on the original physical host and can start IVC communication to peers on the new physical host. In MVAPICH2-ivc, the communication coordinator is responsible to adapt to such changes. The coordinator associates an *ivc state* to each VC. As shown in Figure 6.11, there are four states possible: IVC_CLOSED, IVC_ACTIVE, IVC_CONNECTED and IVC_SUSPENDING. At the initialization stage, each VC is either in the IVC_ACTIVE state if the IVC connection is set up, or in the IVC_CLOSED state, which indicates that IVC is not available and communication to that peer has to go through the network.

When a VM migrates, IVC communication will no longer be available to peers on the original host. As we have discussed, the communication coordinator will be notified, along with a data buffer containing the contents of the receive ring when an IVC connection is torn down. The coordinator then changes the state to IVC_SUSPENDING. In this state, all MPI-level send operations are temporarily blocked until the coordinator transmits all outstanding requests in the IVC outstanding send queue through the network channel. Also, all MPI-level receive operations are fulfilled from the data buffer received from IVC until all data in the buffer is consumed. Both of these steps are necessary to guarantee in-order delivery of MPI messages. Next, the coordinator changes the *ivc state* to IVC_CLOSED and removes the VC from the IVC active list. Communication between this pair of processes then flows through the network channel.

Once migrated, IVC will be available to peers on the new host. The coordinator will get a callback from IVC and set up IVC connections to eligible peers. IVC cannot be immediately used, however, since there may be pending messages on the network channel. To reach a consistent state, the communication coordinators on both sides of the VC change the *ivc state* to IVC_CONNECTED and send a flush message through the network channel. Once the coordinator receives a flush message, no more messages will arrive from the network channel. The *ivc state* is changed to IVC_ACTIVE and the VC is added to the IVC active list. Both sides can now communicate through the IVC channel.

6.3 Evaluation of Inter-VM Communication

In this section we present an integrated evaluation of a VM-based HPC environment running MVAPICH2-ivc with each process in a separate VM. We first evaluate the benefits achieved through IVC using a set of micro-benchmark and application-level benchmarks. We show that on multi-core systems MVAPICH2-ivc shows clear improvement compared with unmodified MVAPICH2, which cannot take advantage of shared memory communication when processes are on distinct VMs. We demonstrate that the performance of MVAPICH2-ivc is very close to that of MVAPICH2 in a native (non-virtualized) environment. Evaluation on up to 128 processes shows that a VM-based HPC environment can deliver very close application-level performance compared to a native environment.

6.3.1 Experimental Setup

The experiments were carried out on two testbeds. Testbed A consists of 64 computing nodes. There are 32 nodes with dual Opteron 254 (single core) processors and 4GB of RAM each and 32 nodes with dual Intel 3.6 GHz Xeon processors and 2GB RAM each. Testbed B consists of computing nodes with dual Intel Clovertown (quad-core) processors, for a total of 8 cores and 4GB of RAM. Both testbeds are connected through PCI-Express DDR InfiniBand HCAs (20 Gbps). Xen-3.0.4 with the 2.6.16.38 kernel is used on all computing nodes for VM-based environments. We launch the same number of VMs as the number of processors (cores) on each physical host, and host one computing process per VM.

We evaluate VM-based environments with MVAPICH2-ivc and unmodified MVAPICH2, each using VMM-bypass I/O. We also compare against the performance of MVAPICH2 in native environments. More specifically, our evaluation is conducted with the following three configurations:

- **IVC** - VM-based environment running MVAPICH2-ivc, which communicates through IVC if the processes are hosted in VMs on the same physical host.
- **No-IVC** - VM-based environment running unmodified MVAPICH2, which always communicates through the network since each process is in a separate VM.
- **Native** - Native environment running unmodified MVAPICH2, which uses shared memory communication between processes on the same node.

6.3.2 Micro-benchmark Evaluation

In this section, the performance of MVAPICH2-ivc is evaluated using a set of micro-benchmarks. First a comparison was made of the basic latency and bandwidth achieved by each of the three configurations, when the computing processes are on the same physical host. Next, the performance of the collective operations using the Intel MPI Benchmarks (IMB 3.0) [22] was measured.

Figure 6.12 illustrates the MPI-level latency reported for the OSU benchmarks [37]. For various message sizes, this test 'ping-pongs' messages between two processes for a number of iterations and reports the average one-way latency observed. IVC communication is able to achieve latency around 1.2μs for 4 byte messages, which, in the worst case, is only about 0.2μs higher than MVAPICH2 communicating through shared memory in a native environment. The IVC latency is slightly higher because MVAPICH2 has recently incorporated several optimizations for shared memory communication [8]. We plan to incorporate those optimizations in the future, as they should be applicable to IVC as well. In both cases, the latency is much lower than the no-IVC case, where communication via network loopback shows a 3.16μs latency for 4 byte messages.

Figure 6.13 presents MPI-level uni-directional bandwidth. In this test, a process sends a window of messages to its peer using non-blocking MPI sends and waits for an acknowledgment. The total message volume sent divided by the total time is reported as the bandwidth. Both IVC and native cases achieve much higher bandwidth for medium-sized messages than in the no-IVC case. An interesting observation is that IVC achieves higher bandwidth than the native case. This can be due to two reasons: first, MVAPICH2's optimized shared memory communication requires more complicated protocols for sending large messages, thus adding some overhead; second, IVC uses only 16 pages as the shared memory region, but MVAPICH2 uses a few million bytes. At a micro-benchmark level, a larger shared memory buffer can slightly hurt performance due to cache effects. A smaller shared memory region, however, holds less data, thus requiring the peer to receive data fast enough to maintain a high throughput. This can lead to less efficient communication progress for applications using large messages very frequently because processes are likely to be skewed. For IVC, we find 16 pages as a good choice for the shared memory buffer size [4].

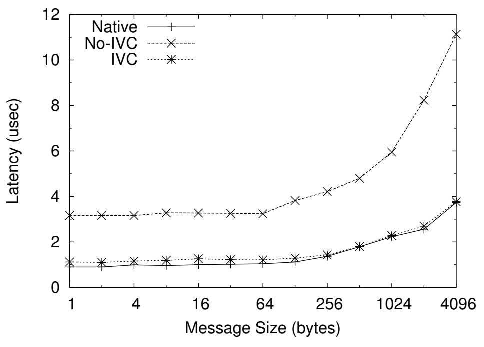


Figure 6.12: Latency

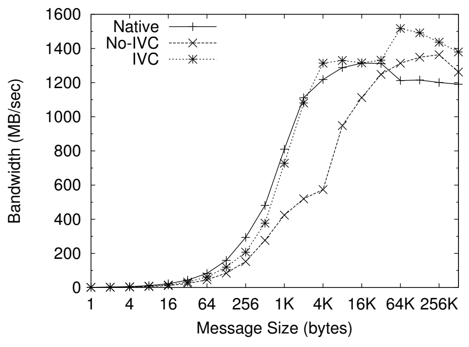


Figure 6.13: Bandwidth

Figures 6.14 and 6.15 illustrate the ability of MVAPICH2-ivc to automatically select the best available communication method after VM migration. In Figure 6.14, we kept running a latency test with 2KB messages. The test was first carried out between two VMs on distinct physical hosts. At around iteration 400, we migrated one VM so that both VMs were hosted on the same physical node. From the figure it

[4]We do not use larger number of pages because the current Xen-3.0.4 allows at most 1K pages to be shared per VM. While this restriction can be fixed in the future, 16 page shared buffers will allow us to support up to 60 VMs per physical node with the current implementation.

can be observed that the latency dropped from 9.2μs to 2.8μs because MVAPICH2-ivc started to use IVC for communication. At about iteration 1100, the VMs were again migrated to distinct physical hosts, which caused the latency to increase to the original 9.2μs. The drastic increase in latency during migration was because network communication freezes during VM migration, which is explained in more detail in [18]. In Figure 6.15, the same trend for a bandwidth test is observed, which reports bandwidth achieved for 2KB messages while the VM migrates. When two VMs are located on the same host (iteration 1100 to iteration 2000), communication through IVC increases the bandwidth from around 720MB/s to 1100MB/s.

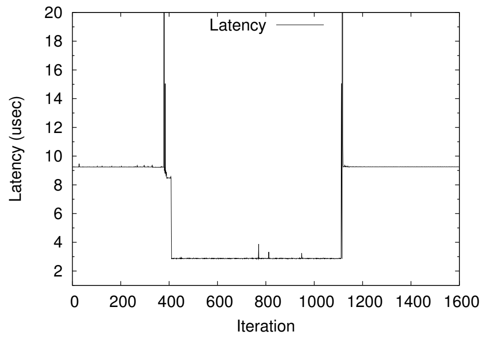


Figure 6.14: Migration during latency test

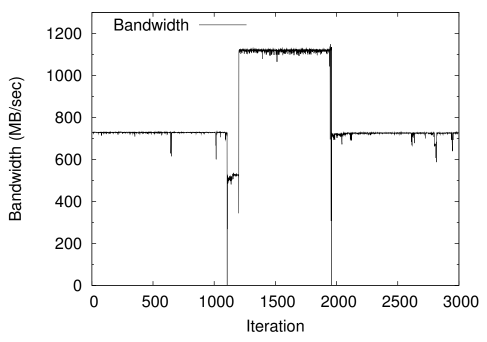
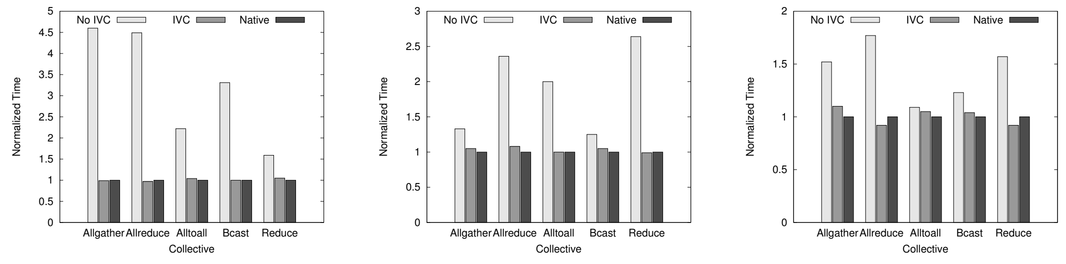


Figure 6.15: Migration during bandwidth test

Next, the performance of collective operations using the Intel MPI Benchmarks was evaluated. In Figure 6.16, several of the most commonly used collective operations are compared using all three configurations. The tests were conducted on Testbed B, using 2 nodes with 8 cores each (in total 16 processes). The collective performance is reported for small (16 bytes), medium (16KB), and large (256KB) size operations, with all results normalized to the Native configuration. Similar to the trends observed in the latency and bandwidth benchmarks, IVC significantly reduces the time needed for each collective operation as compared with the no-IVC case. It is able to deliver comparable performance as a native environment. Another important observation is that even though the no-IVC case achieves almost the same bandwidth for 256KB messages as IVC in Figure 6.13, it still performs significantly worse for some of the collective operations. This is due to the effect of network contention. On multi-core systems with 8 cores per node, the network performance can be largely degraded when 8 computing processes access the network simultaneously. Though the processes are also competing for memory bandwidth when communicating through IVC, memory bandwidth is typically much higher. This effect shows up in large message collectives, which further demonstrates the importance of shared memory communication for VM-based environments.



(a) 16 Bytes (b) 16K Bytes (c) 256K Bytes

Figure 6.16: Comparison of collective operations (2 nodes with 8 cores each)

6.3.3 Application-level Benchmark Evaluation

In this section, several application-level benchmarks are used to evaluate the performance of MVAPICH2-ivc. As in the micro-benchmarks evaluation, we evaluate configurations of IVC (MVAPICH2-ivc in VM environment), No-IVC (unmodified MVAPICH2 in a VM environment) and Native (MVAPICH2 in a native environment).

We use several applications in our evaluations. These applications have various communication patterns, allowing a thorough performance comparison of our three configurations. The applications used in the evaluation are:

- **NAS Parallel Benchmark Suite** - NAS [39] contains a set of benchmarks which are derived from the computing kernels common in Computational Fluid Dynamics (CFD) applications.

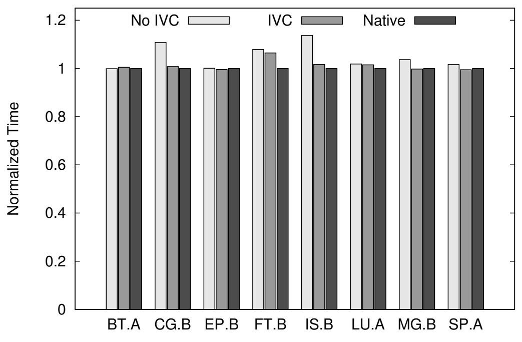
- **LAMMPS** - LAMMPS stands for Large-scale Atomic/
Molecular Massively Parallel Simulator [27]. It is a classical molecular dynamics simulator from Sandia National Laboratory.

