University of Cincinnati

Date: 11/8/2012

I, Robert Lancaster , hereby submit this original work as part of the requirements for the degree of Master of Science in Computer Engineering.

It is entitled:

Low Latency Networking in Virtualized Environments

Student's name: **Robert Lancaster**

This work and its defense approved by:

Committee chair: Philip Wilsey, PhD

Committee member: Fred Beyette, PhD

Committee member: Wen Ben Jone, PhD

3085

Low Latency Networking in Virtualized Environments

A thesis submitted to the

Division of Research and Advanced Studies
of the University of Cincinnati

in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in the School of Electric and Computing Systems
of the College of Engineering and Applied Sciences

October 31, 2010

by

Robert J. Lancaster
BS Computer Engineering, University of Cincinnati, 2010

Thesis Advisor and Committee Chair: Dr. Philip A. Wilsey

Abstract

Beowulf clusters are popular in the field of high performance computing (HPC). Customized operating systems have been used to achieve speedup in HPC by providing specific mechanisms to support the application and by eliminating OS jitter. Virtualized operating systems make it possible to run customized operating systems in a shared environment. The principle draw back to virtualized operating systems for HPC is the added I/O latency of virtualization.

Para-virtualized I/O when coupled with a lightweight protocol. Can serve to reduce and in many cases eliminate the latency gap between native network I/O and virtualized network I/O. This study finds the latency performance of para-virtualized Infiniband over Ethernet matches or exceeds the performance of TCP/IP native for messages over 128 bytes.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Beowulf clusters are a staple of high performance computing (HPC). Parts of many complex problems can be broken up and distributed across different machines on a network or to different physical cores on a single machine. Many parallel processing applications use a hybrid model that sends tasks to both a Beowulf cluster and the discreet cores on each machine in the cluster. A parallel application can be characterized as either fine grained or course grained. Course grained parallelism refers to distributing a large portion of a problem to each node in a cluster. Fine grained parallelism refers to each node doing small pieces of the problem. Course grained parallel applications have a tendency to be I/O bandwidth bound while fine grained parallel applications have a tendency to be I/O latency bound. There are a large number of fine grained HPC applications such as gasoline and weather simulations [1]. Fine grained parallel applications offer many challenges to distributed computing and are a driving factor behind this study. There are many adjustments that can be made to a system to improve its processing latency.

Application specific operating systems are sometimes developed to improve (or in some cases, enable) the runtime performance of computationally expensive applications [2,3]. Jefferson saw significant speed up while using the Time Warp Operating System for parallel discrete event simulation. [2]. Operating system jitter can also be reduced by using custom operating systems [4]. This reduction is achieved by removing pieces of the OS that are not needed for the specific application of the OS. The benefits of an application specific OS are attractive for HPC applications in a cluster environment. The chief disadvantage of these solutions is that they require that expensive parallel computer hardware be dedicated to a specific application.

Unfortunately, this is not feasible for environments where hardware is shared between a community of users running applications from different domains. Fortunately, virtualization may provide an opportunity to realize the benefits of application specific operating systems. More precisely, a hardware platform supporting virtualization can be configured with multiple guest operating system environments. Some of those guest environments can run traditional server or workstation based operating systems while others can support application specific operating systems. However, in order for this solution to be effective with respect to HPC the runtime overheads of virtualization need to be addressed. The principal challenge that needs to be better understood and alleviated is the message passing performance between virtual guest environments on separate hardware with in a Beowulf cluster. Due to the fine grained nature of most HPC applications low-latency communication is a necessity.