- **NAMD** - NAMD is a molecular dynamics program for high performance simulation of large biomolecular systems [46]. It is based on Charm++ parallel objects, which is a machine independent parallel programming system. NAMD can use various data sets as input files. We use one called `apoa1`, which models a bloodstream lipoprotein particle.

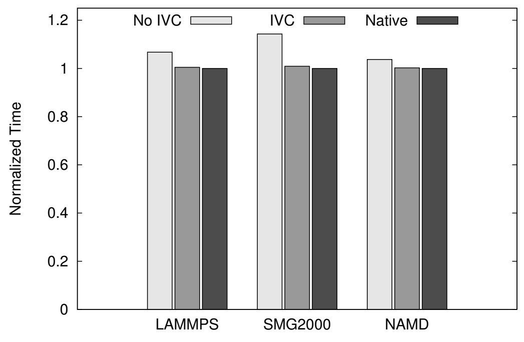
- **SMG2000** - SMG2000 [6] is a parallel semicoarsening multigrid solver for the linear systems on distributed memory computers. It is written in C using MPI.

- **HPL** - High Performance Linpack (HPL) is the parallel implementation of Linpack [45] and the performance measure for ranking computer systems for the Top 500 supercomputer list.

Figure 6.17 shows the evaluation results on Testbed B, with all results normalized to performance achieved in the native environment. IVC is able to greatly close the performance gap between the no-IVC and native cases. Compared with no-IVC, IVC improves performance by up to 11% for the NAS Parallel Benchmarks – IS (11%) and CG (9%). We observe 5.9%, 11.8%, and 3.4% improvements in LAMMPS, SMG2000 and NAMD, respectively.



(a) NAS Parallel Benchmarks

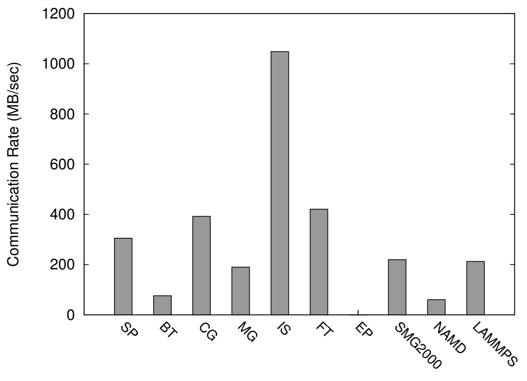


(b) LAMMPS, SMG2000 and NAMD

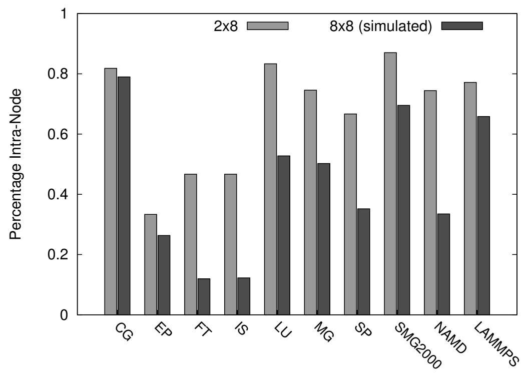
Figure 6.17: Application-level evaluation on Testbed B (2 nodes with 8 cores each)

We further analyze the communication patterns of these benchmarks. Figure 6.18(a) introduces a communication rate metric, which is the total volume of messages sent by each process divided by execution time. This represents an approximation of how frequent the application communicates. As we can see, for several applications which have a high communication rate, such as NAS-IS, NAS-CG, SMG2000 and

LAMMPS, IVC achieves performance improvement as compared with no-IVC. When running on a larger number of computing nodes, some applications could benefit less from IVC because a larger percentage of peers are not located on the same physical host. However, Figure 6.18(b) suggests that IVC is still very important in many cases. We analyze the percentage of data volume that will be sent through IVC on clusters with 8 core computing nodes. We find that IVC communication is well above average, especially for CG, MG, SMG2000 and LAMMPS, because the percentage of IVC communication is higher than 50% even on a 64 core cluster[5]. This is because some applications tend to communicate frequently between neighbors, which have high probability of being on the same host for multi-core systems. For those applications, the benefits of IVC are expected to be observable.



(a) Communication Rate (2 x 8cores)



(b) Percentage volume of IVC communication

Figure 6.18: Communication patterns of the evaluated applications

We also observe that IVC achieves comparable performances with the native configuration. The overhead is marginal (within 1.5%) in most cases. The only exception

[5]Data is collected through simulation on Testbed A, which has a larger number of processors.

is NAS-FT, where a performance degradation of 6% is observed. This is due to the main communication pattern of NAS-FT, an all-to-all personalized operation with very large message sizes. In MVAPICH2 (and also MVAPICH2-ivc), all-to-all personalized operations are implemented on top of non-blocking send operations. As noted earlier, IVC only uses a 16 page shared memory space, which takes multiple iterations of the buffer space to send out a large message, hurting communication progress. Meanwhile, MVAPICH2 incorporated an optimized shared memory communication method proposed by Chai et al. [8], which leads to better performance than IVC. Fortunately, FT is currently the only application we have noticed which has such a communication pattern. Thus, in most cases we will not notice this performance gap. Optimizing IVC for better performance under such communication patterns is also planned, since optimizations used in MVAPICH2 are also possible for IVC.

In order to examine the performance of MVAPICH2-ivc on a larger scale cluster, a 64-node VM-based environment was set up on Testbed A. Figures 6.19 and 6.20 show the performance comparison of NAS Parallel Benchmarks and HPL on Testbed A. Because systems of Testbed A are 2 processors per node with single-core only, the percentage of IVC communication is small compared to inter-node communication through the network. Thus, IVC and no-IVC configurations achieve almost the same performance here and the no-IVC results are omitted for conciseness. Compared with the Native configuration, we observe that the VM-based environment performs comparably. In most cases the performance difference is around 1%, except for NAS-FT, which degrades around 5% because of its large message all-to-all personalized communication pattern.

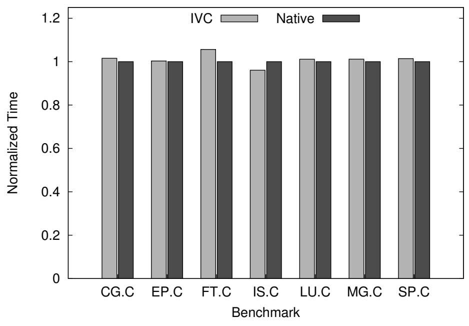


Figure 6.19: Normalized Execution time of NAS (64 nodes with 2 processors each)

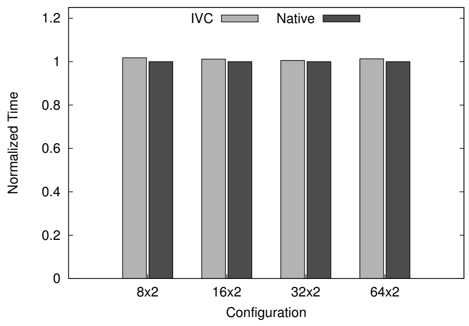


Figure 6.20: Normalized Execution time of HPL

CHAPTER 7

EFFICIENT ONE-COPY SHARED MEMORY COMMUNICATION

In this chapter, we continue the study conducted in the last chapter and present an efficient one-copy shared memory communication scheme for MPI applications in a VM environment. Instead of following the traditional approach used in most MPI implementations, copying data in and out of a pre-allocated shared memory region, our approach dynamically maps the user buffer of the sender between VMs, allowing data to be directly copied to the destination buffer. We also propose a grant/mapping cache to reduce the expensive mapping cost in a VM environment.

The rest of this chapter is organized as follows: we first describe our motivation in Section 7.1. Then, we discuss the design of our one-copy shared memory communication protocol and the grant/mapping cache in Sections 7.2 and 7.3, respectively. Finally, the performance evaluation is presented in Section 7.4.

7.1 Motivation

In the last chapter, we proposed inter-VM communication and its application through an MPI implementation. A shared memory region is created between two computing processes located in separate VMs via the page mapping mechanism from

Xen. After the memory region is mapped to the address space of both communicating processes, communication happens with no difference as in a native environment: To send a message, the sender process copies the data into the shared region and the receiver copies the data from the shared region to the user buffer.

This inter-VM shared memory communication eliminates the performance gap with native computing environments caused by hosting computing processes in separate VMs. However, there is still room for further optimization. For example, the size of the shared memory region is limited. Thus, it may take multiple copies to send large messages, wasting CPU cycles to coordinate between sender and receiver. It should be noted, however, that this problem exists not only for IVC, but for the MPI shared memory communication in native environments as well. In native environments, researchers remedy such limitations by proposing kernel-assisted one-copy communication [24]. The basic idea is to dynamically map the user buffers to the communicating peer's address space so that the data copy can happen directly between the source and destination buffers. This requires only one memory copy to send a message and eliminates the need for an intermediate shared memory region.

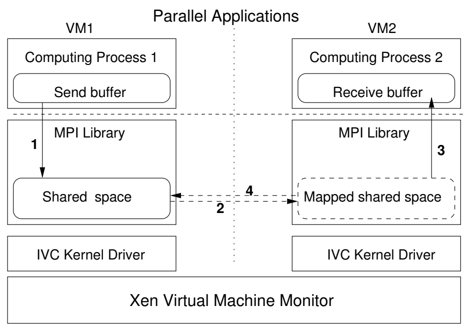
The same idea is also applicable to the VM environment. We can dynamically establish shared mappings of user buffers to allow the receiver process to directly copy data from the sender buffers. There are additional complexities, however. For example, mapping pages between VMs is a costly operation. Furthermore, the number of pages that can be mapped between VMs are limited based on the Xen implementation. Thus, we also propose an Inter-VM Grant/Mapping Cache (IGMC) framework to reduce the mapping cost while using limited resources. We will introduce the design of the basic protocol and the IGMC cache in the next two sections.

7.2 One-copy Protocol for Inter-VM Shared Memory Communication

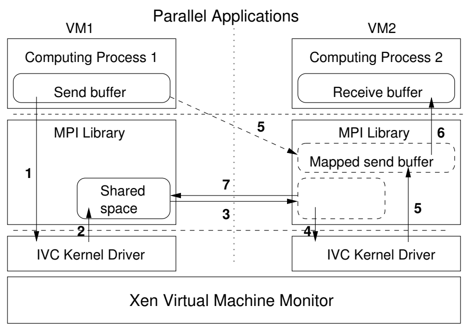
In this section we present the one-copy protocol for efficient inter-VM shared memory communication. The key idea of our design is to dynamically map the user buffer of the sender to the address space of the receiver. With this mapping in place the data can be directly copied into the user buffer of the receiver, saving one copy from the traditional userspace memory copy (two-copy) approach described in the last section.

Figure 7.1 shows an overview of the proposed protocol. It illustrates the process to send an MPI message between two computing processes using the two-copy approach proposed in [19] as well as our new one-copy approach. These two computing processes are hosted in separate VMs on the same physical host. The two-copy approach, as illustrated in Figure 7.1(a), exchanges the message through a pre-mapped shared memory region. This shared memory region is created with the help of the IVC kernel driver and the Xen VMM. The sender copies the data to the shared region (step 1 and 2), the receiver detects the data through polling, and copies the data to the receiver user buffer (step 3). After the copy completes, the receiver acknowledges the message (step 4) so that the sender can re-use the shared memory region.

To reduce the number of copies, we directly map the sender user buffer to the address space of the receiver. Detailed steps are labeled in Figure 7.1(b) and referred to next. Once the MPI library gets a send request, it grants access to the memory pages containing the sender user buffer through the IVC kernel driver (step 1). Unlike the previous method, instead of copying the data to the shared memory region, it only copies the aforementioned reference handles (step 2). The MPI library at the receiver



(a) The two-copy protocol



(b) The one-copy protocol

Figure 7.1: Protocols to send MPI messages

side discovers the send requests (step 3) and maps the user buffer of the sender to its own address space using the grant references from the sender (step 4 and 5). With the mapping in place, the receiver directly copies the data to the destination user buffer (step 6). Once the copy completes, the receiver unmaps the sender pages and notifies the sender so that the sender can revoke the grant of the user buffers (step 7).

Granting page access to remote domains at the sender side and mapping user buffers at the receiver side are both privileged operations that require the involvement of guest kernels as well as VMM. Thus, they are computationally costly operations. Given this startup overhead, we only use the one-copy protocol to send large messages, with small messages still transferred using the traditional two-copy protocol described in Figure 7.1(a).

To further hide the sender-side cost from the receiver we used a pipelined send protocol. Instead of granting access to the entire sender buffer and then copying the reference handles to the shared memory region, the sender buffer is divided into multiple chunks of memory pages. Each time access is granted, one chunk of memory pages and the grant references are immediately sent to the receiver. The receiver then can discover the message and begin the transfer sooner. In this way, the cost of granting page accesses can be totally hidden except for the first chunk. Note that grants do not need to be revoked immediately after the receiver finishes copying one chunk, but can be delayed until the whole message is finished. This eliminates the need to coordinate between sender and receiver while sending a message. Another reason to use a pipeline mechanism is that the number of pages that can be mapped from another domain is limited. Thus, if an application needs to send an exceptionally large message, a pipelined send is the only option. It is to be noted that such case happens rarely since this limit is at least tens of Megabytes.

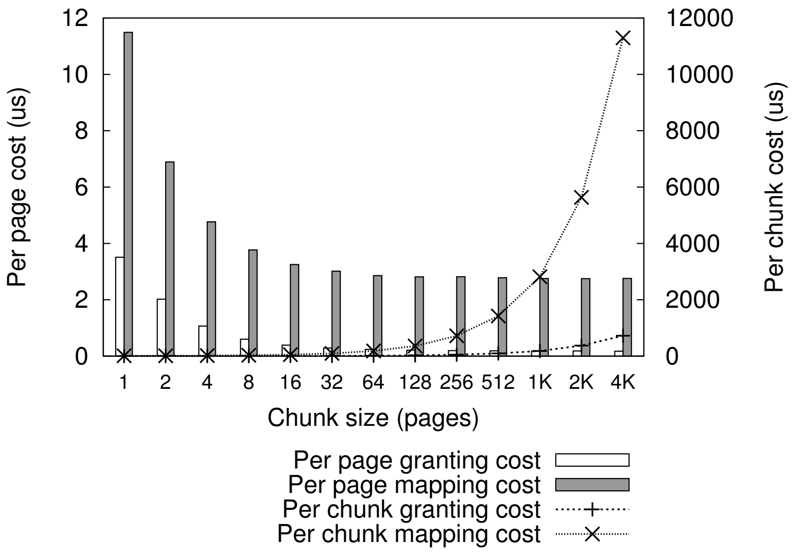


Figure 7.2: Cost of granting/mapping pages

The pipeline size (number of pages per chunk) must be carefully chosen. It is inefficient to use too small of a chunk because there are high startup overheads to grant page access at the sender side and map pages at the receiver side. Larger chunk size allows us to amortize these costs. Figure 7.2 shows the cost to grant/map a chunk of various numbers of pages and the per page cost. We observe that mapping the pages at the receiver side is the most costly operation and a 16-page pipeline size allows us to achieve a reasonable efficiency (grant/map cost per page) to map the pages. This value can be adjusted for different platforms as needed.

Once the pipeline size is determined, we also force the start address of each chunk to be aligned with the pipeline size (64KB aligned in our case) except for the first chunk. As we will see in the next section, this greatly improves the grant/mapping cache hit ratio.

7.3 IGMC: Inter-VM Grant/Mapping Cache

A pipelined send protocol can only hide the cost of granting page access at the sender. At the receiver side, however, mapping pages and copying data are both blocking operations and cannot be overlapped. Unfortunately, as we have observed in Figure 7.2, the receiver mapping is a more costly operation due to heavier VMM involvement than the sender page granting.

To address the high mapping cost, we propose an Inter-VM Grant/Mapping Cache (IGMC). The objective is to keep page chunks mapped at the receiver as long as possible. Then, if the sender user buffer is re-used later, it does not need to be mapped again and the data can directly be copied. In the area of high performance interconnects, similar ideas have been proposed to reduce the registration cost for zero-copy network communication [16]. Despite the similarity, IGMC is much more complicated due to two reasons:

- To keep a page mapping at the receiver, the sender must also keep the corresponding page granted as well. Thus, IGMC consists of two separate caches: a *grant cache* at the sender, which caches all the page chunks that are granted, and a *mapping cache* at the receiver, which caches the mapping of those chunks.
- There are upper limits on both the number of pages that can be granted at the sender as well as the number of pages that can be mapped at the receiver. Thus, cache eviction may occur at either the sender or receiver side.

The caching logic complexity at both sides is hidden within the IGMC, simplifying its use. After the IGMC receives a request to grant a chunk of pages at the sender side, the grant cache is searched for entries that contain the requested page chunk as identified by the start buffer address and the length of the chunk. Since a pipelined send protocol is used, most entries will be of the pipeline chunk size, and the start address of those chunks will be aligned. This allows us to achieve higher hit ratios without complicated mechanisms to expand/merge cache entries. An additional complexity is that the buffer addresses used to index the caches are user-space virtual addresses, which can be backed up by different physical pages in different references, causing false hits. We can use two approaches to address this issue. First, the memory pages are pinned within the OS by the sender until the grants are revoked, so that they will not be swapped out while the receiver is copying the data. Second, we can use `mallopt`[6] to ensure that virtual addresses are not reused when the computing

[6]More advanced mechanisms including intercepting all memory management calls can be used here to avoid false hits. We will not discuss these schemes since they are beyond the scope of our study.

process calls `free` and `malloc` operations. Thus, we can ensure that the same virtual addresses always point to the same data during the application life time.

Each chunk of pages granted at the sender is assigned a unique key. At step 3 in Figure 7.1(b), this key is sent along with the reference handles to the receiver. The mapping cache at the receiver uses this key to determine if the corresponding chunk is already mapped or not.

Using the described protocol the grant cache and the mapping cache will work perfectly given unlimited resources to grant/map pages. Unfortunately, only a limited number of pages can be granted or mapped at one time in Xen. Thus, before creating any new entries in the grant or mapping caches, unused entries may need to be evicted to free resources if the limit is being approached. While identifying unused entries can be detected through reference counters, evicting cache entries requires coordination from both the grant and mapping caches.

7.3.1 Evicting Cache Entries

We limit the number of entries in both the grant cache and the mapping cache[7]. Since each entry corresponds to a chunk of up to the pipeline size, we also have a limit on the maximum number of pages that can be granted/mapped at a single time. IGMC behaves more elegantly in this way by avoiding granting/mapping failure at the guest OS/VMM level. Limiting grant cache entries may not be needed in Xen, because granting access to a page only consumes few bytes of bookkeeping information in the Xen hypervisor. Thus, this is less of a concern compared to the limit on the maximum allowed mapped pages. However, we do not want to limit ourselves to such Xen-specific features; we instead design our protocol be applicable to other VMM

[7]This limit is set to be 8,192 pages in our implementation.

implementations as well. An additional reason to limit the number of entries in the grant cache (at the sender) is that if we run out of resources at the receiver and fail to map the pages, the receiver will have to notify the sender to fall back to the two-copy approach. It is much more efficient for the sender to detect imminent failure so the two-copy approach can be directly selected.

Before evicting grant cache entries and revoking access to the corresponding memory page chunks, those page chunks must not be mapped at the receiver side. If the chunk is still mapped, the sender has to notify the receiver to unmap it before revoking the grant. Thus, at the sender side, we must be able to track which chunks are still in the receiver mapping cache, as well as notify the receiver to unmap a specific chunk. To achieve this, we allocate two bitmaps in the shared memory region, as shown in Figure 7.3. Each of them contains the same number of bits as the maximum number of allowed entries in the grant cache. The sender notifies the receiver to unmap a chunk by setting the corresponding bit in the control bitmap. The receiver will poll on the control bitmap in every library call. Once the receiver discovers the ith bit in the control bitmap is set, it unmaps the entry with the key i. The second bitmap, the mapping bitmap, is set/cleared by the receiver mapping cache. A set bit in the mapping bitmap indicates that the receiver is still keeping the map of the corresponding chunk.

At the receiver side, the mapping cache will evict entries when it is running out of available resources to map new page chunks, or when it is advised by the sender to do so. In either case, the corresponding bits in the mapping bitmap must be cleared to notify the sender that it no longer is maintaining the mapping of the specified chunk.

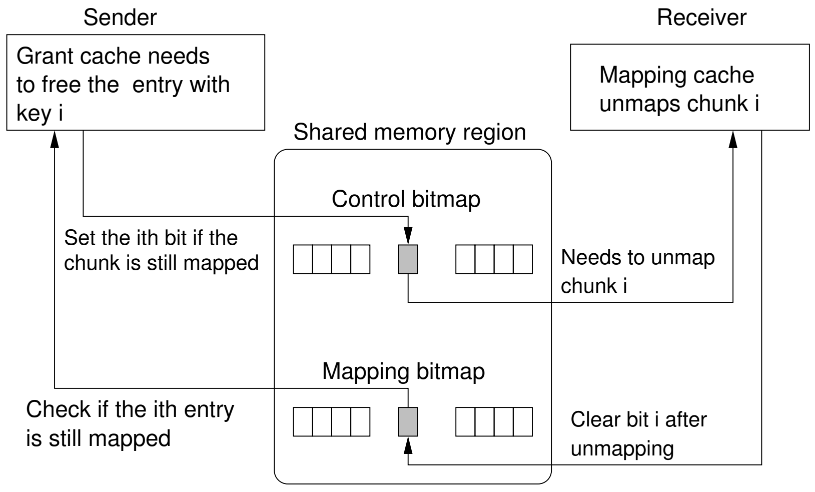

Sender
Grant cache needs to free the entry with key i
Receiver
Mapping cache unmaps chunk i
Shared memory region
Control bitmap
Set the ith bit if the chunk is still mapped
Needs to unmap chunk i
Mapping bitmap
Check if the ith entry is still mapped
Clear bit i after unmapping

Figure 7.3: Exchanging control information through bitmaps

7.3.2 Improving the Cache Hit Ratio

Since it is very costly to map/unmap pages, optimizing the cache hit ratio will be an important issue. We choose to use Least Recently Used (LRU) currently to manage both the grant cache and the mapping cache, since it is relatively simple to implement. Using simple LRU replacement, however, may result in poor hit ratios in some cases. There are numerous studies in the literature to improve the cache hit ratio, such as LRU/k [41], 2Q [25], LIRS [23], etc. We plan to implement 2Q as it admits only re-visited cache entries into the main LRU queue, which can effectively reduce the impact of cache entries that are only referred once. Since the grant cache and the mapping cache can interfere with each other (i.e., before a entry can be removed from the grant cache, the corresponding entry must be removed from the mapping cache), other cache replacement algorithms which take more access history information into account could be more effective. However, the actual impact of different cache replacement algorithms is beyond the scope of our study.

7.3.3 Fall-back Mechanism

No matter how hard we try to optimize the cache replacement algorithm, there could always be a few applications having poor access patterns, which cause a low cache hit ratio. This is true especially if the application working set (user buffers involved in intra-node communication) is too large to fit in the cache. In this case, instead of frequently mapping/unmapping memory pages, which hurts the performance, it is more efficient to fall back to the original two-copy based shared memory communication. The decision on falling back is made based on the hit ratio of the more costly mapping cache at receivers. Each process will track the overall hit ratio and the communication (receive) frequency to each sending peer. If the overall hit ratio is under a certain threshold for a long period of time (i.e., several hundred messages), the process will identify the group of sending peers that communicate the least often and start to use two-copy protocol with these senders. To notify the senders, the process will raise a flag at the shared memory region so that the corresponding sender will send messages using the traditional two-copy protocol. We treat each peer differently since the communication pattern could be different among various peers. It also allows the flexibility to fall back on some of the peers to reduce the application working set. In this case, there is a good chance that the user buffers used in communication to the rest of the peers can fit in the IGMC cache. It is to be noted that the fall-back cases will not happen often, especially for larger scale applications, where only part of the communication is intra-node.

The two-sided fall-back mechanism will also be used for handling non-contiguous datatype communication. This is because "holes" in data buffers may cause inefficiency in granting/mapping all buffer pages. Additionally, the complexity of passing data layout information to the receiver may also overshadow the benefits of the one-copy protocol.

7.4 Evaluation

In this section we evaluate our one-copy approach. We integrate the one-copy protocol into MVAPICH2 [37], and compare with the two-copy based inter-VM shared memory communication protocol discussed in the last chapter. We also compare with unmodified MVAPICH2 running in a native Linux environment. We first demonstrate the improvements on latency and bandwidth benchmarks. Then we present evaluation results using the NAS Parallel Benchmark Suite [39].

7.4.1 Experimental Setup

The experiments were carried out on an Intel Clovertown system. The computing node is equipped with a dual-socket quad-core Xeon E5345 2.33GHz and 6 GB of memory. We run Xen-3.1 with the 2.6.18 kernel on the computing node for VM-based environments. We launch up to eight VMs on the node. Each of VMs will be assigned a dedicated core when running CPU intensive jobs. Each VM consumes 512MB of memory. Even though the modified MPI is certainly capable of running multi-node applications, our experiment is within one node since the focus of this paper is intra-node communication. As demonstrated by our study [8, 19], benefits shown by single-node study can be effectively propagated to multi-node tasks due to the importance of intra-node communication. For native environment evaluation, we run RHEL 5 with the 2.6.18 kernel. We evaluate three configurations in this section:

- **IVC-one-copy** - Our new one-copy based scheme in a VM-based environment;

- **IVC-two-copy** - Previous inter-VM shared memory communication in VM environments, as proposed in the last chapter, using a two-copy method;
- **Native** - Unmodified MVAPICH2 running in the native environment, using a two-copy method.