Virtualized environments have a negative effect on network and I/O latency that cannot be ignored. Latency can have an enormous effect on many HPC applications as detailed by the HPC Advisory Council in their white paper [1]. It is clear that HPC cannot be moved to a virtualized cluster without addressing the I/O latency issue. In virtualized environments there are existing mechanisms that can mitigate or eliminate this issue. The two canonical methods for reducing the cost of virtualized networking are PCI pass-through and para-virtualized network drivers. PCI pass-through consists of passing control of the network card directly to the virtual machine and results in native performance. PCI pass-through requires dedicated hardware. Using 10 Gigabit Ethernet, Infiniband, or Myrinet can cost an extra thousand dollars per node. For that price Infiniband hardware can achieve very low latencies on the order of 1.29 μs [5]. While slightly cheaper, 10 Gigabit Ethernet only shows latency numbers at 10.6 μs for zero length messages [6]. For many clusters this extra cost is not feasible. Para-virtualization uses a communication mechanism between the guest and the host operating system to share the host's network card with the guest. The communication between the guest and the host results in an increase in network latency. This increase is unavoidable. However, lightweight protocols can be used to close the gap between para-virtualized network I/O and native network I/O.

This thesis seeks to improve the network latency of a para-virtualized driver by replacing the standard TCP/IP protocol with the lighter weight Infiniband protocol [7]. The objective is to exploit the reduced processing overhead of the Infiniband protocol to close the gap between native network I/O and virtualized network I/O. This study uses KVM [8] as the virtual machine manager(VMM). The Infiniband over Ethernet

drivers from System Fabric Works are modified to use the `virtio` communication mechanism used in KVM. PCI pass-through is also studied as a possible method of guest-to-guest communication, but only to provide a comparative metric to complete the performance characterization of the spectrum of options that are possible in the experimental virtualized Beowulf Cluster used in this study.

The Infiniband protocol is found to perform well for messages above 128 Kilobytes. This is a result of the low processing overhead of the Infiniband protocol on a per-byte basis. Native TCP/IP speeds are difficult to achieve with small message sizes below 64 bytes. The reason behind this is the cost of interrupting the guest operating system from the host operating system. Also, The speed up achieved from lightweight protocols is highly dependent on the Ethernet hardware used. Many Ethernet drivers do not allow control over the coalescing of transmit and receive interrupts which can have a significant effect on latency. However, even with these limitations this thesis study shows lightweight protocols are effective in reducing latency in both para-virtualized drivers and in native network I/O.

1.1 Thesis Overview

The remainder of this thesis is organized as follows:

Chapter 2 presents an overview of background topics that will prepare the reader with a base for understanding the remainder of the document. It begins with a characterization of network latency and the effects it can have on HPC applications. The different virtualized networking options are also described in detail, which leads into a discussion of lightweight protocols and how there use results in reduced network latency.

Chapter 3 discusses several similar works that have influenced this study.

Chapter 4 Characterizes the high level approach taken to solving the issues of virtualized network latency. It details the reasoning behind the choice of Infiniband as the lightweight protocol used in this study and the use of KVM as the virtual machine manager. Chapter 4 also presents a conceptual design for the para-virtualized drivers written during this study.

Chapter 5 Provides a detailed description of the implementation of both the host and guest side para-virtualized drivers and their interaction with KVM and qemu. It also details the design of the tests using the linux trace tools.

Chapter 6 presents and analyzes the results of the tests described in the preceding chapters. The results

are presented in two sections. The first details the results of the ping-pong latency tests, and the second breaks down the software latencies acquired from the Linux trace tools.

Finally, Chapter 7 draws conclusions from the analysis sections and presents future work that may be done based on the work presented in this paper.

Chapter 2

Background

This section contains explanations of several topics that are pertinent to this study. First, is a summary of current latency studies and how they relate to HPC applications in virtualized environments. A description of currently available virtualized networking options is also included. This is followed by a discussion of the applications of lightweight protocols and reasoning behind the selection of Infiniband as the lightweight protocol used in this study.

2.1 Network Latency

Due to the recent microprocessor innovations concerning virtualization support (AMD-V and VT-d), significant virtualization performance overhead has been eliminated. Some research has shown that CPU intensive tasks with light I/O demands perform well within virtual environments [9]. In fact, the majority of studies on virtualized network I/O have focused on bandwidth since this is the chief concern for the use of virtualization in IT [10–13]. This has lead to a focus on improving bandwidth for virtualized I/O. The problem with this is that many bandwidth improving mechanisms such as throttling and the use of large packets often have a negative impact on latency.