7.4.2 Micro-benchmarks Evaluation

Figures 7.4 and 7.5 are the MPI level latency and bandwidth. In our current design, the new one-copy scheme is only used for messages larger than 64KB. Any smaller messages will be sent using the old two-copy schemes (IVC-two-copy), so we focus on messages larger than 64KB. Our benchmarks repeatedly send and receive using the same user buffer. Thus, our one copy scheme can greatly benefit from the grant/map cache. Also, with the mappings cached, the one-copy scheme simply copies data between the same buffers when sending/receiving from the same position. The performance, therefore, is significantly better due to CPU cache effects as compared with two-copy schemes. Here the IVC-one-copy configuration is able to achieve up to 6.5GB/s bandwidth (in-cache copy) and reduce the latency by up to 75% compared with the native environment.

The One-copy approach shows significant improvement when the same buffer is used for communication. Unfortunately, real applications will not repeatedly send data from the same location without even writing into the buffer. Thus, to remove the CPU cache effects and evaluate the performance more accurately, we modify the benchmarks to send and receive messages into a larger user buffer pool in a cyclic pattern. After each iteration of send/receive, both sender and receiver increase the user buffer address so that no two consecutive operations will use the same buffer.

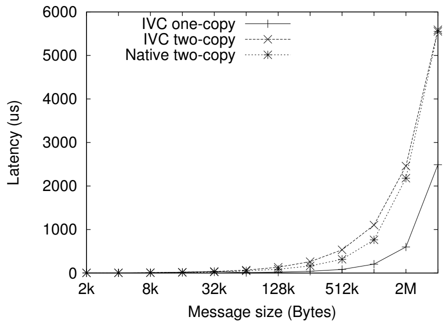


Figure 7.4: MPI latency

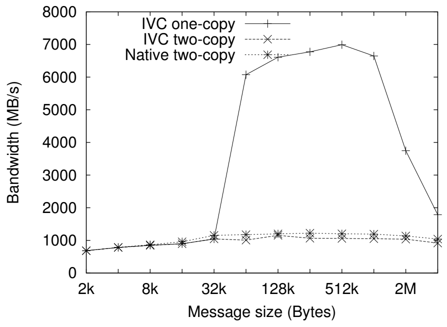


Figure 7.5: MPI bandwidth

Once they reach the end of the buffer pool, however, they will start again from the head of the pool. Figures 7.6 and 7.7 show the latency and bandwidth using a 16MB pool, which is much larger than the L2 cache on the system (4MB). Since the size of the grant/map cache in our design is chosen to be 8K pages, the VM-one-copy scheme will still benefit from it. In this case, the one-copy scheme is able to reduce the latency by up to 35% and increase the bandwidth by up to 38% compared with the native configuration.

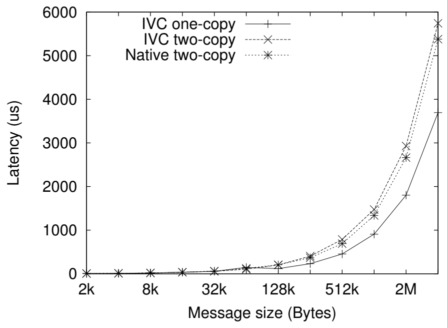


Figure 7.6: MPI latency with cyclic access pattern

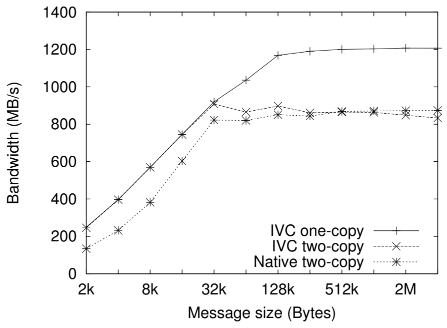


Figure 7.7: MPI bandwidth with cyclic access pattern

If we increase the size of the user buffer pool to 40MB, we will exceed the available entries in the grant/map cache, and it will be forced to evict entries. Those entries, unfortunately, contain the granting/mapping of the user pages that will be re-used later. In this case, the one-copy based scheme is penalized by the high cost of granting/mapping pages in the VM environment. As we can see in Figures 7.8 and 7.9, the one-copy scheme gets up to 39% and 31% worse on latency and bandwidth, respectively, compared with the native environment. This is expected due to the mapping cost illustrated in Figure 7.2. This example demonstrates the importance of optimizing the cache hit ratio and dynamically falling back to the two-copy scheme if too many grant/mapping cache misses are observed. If we fall back to the two-copy based scheme, we will still have the native-level performance, delivered by the IVC-two-copy configuration.

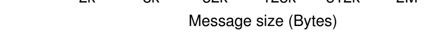


Figure 7.8: MPI latency (mapping cache miss)

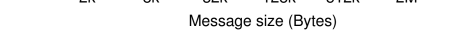


Figure 7.9: MPI bandwidth (mapping cache miss)

Table 7.1: Communication pattern of NPB (Class A)

	> 64KB (count)	> 64KB (volume)	Comm. rate
IS	56.4%	99.9%	497.4 MB/s
CG	0.0%	0.0%	111.5 MB/s
LU	1.60%	67.2%	32.0 MB/s
MG	14.0%	78.5%	66.7 MB/s
BT	73.3%	90.7%	8.6 MB/s
SP	98.8%	100.0%	26.6 MB/s
FT	52.5%	100.0%	211.1 MB/s
EP	0.0%	0.0%	0.0 MB/s

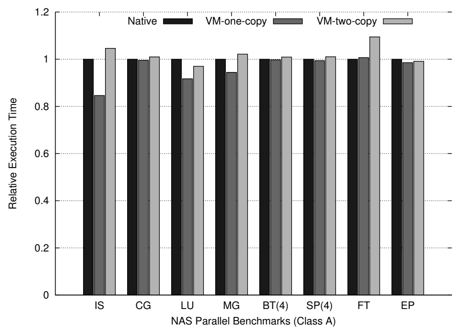
7.4.3 NAS Parallel Benchmarks

In this section, we evaluate our one-copy approach on the NAS Parallel Benchmarks (NPB) [39]. This is a popular parallel benchmarks suite containing computing kernels typical of various fluid dynamics scientific applications.

NPB contains multiple benchmarks, each of which have different communication patterns. Table 7.1 summarizes the communication characteristics of NPB assuming it is running on eight processes (maximum for one dual-socket Quad-core system). Here we show the communication intensity in terms of the volume of messages sent per second, and the percentage of large MPI messages (64KB, the threshold above which messages will be sent using the one-copy approach).

Figure 7.10 presents the performance comparison between the Native, VM-two-copy and VM-one-copy configurations. As the figure shows, our one-copy approach is able to meet or exceed the performance of both Native and VM-two-copy configurations for all benchmarks. In particular, for IS a 15% improvement over Native and 20% improvement over VM-two-copy is achieved. Other benchmarks, including

LU and MG also show a 8%/5% improvement and 6%/7% improvement, respectively over Native and VM-two-copy.



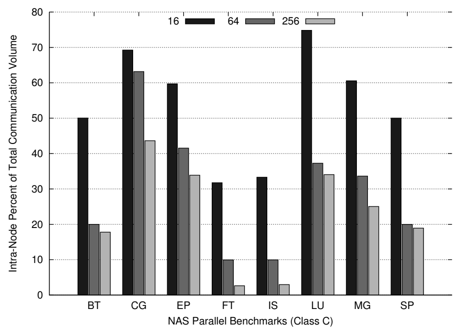


Figure 7.10: Relative performance for NPB

Figure 7.11: Percentage of intra-node communication for larger scale run with NPB

Both message sizes as well as message patterns affect the performance using the different mechanisms. For IS, 56% of messages and 99% of message volume is for messages greater than 64KB. SP shows a similar pattern, with over 98% of messages greater than 64KB. Despite these similarities, the performance improvement for SP using our one-copy design is minimal as compared to IS. LU and MG, with significant large message transfer (67% and 79% of volume), but less than IS or SP, show higher performance using the one-copy mechanism. This can be attributed to a pattern where latency is of increased importance or better processor cache utilization since a one-copy transfer does not pollute the sender cache as occurs with a traditional two-sided copy technique.

Though our MPI is able to run across multiple nodes, the results are shown on a single computing node since the focus of the paper is intra-node communication. On a larger scale, where part of the communication will be inter-node, applications could benefit less as shown in Figure 7.10. However, simulation of communication patterns on a larger cluster as shown in Figure 7.11 suggests that intra-node communication is still very important even when applications span across multiple nodes - some applications tend to communicate more frequently between neighbors. We show the percentage of intra-node communication for NPB on 16, 64, and 256 processes (assuming each computing node has 8 cores) and observe that intra-node communication can be up to 40% even on 256 processes.

CHAPTER 8

HIGH PERFORMANCE VIRTUAL MACHINE MIGRATION OVER RDMA

In this chapter we present the design of high performance VM migration over RDMA, as the highlighted part in Figure 8.1. We first take a closer look at the potential benefits of migration over RDMA, which motivates our design. Then we analyze several design challenges to fully exploit the benefits of RDMA and present how we address those challenges. Finally, we present the performance results in Section 8.4.

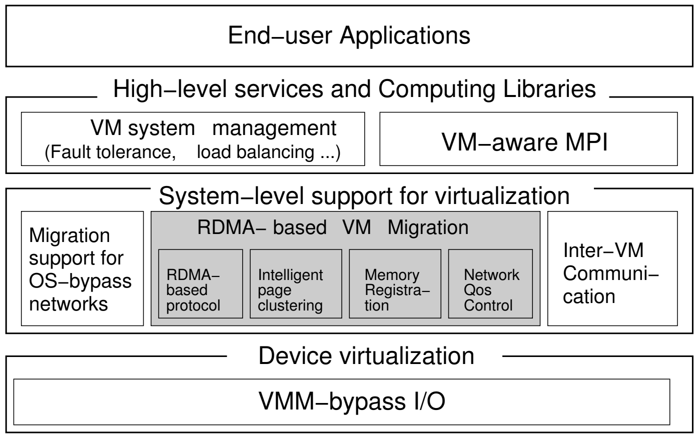

End−user Applications
High−level services and Computing Libraries
VM system management (Fault tolerance, load balancing ...)
VM−aware MPI
System−level support for virtualization
Migration support for OS−bypass networks
RDMA− based VM Migration
RDMA− based protocol
Intelligent page clustering
Memory Registra− tion
Network Qos Control
Inter−VM Communi− cation
Device virtualization
VMM−bypass I/O

Figure 8.1: High performance VM migration in proposed research framework

8.1 Potential Benefits of RDMA based Migration

Besides the increase of bandwidth, RDMA can benefit virtual machine migration mainly from two aspects.

First, RDMA allows the memory to be directly accessed by hardware I/O devices without OS involvement. It indicates that with proper handling of memory buffer registration, the memory pages of the migrating OS instance can be directly sent to the remote host in a zero-copy manner. This avoids the TCP/IP stack processing overhead. Also for virtual machine migration, it avoids excessive context switches between the migrating VM and the privileged domain, which hosts the migration helper process.

Second, the one sided nature of RDMA operations alleviates the burden on the target side during data transfer. This further saving on the CPU cycles is especially important in some cases. For instance, one of the goals of VM technology is server consolidation, where multiple OSes are hosted in one physical box to efficiently utilize the resources. Thus, in many cases a physical server may not have enough CPU resources to handle migration traffic without degrading the hosted application performance.

Direct memory access and the one sided nature of RDMA can significantly reduce the software involvement during migration. As a result, applications hosted will experience much fewer context switches and cache misses, thus being less affected by VM migrations.

8.2 Design Challenges

Though RDMA has the potential to greatly improve the VM migration efficiency, we need to address multiple challenges to fully exploit the benefits of RDMA. Now we take a closer look at these challenges. Our description here focuses on Xen migration and InfiniBand RDMA. However, the issues are common to other VM technologies and RDMA-capable interconnects.

- **Design of efficient migration protocols over RDMA:** As we have mentioned in Section 2.1, there are two kinds of memory pages needed to be transferred during migration. Normal data pages can be directly transferred, but page-table pages need to be pre-processed before being copied out. Our migration protocol should be carefully designed to efficiently handle both types of memory pages. Also, RDMA write and RDMA read both can be utilized for data transfer, but they have different impacts on the source or destination hosts, depending on where the operations are initiated. How to minimize such impact during migration needs careful consideration.

- **Memory Registration:** InfiniBand requires the data buffers to be registered before they can be involved in data transfer. Research works on InfiniBand [29] typically use two schemes to handle registration. Users either copy the data into a pre-registered buffer and send, or register the data buffers on the fly. However, in our case neither of these two schemes work well. Copying the data into pre-registered buffers will consume CPU cycles and pollute data caches, thus involving the same problems as TCP transfer. Registering the user buffers will fail because the involved memory pages do not belong to the migration helper process, or the OS where the InfiniBand driver runs.

- **Non-contiguous Transfer:** Original Xen live migration transfers the memory pages in page granularity. Each time the source host only sends one memory page to the destination host. This may be fine for TCP/IP communication, but it causes underutilization of network link bandwidth when transferring pages over InfiniBand RDMA. It is more desirable to transfer multiple pages contiguous in memory in one RDMA operation to fully utilize the link bandwidth.

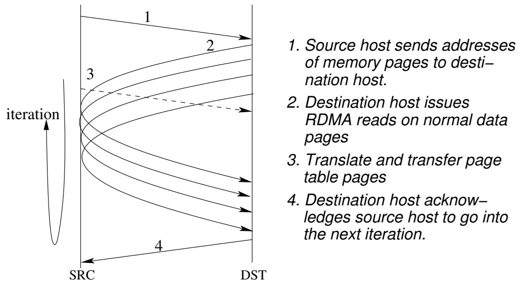
- **Network QoS:** Though RDMA avoids most of the software overhead involved in page transfer, the migration traffic contends with other applications for network bandwidth. It is preferable to explore an intelligent way that minimizes the contention on network bandwidth, while utilizing the network bandwidth efficiently.

8.3 Detailed Design Issues and Solutions

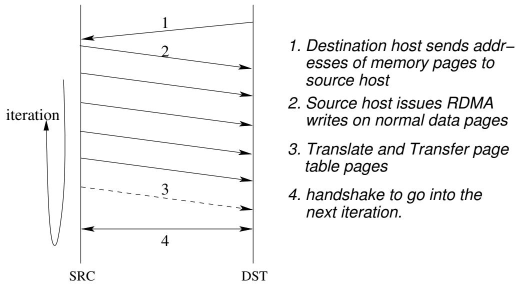
We now describe how we address the aforementioned design challenges:

8.3.1 RDMA based Migration Protocols

As we have mentioned, there are two kinds of memory pages that need to be handled during migration. Normal memory pages will be transferred to the destination host directly, and the page table pages will have to be translated to use machine independent *pfn* before being sent. Translating the page-table pages consumes CPU cycles, while other pages can be directly sent using RDMA.


1
2
3
iteration
4
SRC
DST
1. Source host sends addresses of memory pages to destination host.
2. Destination host issues RDMA reads on normal data pages
3. Translate and transfer page table pages
4. Destination host acknowledges source host to go into the next iteration.

(a) Migration over RDMA read


1
2
3
4
iteration
SRC
DST
1. Destination host sends addresses of memory pages to source host
2. Source host issues RDMA writes on normal data pages
3. Translate and Transfer page table pages
4. handshake to go into the next iteration.

(b) Migration over RDMA write

Figure 8.2: Migration over RDMA

Both RDMA read and RDMA write operations can be used to transfer the memory pages. We have designed protocols based on each of them. Figure 8.2 is a simplified illustration of RDMA related traffic between the migration helper processes in one iteration of the pre-copy stage. The actual design uses the same concept, but is more complex due to other issues such as flow control. Our principle is to issue RDMA operations to send normal memory pages as early as possible. While the transfer is taking place, we start to process the page-table pages, which requires more CPU cycles. In this way, we overlap the translation with data transfer, and achieve minimal total migration time. We use send/receive operations instead of RDMA to send the page-table pages for two reasons. First, the destination host needs to be notified when the page-table pages have arrived, so that it can start translating the page tables. Using send/receive does not require explicit flag messages to synchronize between the source and destination hosts. Also, the number of page-table pages is small, so most migration traffic is still transferred over RDMA.

As can be seen, the RDMA read protocol requires that more work be done at the destination host while the RDMA write protocol puts more burden on the source host. Thus, we dynamically select the suitable protocol based on runtime server workloads. At the beginning of the pre-copy stage, the source and destination hosts exchange load information and the node with the lighter workload will initiate RDMA operations.

8.3.2 Memory Registration

As indicated in Section 8.2, memory registration is a critical issue because none of the existing approaches, either copy-based send or registration on the fly, works well here. We use different techniques to handle this issue based on different types of memory pages.

For page-table pages, the migration helper processes have to parse the pages in order to translate between machine-dependent *machine frame number (mfn)* and machine-independent *physical frame number (pfn)*. Thus, there will be no additional cost to use a copy-base approach. On the source host, the migration helper process writes the translated pages directly to the pre-registered buffers and then the data can be sent out to the corresponding pre-registered buffers on the destination. On the destination host, the migration helper process reads the data from the pre-registered buffers and writes the translation results into the new page table pages.

For other memory pages there will be an additional cost to use a copy-based approach. And the migration helper process cannot register the memory pages belonging to the migrating VM directly. Fortunately, InfiniBand supports direct data transfer using hardware addresses in kernel space, which allows memory pages addressed by hardware DMA addresses to be directly used in data transfer. The hardware addresses are known in our case, by directly reading the page-table pages (*mfn*). The only remaining issue now is that the helper processes in Xen are user-level programs and cannot utilize this kernel function. We make modifications to InfiniBand drivers to extend this functionality to user-level processes and hence bypass the memory registration issue. Note that this modification does not raise any security concerns because we only export the interface to user processes in the control domain (Dom0), where all programs running in this domain are trusted to be secure and reliable.

8.3.3 Page Clustering

Here we first analyze how Xen organizes the memory pages of a guest VM, and then propose a "page-clustering" technique to address the issue of network under-utilization caused by non-contiguous transfer. As shown in Figure 8.3, Xen maintains an address mapping table which maps machine-independent *pfn* to machine-dependent *mfn* for each guest VM. This mapping can be arbitrary and the physical layout of the memory pages used by a guest VM may not be contiguous. During migration, a memory page is copied to a destination memory page corresponding to the same *pfn*, which guarantees application transparency to migration. For example, in Figure 8.3, physical page 1 is copied to physical page 2 on the destination host, because their corresponding *pfn* are both 3. Xen randomly decides the order to transfer pages to better estimate the page dirty rate. The non-contiguous physical memory layout together with such randomness makes it very unlikely that two consecutive transfers involve contiguous memory pages so that they can be combined.

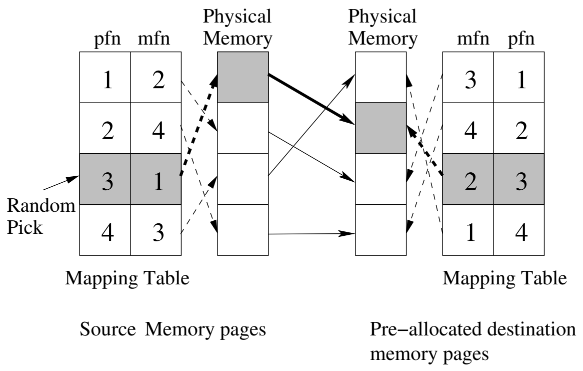
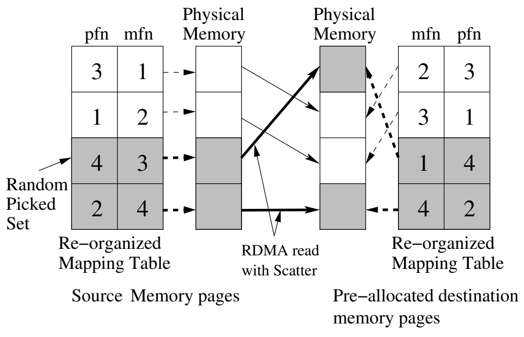
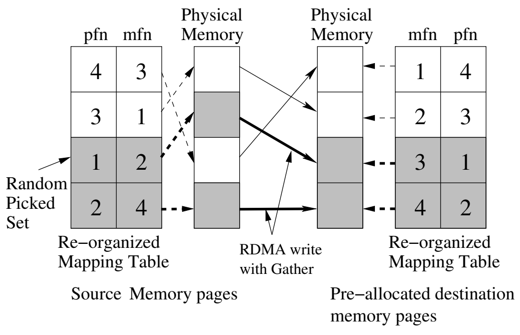


Figure 8.3: Memory page management for a "tiny" OS with four pages

We propose *page clustering* to serve two purposes: first, to send as many pages as possible in one RDMA operation to efficiently utilize the link bandwidth; second, to keep a certain level of randomness in the transfer order for accurate estimation of page dirty rate. Figure 8.4(a) illustrates the main idea of *page clustering* using RDMA read. We first reorganize the mapping tables based on the order of *mfn* at the source host. Now contiguous physical memory pages correspond to contiguous entries in the re-organized mapping table. In order to keep randomness, we cluster the entries of the re-organized mapping tables into multiple sets. Each set contains a number of contiguous entries, which can be transferred in one RDMA operation under most circumstances (We have to use multiple RDMA operations in case a set contains the non-contiguous portion of physical memory pages used by the VM). Each time we randomly pick a set of pages to transfer. As shown in the figure, with sets of size two the whole memory can be transferred within two RDMA read operations. The size of each set is chosen empirically. We use 32 in our actual implementation. Note that the memory pages on the destination host need not be contiguous, since InfiniBand supports RDMA read with scatter operation. The RDMA write protocol also uses the similar idea, except that we need to reorganize the mapping tables based on the *mfn* at the destination host to take advantage of RDMA write with gather, as shown in Figure 8.4(b).



(a) RDMA read



(b) RDMA write

Figure 8.4: Re-organizing mapping tables for page clustering

8.3.4 Network Quality of Service

By using RDMA-based schemes we can achieve minimal software overhead during migration. However, the migration traffic will unavoidably consume a certain amount of network bandwidth, and thus may affect the performance of other hosted communication-intensive applications during migration.

To minimize network contention, Xen uses a dynamic adaptive algorithm to limit the transfer rate of the migration traffic. It always starts from a low transfer rate limit at the first iteration of pre-copy. Then the rate limit is set to a constant increment to the page dirty rate of the previous iteration, until it exceeds a high rate limit, when Xen will terminate the pre-copy stage. Although the same scheme can be used for RDMA-based migration, we would like a more intelligent scheme because RDMA provides much higher network bandwidth. If there is no other network traffic, limiting the transfer rate unnecessarily prolongs the total migration time. We want the pre-copy stage to be as short as possible if there is enough network bandwidth, but to alleviate the network contention if other applications are using the network.

We modify the adaptive rate-limit algorithm used by Xen to meet our purpose. We start from the highest rate limit by assuming there is no other application using the network. After sending a batch of pages, we estimate the theoretical bandwidth the migration traffic should achieve based on the average size of each RDMA operation. If the actual bandwidth is smaller than that (the empirical threshold would be 80% of the estimation), it probably means that there are other applications sharing the network, either at the source or destination host. Then we reduce the rate of migration traffic by controlling the issuance of RDMA operations. We control the transfer rate under a pre-defined low rate limit, or a constant increment to the page dirty rate of the previous round, whichever is higher. If this rate is lower than the high rate limit, we try to raise the rate limit after sending a number of pages. If there is no other application sharing the network at the time, we will be able to achieve a full bandwidth. In this case, we will keep sending at the high rate limit. Otherwise, we will remain at the low rate limit some more time before trying to raise the limit again. Because RDMA transfers require very little CPU involvement, its throughput depends mainly on the network utilization. Thus, our scheme works well to detect the network contention, and is able to efficiently utilize the link bandwidth when there is less contention on network resources.

8.4 Evaluation of RDMA-based Virtual Machine Migration

Here we present the performance evaluation of RDMA-based migration. We first evaluate the basic migration performance with respect to total migration time and migration downtime. Then we examine the impact of migration on hosted applications using SPEC CINT2000 [56] and NAS Parallel Benchmarks [39]. Finally we evaluate the effect of our adaptive rate limit mechanism on network QoS.

8.4.1 Experimental Setup

We implement our RDMA-based migration design with InfiniBand OpenFabrics verbs [42] and Xen-3.0.3 release [74]. We compare our implementation with the original Xen migration over TCP. To show better performance with the TCP stack, all TCP/IP related evaluations are carried over IP over InfiniBand (IPoIB [20]). Though not shown here, we found that migration over IPoIB always achieves better performance than using the GiGE control networks of the cluster.

The experiments are carried out on an InfiniBand cluster. Each system in the cluster is equipped with dual Intel Xeon 2.66 GHz CPUs, 2 GB memory and a Mellanox MT23108 PCI-X InfiniBand HCA. The systems are connected with an InfiniScale InfiniBand switch. The cluster is also connected with Gigabit Ethernet as the control network. Xen-3.0.3 with the 2.6.16.29 kernel is used on all computing nodes. Domain 0 is configured to use 512 MB of memory.

8.4.2 Basic Migration Performance

In this section we take a look at the basic migration performance achieved through RDMA operations. We first examine the effect of the page-clustering scheme proposed in Section 8.3.