In contrast to studies and improvements with network bandwidth in virtualized networks, achieving satisfactory message latency performance for I/O intensive tasks remains problematic [14]. Optimizing network performance, specifically latency, is of particular value to distributed applications that tend to frequently exchange small messages. Fortunately, within a local area virtualized Beowulf cluster the support

for the standard TCP/IP communication protocol may not be necessary. Thus, removing some of the software layers in the messaging layers may provide significant opportunities to optimize performance.

While network latency in virtualized networks is largely unstudied, an in-depth paper by Larson [6] provides some interesting insights on native (non-virtual) end-to-end latency in TCP/IP networks. Larsen used nano-second resolution timers to record the send and receive times of zero-length messages. In particular, Larson's results show an average send time latency of 3.75μs, with 2.6μs of that time is spent in software. On the receive side, Larsen measured a average receive time latency of 7.75μs, with 6.3μs of that time spent in software. This means that software processing is 62% of the transmit latency and 81% of the receive latency. The high processing cost of the TCP/IP can have serious impact on a virtualized system since there is the added time to send an interrupt from the host to the guest and from the guest kernel to the guest userspace. Since TCP/IP is computationally expensive a lightweight protocol may be more appropriate for virtualized network I/O.

2.2 Typical Network Interfaces in Virtual Machines

The networking interfaces presented to the virtual machine can be configured in three principle ways, namely: *full virtualization*, *para-virtualized*, or as *direct device access* (or *PCI pass-through*) devices [15]. In full virtualization, a fully emulated hardware device is presented to the virtual machine. This has the advantage that any guest system can run unmodified in the virtual machine using standard hardware based device drivers. The downside is the runtime emulation costs that the host hypervisor adds to each networking access. To address the overhead costs of full virtualization, many hypervisors define a *para-virtualized* device API for guest machines to use. This requires that the guest system be modified to include another network driver for the para-virtualized device. The advantage is substantially higher networking performance.

An alternative to shared virtualization or para-virtualization that provides high performance I/O interfaces to the guest environment is known as *direct device access* (otherwise known as *PCI pass-through*). This technique was adopted early on by Xen [16] and later added to KVM upon the development of IOMMU [17] technologies (Intel's VT-d [18]). This feature allows the host to assign the guest environment exclusive and direct access to system hardware (such as network interfaces, frame buffers, disk drives, *etc*) to improve performance. This allows the the guest to operate at near native speeds for services related

to the hardware involved. The obvious drawback is scalability and the fact that the host system (and other guest environments if multiple are resident) lose access to the hardware. Despite the logistical drawbacks of the configuration, the technology has been shown to perform extremely well [19].

2.3 Lightweight Protocols

Another possible lever to improve network latency performance is to use lightweight communication protocols in place of the standard TCP/IP or MPI protocols. Lightweight protocols are attractive to virtualization since they eliminate message processing overhead, which serves to reduce the overall latency of each message [7, 20–22]. A lightweight protocol can be used to help the guest reduce latency costs and potentially allow them to reach the native speeds of a heavier protocol such as TCP/IP. For example, protocols such as GAMMA have been shown to outperform native TCP/IP due to reduced processing overhead [20]. Other lightweight protocols include GAMMA, UC GAMMA, Myrinet, and Infiniband.

The Infiniband protocol is used by Infiniband hardware. Given the widespread use of Infiniband hardware this makes Infiniband a compelling lightweight protocol for use over Ethernet hardware [23]. The use of Infiniband over Ethernet would also make the migration of many HPC applications to the new system simple since many support Infiniband hardware [1,23–25]. Mellanox also states in their white paper that this also simplifies the software stack in Linux and Windows since open source implementations of Infiniband over Ethernet already exist [23].

In this work, we contrast the performance of TCP/IP and Infiniband for communication between in various configurations of native and virtualized machines. The Infiniband protocol was chosen for study due to its widespread use and the existence of an open source Infiniband over Ethernet (IBoE) Linux kernel driver from System Fabric Works [23]. The driver completely skips IP routing and simply pastes the Infiniband protocol on top of the Ethernet header frame. Since most HPC applications operate on their own subnet this should not present a problem for widespread use. In addition, the driver is implemented so that the NIC retains the capability to transmit messages using either protocol (Infiniband or TCP/IP). Thus, there is no sacrifice in connectivity for remote management and configuration. The Infiniband protocol also has some useful features such as atomic reads and writes and RDMA. Having these features built into a protocol can be very useful in HPC configurations. Lastly, there is no loss of functionality from TCP/IP since Infiniband

has both Connection based and connection-less message types in the protocol.