Figure 8.5 compares the total time to migrate VMs with different sizes of memory configurations. We compare four different schemes: migration using RDMA read or RDMA write, and with or without page-clustering. Because page-clustering tries to send larger trunks of memory pages to utilize link bandwidth more efficiently, we observe that it can constantly reduce the total migration time, up to 27% in the case of migrating a 1GB VM using RDMA Read. For RDMA write, we do not see as many benefits of page-clustering as RDMA read. This is because for messages around 4KB, InfiniBand has more optimized RDMA write performance, the bandwidth improvement from sending larger messages becomes smaller. Since page-clustering constantly shows better performance, we use page-clustering in all our later evaluations.

Next we compare the total migration time achieved over IPoIB, RDMA read and RDMA write operations. Figure 8.6 shows the total migration time needed to migrate a VM with varied memory configurations. As we can see, due to the increased bandwidth provided by InfiniBand and RDMA, the total migration can be reduced by up to 80% by using RDMA operations. RDMA read-based migration has slightly higher migration time, this is because the InfiniBand RDMA write operation typically provides a higher bandwidth.

Figure 8.7 shows a root-to-all style migration test. We first launch multiple virtual machines on a source node, with each using 256 MB of memory. We then start migrating all of them to different hosts at the same time and measure the time to finish all migrations. This emulates the requirements posed by proactive fault tolerance, where all hosted VMs need to be migrated to other hosts as fast as possible upon receiving a failure signal. We also show the migration time normalized to the case of migrating one VM. For IPoIB, there is a sudden increase when the number of migrating VMs reaches 3. This is because we have two CPUs on each physical host. Handing the traffic of three migrations leads to contention on not only the network bandwidth, but also the CPU resources. For migration over RDMA, we observe an almost linear increase of the total migration time. RDMA read scales the best here because it puts the least burden on the source physical host so that the contention on the network is almost the only factor affecting the total migration time in this case.

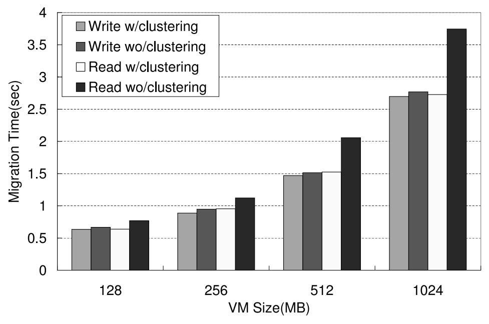


Figure 8.5: Benefits of page-clustering

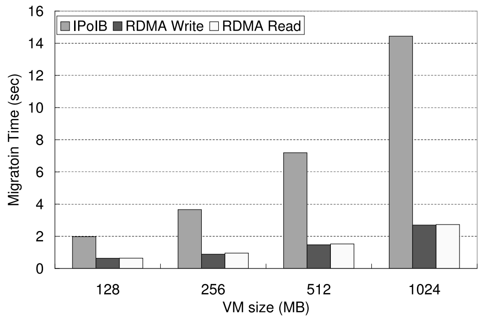


Figure 8.6: Total migration time

We have been evaluating the total migration time that may be hidden from applications through live migration. With live migration, the application will only perceive the migration downtime, which mainly depends on two factors: First is the application hosted on the migrating VM; the faster an application dirties memory pages, the more memory pages may need to be sent in the last iteration of the pre-copy stage, which prolongs the downtime. Second is the network bandwidth; a higher bandwidth shortens the time spent in the last pre-copy iteration, resulting in shorter downtime. To measure the migration downtime, we use a latency test. We start a ping-pong latency test over a very accurate approximation of the migration downtime, because a typical roundtrip over InfiniBand will take less than 10 μs.

We conduct the test while having a process continuously tainting a pool of memory in the migrating VM. We vary the size of the pool to emulate applications dirtying the memory pages at different rates. Only RDMA read results are shown here because RDMA write performs very similarly. As shown in Figure 8.8, the downtimes of migrating over RDMA or IPoIB hardly differ in the case of no memory tainting, because the time to transfer the dirty pages in the last iteration is very small compared with other migration overhead such as re-initializing the device drivers. While increasing the size of the pool, we see a larger gap in the downtime, because it takes a longer time to transfer the dirty memory pages in the last iteration of the pre-copy stage. Due to the high bandwidth achieved through RDMA, the downtime can be reduced drastically, up to 77% in the case of tainting a pool of 256MB memory.

In summary, due to increased bandwidth, RDMA operations can significantly reduce the total migration time and the migration downtime in most cases. Low software overhead also gives RDMA extra advantages while handling the traffic of multiple migrations at the same time.

8.4.3 Impact of Migration on Hosted Applications

Now we evaluate the actual impact of migration on applications hosted in the migrating VM. We use SPEC CPU2000 [56], which provides a comparative measure

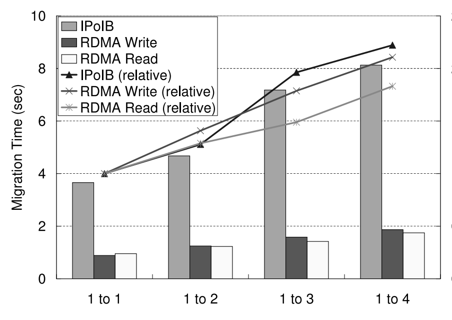


Figure 8.7: "Root-to-all" migration

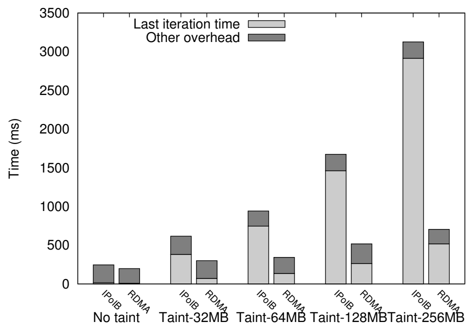


Figure 8.8: Migration downtime

of compute-intensive performance across a practical range of hardware. We run the integer benchmarks (CINT) in a 512MB guest VM and migrate the VM back and forth between two different physical hosts. Figure 8.9 shows the running time of each CINT benchmark after migrating the host VM eight times. As we can see, live migration is able to hide the majority of the total migration time to the hosted applications. However, even in this case, the RDMA-based scheme is able to reduce the migration overhead over IPoIB by an average of 54%.

For the results in Figure 8.9, we have 2 CPUs on each host, providing enough resources to handle the migration traffic while the guest VM is using one CPU for computing. As we mentioned in Section 8.1, in a real production virtual machine environment, we may consolidate many servers onto one physical host, leaving very few CPU resources to handle migration traffic. To emulate this case, we disable one CPU on the physical hosts and conduct the same test, as shown in Figure 8.10. We observe that migration over IPoIB incurs much more overhead in this case due to the contention for CPU resources, while migration over RDMA does not have much more overhead than the 2 CPU case. Compared with migration over IPoIB, RDMA-based migration reduces the impact on applications by up to 89%, by an average of 70%.

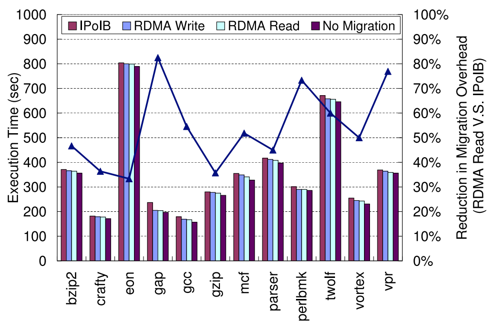


Figure 8.9: SPEC CINT 2000 (2 CPUs)

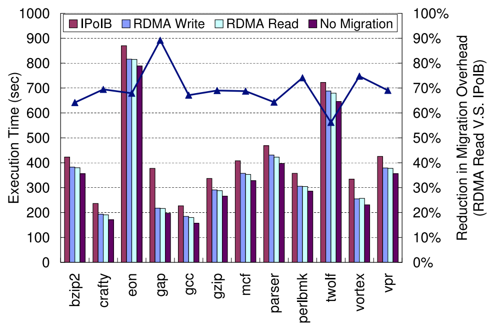


Figure 8.10: SPEC CINT 2000 (1 CPU)

Migration will affect the application performance not only on the migrating VM, but also on the other non-migrating VMs on the same physical host as well. We evaluate this impact in Figure 8.11. We first launch a VM on a physical node, and run SPEC CINT benchmarks in this VM. Then we migrate another VM to and from that physical host at 30-second intervals to study the impact of migrations on the total execution time. We use one CPU in this experiment. We observe the same trend that migration over RDMA reduces the overhead by an average of 64% compared with IPoIB. Here we also show the hybrid approach. Based on server loads, the hybrid approach automatically chooses RDMA read when migrating VM out of the host and RDMA write when migrating VM in. Table 8.1 shows the sample counts of total instructions executed in the privileged domain, total L2 cache misses and total TLB misses during each benchmark run. For RDMA-based migration we show the percentage of reductions compared to IPoIB. All of these costs are directly contributing to the overhead of migration. We observe that RDMA-based migration can reduce all the costs significantly, and the hybrid scheme reduces the overhead further compared to RDMA read. The RDMA write scheme, by which the server has less burden when migrating VMs in but more when migrating VMs out, shows very similar numbers compared to RDMA read. Thus we omit RDMA write data for conciseness.

In summary, RDMA-based migration can significantly reduce the migration overhead observed by applications hosted on both the migrating VM and the non-migrating VMs. This is especially true when the server is highly loaded and has less CPU resources to handle the migration traffic.

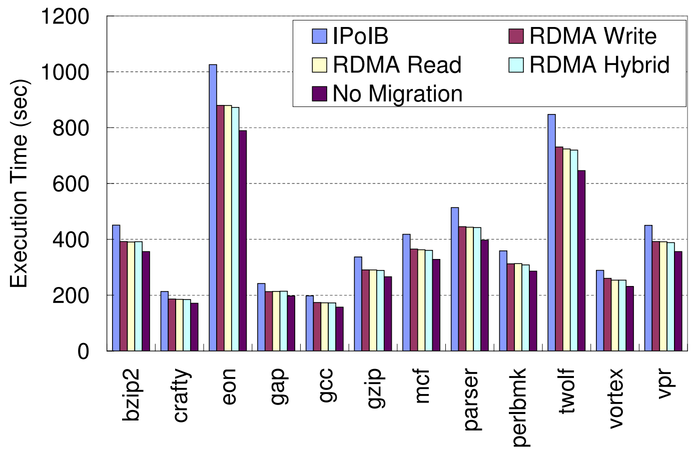


Figure 8.11: Impact of migration on applications in a non-migrating VM

8.4.4 Impact of Migration on Parallel Applications

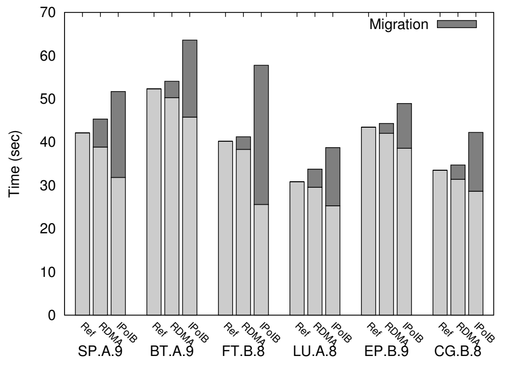
We also study the impact of migration on Parallel HPC applications. We conducted an experiment using the NAS Parallel Benchmarks (NPB) [39], which are a set of computing kernels widely used by various classes of scientific applications.

Table 8.1: Sample instruction count, L2 cache misses and TLB misses (collected using Xenoprof [33])

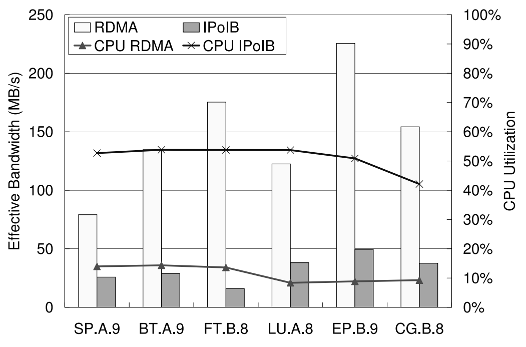
Profile		bzip2	crafty	eon	gap	gcc	gzip	mcf	parser	perlbmk	twolf	vortex	vpr
Inst. Count	IPoIB	24178	9732	58999	12214	9908	16898	21434	28890	19590	47804	14688	23891
	R-Read	-62.7%	-61.1%	-62.5%	-62.0%	-58.4%	-60.6%	-61.8%	-63.3%	-62.5%	-62.5%	-62.40%	-62.06%
	Hybrid	-64.3%	-63.6%	-65.4%	-63.7%	-59.6%	-62.6%	-63.4%	-65.5%	-63.7%	-64.7%	-64.37%	-64.19%
L2 Cache Miss	IPoIB	12372	1718	5714	3917	5359	2285	31554	8196	3523	27384	5567	16176
	R-Read	-10.8%	-36.9%	-56.8%	-15.4%	-13.4%	-43.6%	-3.8%	-21.7%	-28.4%	-12.6%	-13.87%	-10.10%
	Hybrid	-11.1%	-39.6%	-58.8%	-15.0%	-15.7%	-45.3%	-4.0%	-22.6%	-28.5%	-14.8%	-14.03%	-9.81%
TLB Miss	IPoIB	46784	153011	789042	27473	69309	33739	42116	59657	82135	216593	71239	67562
	R-Read	-69.9%	-10.2%	-11.0%	-61.9%	-19.2%	-68.1%	-69.9%	-66.1%	-33.2%	-31.1%	-28.48%	-49.78%
	Hybrid	-73.0%	-10.5%	-10.4%	-64.5%	-19.9%	-70.4%	-73.1%	-68.4%	-34.1%	-32.0%	-29.65%	-51.78%

We use MVAPICH here. The benchmarks run with 8 or 9 processes on separate VMs using 512MB on different physical hosts. We then migrate a VM once during the execution to study the impact of migration on the total execution time, as shown in Figure 8.12(a). Here RDMA read is used for migration, because the destination host has a lower load than the source host. As we can see, the overhead caused by migration is significantly reduced by RDMA, an average of 79% compared with migration over IPoIB. We also mark out the total migration time in the figure. Because of live migration, the total migration time is not directly reflected in the increase of total execution time. We observe that IPoIB has a much longer migration time due to the lower transfer rate and the contention on CPU resources. In HPC it is very unlikely that people will spare one CPU for migration traffic. Thus we use only one CPU on each host in this experiment. As a result, the migration overhead here for TCP/IPoIB is significantly higher than reported by other relevant studies as in [38, 18].

Figure 8.12(b) further explains the gap we observed between migration over IPoIB and RDMA. As we can see, while migrating over IPoIB, the migration helper process in Dom0 uses up to 53% of the CPU resources but only achieves an effective migration throughput up to 49MB/s (calculated by dividing the memory footprint of the



(a) Total execution time



(b) Effective bandwidth and Dom0 CPU utilization

Figure 8.12: Impact of migration on NAS Parallel Benchmarks

migrating OS by the total migration time). Migrating over RDMA, in contrast, is able to deliver up to 225MB/s while using a maximum of 14% of the CPU resources.

8.4.5 Impact of Adaptive Limit Control on Network QoS

In this section we demonstrate the effectiveness of our adaptive rate-limit mechanism described in Section 8.4.5. We set the upper limit of page transfer rate to be 300 MB/s and the lower limit to be 50 MB/s. As shown in Figure 8.13, we first start a bi-directional bandwidth test between two physical hosts, where we observe around 650MB/s throughput. At the 5th second, we start to migrate an 1GB VM between these two hosts. As we can see, the migration process first tries to send pages at the higher rate limit. However, because of the bi-directional bandwidth test, it is only able to achieve around 200 MB/s, which is less than the threshold (80%). The migration process then detects that network contention exists and starts to send pages at the lower rate. Thus, from the bi-directional bandwidth test we observe an initial drop, but very quickly the throughput comes back to 600MB/s level. The migration process tries to get back to the higher rate several times between the 5th and the 15th seconds, but immediately detects that there is still network contention and remains at the lower rate. At the 15th second we stop the bandwidth test, after that the migration traffic detects that it is able to achieve a reasonably high bandwidth (around 267 MB/s), and thus keeps sending pages at the higher rate.

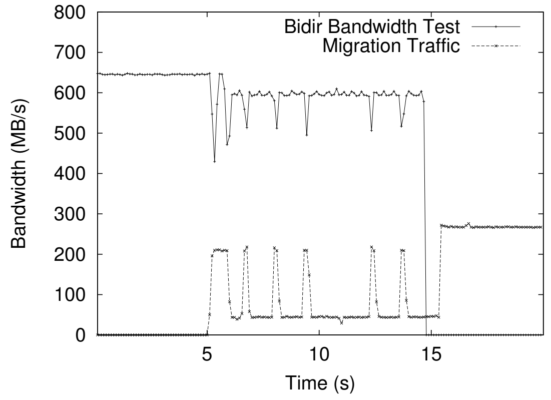


Figure 8.13: Adaptive rate limit control

CHAPTER 9

OPEN SOURCE SOFTWARE RELEASE AND ITS IMPACT

The major portion of the work described in this dissertation has been incorporated into open source software releases.

XenIB, the VMM-bypass I/O driver for InfiniBand in the Xen VM environment, has been initially released through the IBM Linux Technology Center. It has immediately attracted the attention of the InfiniBand community. It has subsequently been maintained through Novell, and is accessible from the OpenFabrics Alliance [42] and Mellanox website [30] (a major InfiniBand vendor). The XenIB software release has also motivated much academic research. The original papers described the XenIB design have been cited more than 70 times according to Google Scholars. And of those, two [75, 52] are using XenIB software as important components of their studies.

The MVAPICH2 work described in Section 6.2.1 is the basis of our MVAPICH2 software package and is also distributed in an open-source manner. The duration of the work in this dissertation has spanned several release versions from 0.6.0 to 1.2 (the most current), and will continue to be used and improved in the upcoming releases.

Current MVAPICH2 software supports many software interfaces, including OpenFabrics [42], uDAPL [59], and VAPI([31], to be obsolete in version 1.2). MVAPICH2 also supports 10GigE networks through iWARP support which is integrated in the OpenFabrics software package. In addition, any network which implements the network independent uDAPL interface may make use of MVAPICH2. Further, MVAPICH2 supports and has been tested on a wide variety of target architectures, such IA32, EM64T, X86_64, IBM Power, and IA64. Besides its high performance and scalable point-to-point message passing and collective implementation, MVAPICH2 also supports several other advanced features, including full Remote Memory Access (RMA) operations defined in the MPI2 standard, applications-transparent Checkpoint/Restart, and dynamic process management in the upcoming release.

MVAPICH/MVAPICH2 software was first released in 2002. Currently there are more than 715 computing sites and organizations worldwide have downloaded this software. In addition, nearly every InfiniBand vendor and the Open Source OpenFabrics stack includes our software in their packages. MVAPICH2 software has been used on some of the most powerful computers, as ranked by Top 500 [53]. Examples include the 62,976-core Sun Blade System (Ranger) cluster at the Texas Advanced Computing Center/University of Texas (TACC), and the 9600 core Abe cluster at the National Center for Supercomputing Applications/University of Illinois (NCSA).

CHAPTER 10

CONCLUSION AND FUTURE RESEARCH DIRECTIONS

This dissertation works towards high performance network I/O virtualization, aiming to break the bottleneck on network I/O for VM technology. The successful research reported in this dissertation eliminates a major barrier that blocks the HPC community from embracing VMs, which are highly desired for manageability reasons. Our work focuses on multiple aspects of the system software stack, and has a significant impact on both industry and academia by contributing novel designs and concepts. In this chapter, we first summarize the research contributions in Section 10.1, and then discuss some possible future research directions in Section 10.2.

10.1 Summary of Research Contributions

There are multiple research contributions from this dissertation. Our research targets the areas of HPC and VM technology, and can benefit both areas. The technology proposed in this dissertation, including VMM-bypass I/O, inter-VM communication and RDMA-based VM migration, can be applied to VM-based environments for both data centers and HPC. We open up this direction and propose VM-based platforms as a possible solution to future large scale cluster computing, and prove its feasibility.

This will have a large impact on HPC. We summarize our research contributions in detail in the rest of this section.

10.1.1 VMM-bypass I/O to Remove Performance Bottleneck in VMs

In Chapter 4, 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. While I/O performance is still considered as one of the major performance bottlenecks in VMs, VMM-bypass I/O significantly improves I/O performance in VMs by completely eliminating context switching overhead between a VM and the VMM or two different VMs caused by current I/O virtualization approaches.

We also developed Xen-IB, as a demonstration of VMM-bypass on InfiniBand in 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 privileged operations. We further support iWARP/10GibE with Xen-IB and achieve similar results, which proves the effectiveness of the VMM-bypass I/O approach.

While VMM-bypass I/O has its own restrictions that it has to be deployed on OS-bypass capable networks, it is still very promising for HPC, the focus of our research. For HPC systems, performance is critical and OS-bypass interconnects such as InfiniBand are considered as standard hardware. Our research removes a major performance bottleneck for VMs and makes them a feasible solution to manage HPC clusters.

The concept of VMM-bypass is also impacting other research efforts to optimize non OS-bypass networks in VMs. Our paper has been cited in several major conference papers including [72, 50], whose target is optimizing TCP/IP-compatible networks in VMs using VMM-bypass concepts. Researchers in [75] and [52] have deployed an InfiniBand cluster with Xen VMs using drivers based on Xen-IB.

10.1.2 Migration of OS-bypass Networks

Migrating network resources is one of the key problems that need to be addressed in the VM migration process. Existing studies of VM migration have focused on traditional I/O interfaces such as Ethernet. However, modern OS-bypass capable interconnects with intelligent NICs pose significantly more challenges as they have additional features including hardware-level reliable services and direct I/O accesses. Especially with the deployment of VMM-bypass I/O, migration of OS-bypass networks becomes increasingly important.

We investigate the solution using a software-based approach. In Chapter 5, we described Nomad, a design for migrating OS-bypass interconnects. We discussed in detail the challenges of migrating modern interconnects due to hardware-level reliable services and direct I/O accesses. We propose a possible solution based on namespace virtualization and coordination protocols, and deliver a prototype implementation.

Our research has stimulated other investigations of this migration challenge. For example, our paper is cited by [52], which proposes migration of InfiniBand resources by applying the concept of coordination at middleware layer (ARMCI).

10.1.3 Inter-VM Shared Memory Communication

In a VM environment, multiple VMs are consolidated onto one physical computing node. There are cases, such as hosting a distributed job, that require these VMs communicate with each other. In the traditional way, these VMs will not understand that their communicating peers are on the same host so the communication will go through network loopbacks. There are much more efficient communication schemes, however, using copy-based approaches over shared memory regions.

We pointed this out in Chapter 6 and proposed *IVC*, an Inter-VM communication library to support shared memory communication between distinct VMs on the same physical host in a Xen environment. *IVC* has a general socket-style API and can be used by parallel application in a VM environment. Our approach also takes VM migration into proper consideration.

We further improve our solution by using a one-copy communication protocol in Chapter 7. We dynamically map the sender user buffer to the address space of the receiver. As compared with traditional two-copy approaches, which are most commonly used, one-copy approaches save the cost of copying data from the sender buffer to the shared memory space. Because granting/mapping operations in VM environments are expensive, we also propose a grant/map cache to amortize this cost.

Inter-VM communication is an important topic in VMs. Researchers [76] at IBM propose similar solutions independently from our work. Our paper is also cited by [68], which achieves TCP-compatible inter-VM communication.

10.1.4 Evaluation with Micro-benchmarks and Real Applications

We have discussed the potentials of a VM-based environment for HPC. Other related research [34, 38, 40] also suggest that VMs can bring benefits including manageability, flexibility, adaptability, and fault tolerance. However, for the HPC community, performance remains a major concern with respect to VM technology.

At the end of every design chapter, we conducted a thorough performance evaluation with both micro-benchmarks and applications-level benchmarks. The main contributions of these evaluations are twofold: first, with micro-benchmarks we demonstrate that virtualizing communication subsystems with VMM-bypass I/O and inter-VM shared memory introduces negligible overhead. Second, the application-level benchmarks we use are commonly related to real life applications: NAS Parallel Benchmarks (NPB) are the computing kernels common on Computing Fluid Dynamics (CFD) applications; High Performance Linpack (HPL) is the parallel implementation of Linpack [45] and the performance measure for ranking the computer systems of the Top 500 supercomputer list; We also use LAMMPS, SMG2000, NAMD, which are real world applications themselves. These applications have different memory usage and communication patterns, and can be good indicators of how user applications will perform. By achieving native-level performance with these benchmarks, we can prove to the community that performance overhead should no longer be a major concern for VM-based HPC environments.

10.1.5 VM Migration with RDMA

One of the most useful features provided by VM technologies is the ability to migrate running OS instances across distinct physical nodes. As a basis for many administration tools in modern clusters and data centers, VM migration is desired to be extremely efficient in reducing both migration time and performance impact on hosted applications.

Currently, most VM environments use the Socket interface and the TCP/IP protocol to transfer VM migration traffic. While migrating over the TCP socket ensures that the solution can be applicable to the majority of industry computing environments, it can also lead to suboptimal performance due to the high protocol overhead, heavy kernel involvement and extra synchronization requirements of the two-sided operations. This overall overhead may overshadow the benefits of migration.