Chapter 3

Related Work

There are no shortage of papers and studies that characterize the network latency problem. However, many of the papers only study the problem from a network hardware standpoint [1, 11, 25–27]. The network hardware based research focuses on the latency performance of various types of network hardware such as Infiniband, Myrinet, and 10G Ethernet. Even fewer studies focus on latency in virtualized environments [10, 12,14,24,28,29]. The studies on network virtualization focus largely on bandwidth and only mention latency briefly if at all. The studies that do mention virtualized network latency compare either the performance of different virtual machine managers or emulated drivers to para-virtualized drivers and fail to present the full virtualized latency picture. This chapter will focus on Larson's end-to-end network latency paper and the HPC Advisory Council's white paper on the effects of network latency on various HPC applications [1, 6]. The data from these two sources is combined with the results of other papers in the area to give an understanding of where this research fits within the field of HPC.

Larson's paper on end-to-end latency on one gigabit and ten gigabit Ethernet cards presents a breakdown the latencies incurred at every step along the message path [6]. Table 3.1 breaks the latencies into their software and hardware components. The most revealing numbers, in relationship to this study, have to do with the time each packet spends in software on both the transmit and receive stacks in the OS. In particular,

Table 3.1: Larson Data Software/Hardware Latency Breakdown

	Software Latency (ns)	Hardware Latency (ns)
Send:	2999	1160
Receive:	6377	1370

Larson found that 62% of TCP/IP transmit latency and 81% of receive latency resides in software. On a TCP/IP transmit 2.6μs of latency occurs on a zero length message before the Ethernet cards driver is even called. Upon receiving a zero length TCP/IP message 3.8μs of processing happens after the driver code finishes. With the minimum one way latency being 14μs for a zero length message. With latencies this low the time spent in software is significant. Larson's paper is responsible for showing definitively that these latencies exist. Unfortunately, Larson's paper does not run tests at differing packet sizes and while many HPC applications send small packet sizes none of them send zero length messages. The study in this thesis presents similar measurements at varying packet sizes and shows that the software processing latency overhead does increase substantially as the packet size increases.

The HPC Advisory Council paper compares the performance of 9 applications using gigabit Ethernet, 10 gigabit Ethernet, and Infiniband network interconnects [1]. The study concludes that, for applications that are driven by Input/Output operations per second, using Infiniband hardware can show significant performance increases and also result is power savings due to reduced processing time. The report describes that latency is of great importance to many HPC applications and it also reports bandwidth and latency tests for all three of the cards tested. However, the latency tests are for payloads up to 64 bytes, which is still a very small message.

There are two papers on virtualized network I/O that relate closely to the work presented in this document. The first being Liu's paper on VMM-Bypass I/O which shows that it is possible to achieve native speeds on Infiniband hardware from an Xen guest. This study is particularly relevant since Liu is also looking at virtualized I/O from a latency standpoint [24]. Liu approached the problem by using Infiniband hardware and used the userspace write functionality of the Infiniband hardware to completely bypass the host operating system for reads and writes. By bypassing the host operating system speeds close to that of native infiniband were achived. Using a send/recieve ping-pong test a minimum latency of 6.6 μs was shown. The research in this paper presents a solution that does not require specialized hardware. Since the majority of gigabit Ethernet hardware does not present userspace access functionality bypassing the host OS is not a option in this study.

The second paper by Xia, presents a custom para-virtualized networking solution using overlays [29]. Xia's work also focuses on using standard Ethernet network hardware. The paper presents a para-virtualized

Ethernet driver written for the Palacios VMM. However, even with the efficient approach presented in the paper the latency of the para-virtualized driver is still two times that of the native gigabit Ethernet driver. The research presented in this paper uses a similar approach to Xia's para-virtualized driver, but goes further by using a light weight protocol in the place of TCP/IP.