In Chapter 8, we proposed a high performance virtual machine migration design based on RDMA. We addressed several challenges to fully exploit the benefits of RDMA, including efficient protocol designs, memory registration, non-contiguous transfer and network QoS. Our design significantly improves the efficiency of virtual machine migration, in terms of both total migration time and software overhead. Our effort brings significantly more flexibility to the management of VM environment by making the migration cost much lower. This is especially important when RDMA-capable networks are increasingly popular on industry computing systems.

10.2 Future Research Directions

In this dissertation, we look at improving VM technology from the perspective of the communication subsystem, and point out that VM-based computing is very promising. While this dissertation has addressed several important issues at multiple levels of the system software stack, there are still many possible future research directions:

- **A general framework for virtualizing OS-bypass networks:** In this dissertation we discussed the virtualization of InfiniBand and iWARP/10GibE. However, the actual implementations are customized to Mellanox and Chelsio NICs. There are multiple such kinds of OS-bypass NICs popular in the community. While developing VMM-bypass drivers for those HCAs is mainly an implementation task, we find it especially important to design a general framework for virtualizing OS-bypass networks. It is important to abstract out the device-independent parts of the drivers for various popular OS-bypass HCAs, which will minimize the effort to virtualize a new OS-bypass HCA.

- **High performance virtualization for Non OS-bypass networks:** While OS-bypass networks dominate the HPC area, many enterprise environments, such as data centers, are reluctant to abandon TCP/IP networks and are using Ethernet family devices. Unfortunately, most Ethernet devices are traditional PCI devices and do not have OS-bypass capabilities, thus cannot be virtualized using VMM-bypass. In order to optimize the virtualization performance for PCI devices, Xen supports a scheme called PCI pass-through, which currently is a boot-time option and dedicates a PCI device to be exclusively used in a specific guest VM. The PCI pass-through has limitations; a PCI device must be statically assigned to a guest VM and cannot be adjusted according to the runtime workloads of the host VMs. As a result, it will be interesting to explore dynamic PCI pass-through techniques, which are able to dynamically assign a

PCI device to the guest VM which has the heaviest network I/O, while other VMs will access the network through the current split-device driver model. Such designs compliment VMM-bypass I/O, providing a complete solution to high performance network I/O in VM environments.

- **Study of machine-level checkpointing and process-level checkpointing:** Many VM-based environments support checkpoint/restart (CR). CR happens at the entire virtual machine level, which hides complexities compared with process-level CR solutions. Unfortunately, such simplicity is usually penalized by higher cost, such as larger amounts of memory content transferred to storage. However, in the context of high performance storage systems connecting through high speed interconnects, such added overhead can be minimal, which justifies system-level fault tolerance solutions through VMs. A systematic evaluation of VM-based CR framework compared with other process-level CR solutions will also be an important future direction.

- **Cluster management framework:** While performance is an important factor for VM environments, it is the benefits for management that make VM an attractive technology for HPC platforms. Most of the existing research efforts, however, focus on managing data centers with VMs. Though many of the techniques are common, there are special management issues on a VM-based environment for HPC. For example, we have discussed using customized OSes for applications. Customized OSes can potentially improve performance but also lead to challenges for management: how can we quickly find the necessary configuration needed for an application? How can we efficiently create an OS image based on such a configuration? Once the image is ready, how can we distribute and deploy it over multiple nodes as fast as possible? Another example is to study the failure patterns of clusters. We have mentioned that VM migration can be used for proactive fault tolerance. But how can we precisely decide when a physical computing node is about to fail? What kind of system information do we need to constantly monitor in order to predict failures? Can we do a better job with studies on the failure patterns of clusters? Performance profiling/monitoring is another interesting topic in this direction. Precise but lightweight performance monitoring can help system administrators to decide when and where to migrate a VM.

BIBLIOGRAPHY

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

[2] Argonne National Laboratory. http://www-unix.mcs.anl.gov/mpi/mpich/.

[3] Argonne National Laboratory. Zeptoos: The small linux for big computers. http://www-unix.mcs.anl.gov/zeptoos/.

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

[5] 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.

[6] P. N. Brown, R. D. Falgout, and J. E. Jones. Semicoarsening multigrid on distributed memory machines. *SIAM Journal on Scientific Computing*, 21:1823–1834, 2000.

[7] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems*, 15(4):412–447, 1997.

[8] L. Chai, A. Hartono, and D. K. Panda. Designing High Performance and Scalable MPI Intra-node Communication Support for Clusters. In *The IEEE International Conference on Cluster Computing (Cluster 2006)*, Barcelona, Spain, Sept. 2006.

[9] Chelsio Communications. http://www.chelsio.com/.

[10] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live Migration of Virtual Machines. In *Proceedings of 2nd Symposium on Networked Systems Design and Implementation*, 2005.

[11] D. Dalessandro, A. Devulapalli, and P. Wyckoff. Design and Implementation of the iWarp Protocol in Software. In *Proc. of Parallel and Distributed Computing and Systems (PDCS)*, Phoenix, AZ, November 2005.

[12] 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.

[13] FastOS: Forum to Address Scalable Technology for runtime and Operating Systems. http://www.cs.unm.edu/ fastos/.

[14] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and $\tilde{\text{M}}$. Williamson. Reconstructing I/O. Technical Report UCAM-CL-TR-596, University of Cambridge, UK, August 2004.

[15] 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.

[16] H. Tezuka and F. O'Carroll and 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] Steven M. Hand. Self-paging in the nemesis operating system. In *Operating Systems Design and Implementation, USENIX*, pages 73–86, 1999.

[18] W. Huang, J. Liu, M. Koop, B. Abali, and D. Panda. Nomad: Migrating OS-bypass Networks in Virtual Machines. In *Third International ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE'07)*, San Diego, California, 2007.

[19] Wei Huang, Matthew Koop, Qi Gao, and Dhabaleswar Panda. Virtual Machine Aware Communication Libraries for High Performance Computing. In *Proceedings of SC'07*, Reno, NV, November 2007.

[20] IETF IPoIB Workgroup. http://www.ietf.org/html.charters/ipoib-charter.html.

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

[22] Intel Corporation. Intel Cluster Toolkit 3.0 for Linux.

[23] Song Jiang and Xiaodong Zhang. LIRS: An Efficient Low Inter-Reference Recency Set Replacement Policy to Improve Buffer Cache Performance. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Marina Del Rey, CA, 2002.

[24] Hyun-Wook Jin, Sayantan Sur, Lei Chai, and D. K. Panda. Design and Performance Evaluation of LiMIC (Linux Kernel Module for MPI Intra-node Communication) on InfiniBand Cluster. In *International Conference on Parallel Processing*, Sept. 2005.

[25] Theodore Johnson and Dennis Shasha. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *VLDB '94: Proceedings of the 20th International Conference on Very Large Data Bases*, San Francisco, CA, USA, 1994.

[26] Orran Krieger, Marc Auslander, Bryan Rosenburg, Robert Wisniewski, Jimi Xenidis, Dilma Da Silva, Michal Ostrowski, Jonathan Appavoo, Maria Butrico, Mark Mergen, Amos Waterland, and Volkmar Uhlig. K42: Building a Complete Operating System. In *EuroSys 2006*, Leuven, Belgium, April 2006.

[27] LAMMPS Molecular Dynamics Simulator. http://lammps.sandia.gov/.

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

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

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

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

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

[33] Aravind Menon, Jose Renato Santos, Yoshio Turner, G. Janakiraman, and Willy Zwaenepoel. Diagnosing Performance Overheads in the Xen Virtual Machine Environment. In *Proceedings of the 1st ACM/USENIX Conference on Virtual Execution Environments (VEE'05)*, June 2005.

[34] Mark F. Mergen, Volkmar Uhlig, Orran Krieger, and Jimi Xenidis. Virtualization for High-Performance Computing. *SIGOPS Oper. Syst. Rev.*, 40(2):8–11, 2006.

[35] MOLAR: Modular Linux and Adaptive Runtime Support for High-end Computing Operating and Runtime Systems. http://forge-fre.ornl.gov/molar/.

[36] MPICH2 Homepage. http://www-unix.mcs.anl.gov/mpi/mpich2/.

[37] MVAPICH Project Website. http://nowlab.cse.ohio-state.edu/projects/mpi-iba/index.html.

[38] A. B. Nagarajan and F. Mueller. Proactive Fault Tolerance for HPC with Xen Virtualization. TR2007-1, North Carolina State University, Jan. 2007.

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

[40] Ripal Nathuji and Karsten Schwan. VirtualPower: coordinated power management in virtualized enterprise systems. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 265–278, New York, NY, USA, 2007. ACM.

[41] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The LRU-K Page Replacement Algorithm For Database Disk Buffering. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, 1993.

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

[43] OpenMPI. http://www.open-mpi.org/.

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

[45] A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary. HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers. http://www.netlib.org/benchmark/hpl/.

[46] James C. Phillips, Rosemary Braun, Wei Wang, James Gumbart, Emad Tajkhorshid, Elizabeth Villa, Christophe Chipot, Robert D. Skeel, Laxmikant Kale, and Klaus Schulten. Scalable molecular dynamics with NAMD. *Journal of Computational Chemistry*, 26:1781–1802, 2005.

[47] Portable Batch System (OpenPBS). http://www.openpbs.org/.

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

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

[50] Himanshu Raj and Karsten Schwan. High Performance and Scalable I/O Virtualization via Self-virtualized Devices. In *HPDC '07: Proceedings of the 16th international symposium on High performance distributed computing*, pages 179–188, New York, NY, USA, 2007. ACM.

[51] RDMA Consortium. Architectural Specifications for RDMA over TCP/IP. http://www.rdmaconsortium.org/.

[52] Daniele Paolo Scarpazza, Patrick Mullaney, Oreste Villa, Fabrizio Petrini, Vinod Tipparaju, and Jarek Nieplocha. Transparent System-level Migration of PGAS Applications using Xen on InfiniBand. In *IEEE International Conference on Cluster Computing (Cluster'07)*, Austin, TX, 2007.

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

[54] SLURM: A Highly Scalable Resource Manager. http://www.llnl.gov/linux/slurm/.

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

[56] SPEC CPU 2000 Benchmark. http://www.spec.org/.

[57] 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.

[58] University of Cambridge. Xen Interface Manual.

[59] User-Level Direct Access Transport APIs (uDAPL). uDAPL API Spec (Version 2.0).

[60] Geoffroy Vallee, Thomas Naughton, Hong Ong, and Stephen L. Scott. Checkpoint/Restart of Virtual Machines Based on Xen. In *High Availability and Performance Computing Workshop (HAPCW'06)*, Santa Fe, NM, USA, 2006.

[61] 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.

[62] 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.

[63] W. Huang, G. Santhanaraman, H.-W. Jin, and D. K. Panda. Design Alternatives and Performance Trade-offs for Implementing MPI-2 over InfiniBand. In *EuroPVM/MPI 05*, Sept. 2005.

[64] W. Huang, G. Santhanaraman, H.-W. Jin, and D. K. Panda. Scheduling of MPI-2 One Sided Operations over InfiniBand. In *CAC 05 (in conjunction with IPDPS 05)*, Apr. 2005.

[65] W. Huang, J. Liu, B. Abali and D. K. Panda. A Case for High Performance Computing with Virtual Machines. In *Proceedings of the 20th ACM International Conference on Supercomputing*, 2006.

[66] W. Jiang, J. Liu, H. -W. Jin, D. K. Panda, W. Gropp, and R. Thakur. High Performance MPI-2 One-Sided Communication over InfiniBand. In *CCGrid 04*, Apr. 2004.

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

[68] Jian Wang, Kwame-Lante Wright, and Kartik Gopalan. XenLoop: A Transparent High Performance Inter-VM Network Loopback. In *Proc. of International Symposium on High Performance Distributed Computing (HPDC)*, June 2008.

[69] MVAPICH Project Website. http://nowlab.cse.ohio-state.edu/projects/mpi-iba/index.html.

[70] 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.

[71] 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.

[72] P. Willmann, J. Shafer, D. Carr, A. Menon, S. Rixner, A. Cox, and W. Zwaenepoel. Concurrent Direct Network Access for Virtual Machine Monitors. In *The 13th International Symposium on High-Performance Computer Architecture (HPCA-13)*, Phoenix, AZ, Feb. 2007.

[73] Xen Wiki. http://wiki.xensource.com/xenwiki/xenbus.

[74] XenSource. http://www.xensource.com/.

[75] Weikuan Yu and Jeff Vetter. Xen-Based HPC: A Parallel IO Perspective. In *8th IEEE International Symposium on Cluster Computing and the Grid (CCGrid'08)*, Lyon, France, May 2008.

[76] Xiaolan Zhang, Suzanne Mcintosh, Pankaj Rohatgi, and John Linwood Griffin. XenSocket: A high-throughput interdomain transport for VMs. In *ACM/IFIP/USENIX 8th International Middleware Conference (Middleware'07)*, Newport Beach, CA, November 2007.