The HPC Advisory Council paper and the Larson paper combined with the papers comparing emulated and para-virtualized network I/O all lead to the study in this thesis. The HPC Advisory Council shows that higher latency causes HPC applications to suffer. So the increased latencies shown by emulated and para-virtualized TCP/IP are not acceptable for use in a virtualized HPC cluster. Larson shows that a significant portion of network latency occurs within software. The aim of this study is to bring para-virtualization and light weight messaging together to bring native TCP/IP latencies within reach of para-virtualized network I/O.

Chapter 4

Overview of the Approach

This chapter covers the evolution of this study from the desire for low latency virtualized communication to the design and implementation of the pieces in this study. It starts with the search for the lightweight protocol used in this study. The choice of KVM for the virtual machine manager is explained and overviews the paradigm behind the design of the para-virtualized Infiniband drivers.

Solutions for virtualized guest-to-guest communication began with a search to determine which lightweight protocol was to be used. It is well known that para-virtualized TCP/IP has a very high latency cost. Therefore, in order to achieve low latency network communication through para-virtualization we needed the help of a protocol with a low software latency. Several protocols were reviewed. These protocols included GAMMA [20], UC GAMMA, Myrinet [30], and Infiniband [7]. GAMMA and its UC variant were promising, however they require the modification of the Ethernet driver. This is a negative since we want to change the host kernel as little as possible and we want our design to be easily applied to any system. Requiring the modification of the Ethernet driver for any new hardware would be a barrier to the use of this driver. So the Myrinet and Infiniband protocols are the leading candidates. Between the two, Infiniband was chosen because its use is more widespread. The widespread use of Infiniband ensures that there are already code bases that support the use of the Infiniband protocol this further reduces the barrier to the use of the pare-virtualized code. Furthermore, an implementation of Infiniband over Ethernet (IBoE) already exists in source form. This Open Source implementation was written by System Fabric Works and distributed on their website. Since the native drivers for Infiniband were already available and showed good latency char-

acteristics compared to native TCP/IP we could shift our focus to building a low latency para-virtualized version of this implementation.

There are several reasons behind the choice of KVM for the virtual machine manager used in this study. The two main open source VMMs are KVM [8] and XEN [15]. At the time this study began there was more support in the Linux community for KVM. KVM was already an active part of the Linux kernel and was well documented. XEN while at the start of this study was a part of the Linux kernel Its use was not widespread and had poor community support. Using KVM also meant that it would be significantly easier to port the Linux IBoE driver to use the virtio libraries since it was already within the Linux kernel.

With the inclusion of the System Fabric Works IBoE driver in the kernel the construction of the para-virtualized driver was simplified since we did not have to build our implementation from the ground up. The modularity of the IBoE driver allowed for the replacement of the network driver code with a para-virtualized version of the code. The IBoE driver with the para-virtualized network driver is run on the guest machine. A host driver was then designed for the host machine to intercept Infiniband packets from the hosts NIC and send them up to the guest driver using the kernels vhost virtio implementation. An overview of the para-virtualized IBoE driver can be seen in Figure 4.1. The configuration and creation of virtual guests in KVM is managed by the QEMU application. This also had to be modified to support IBoE and configure the vhost-ib driver in the host kernel.

The virtual machine networking configurations examined in this study are (i) para-virtualized TCP/IP, (ii) para-virtualized IBoE, (iii) TCP/IP PCI pass-through, and (iv) IBoE PCI pass-through. Native TCP/IP, native IBoE were also analyzed to give a baseline for the analysis of the costs of virtualized network I/O.

In order to compare the latencies of these various configurations a framework was needed to provide a consistent structure for measuring the latency in each configuration independent of the protocol being used. A ping-pong framework developed by William Magato in testing UC GAMMA was adapted to work with the IBoE driver. The operation of this framework can be seen in Figure 4.2. The operation of the ping-pong application is a 4 step process that is detailed below:

1. Ping message is sent to server.

2. Server responds with an ACK to the client.

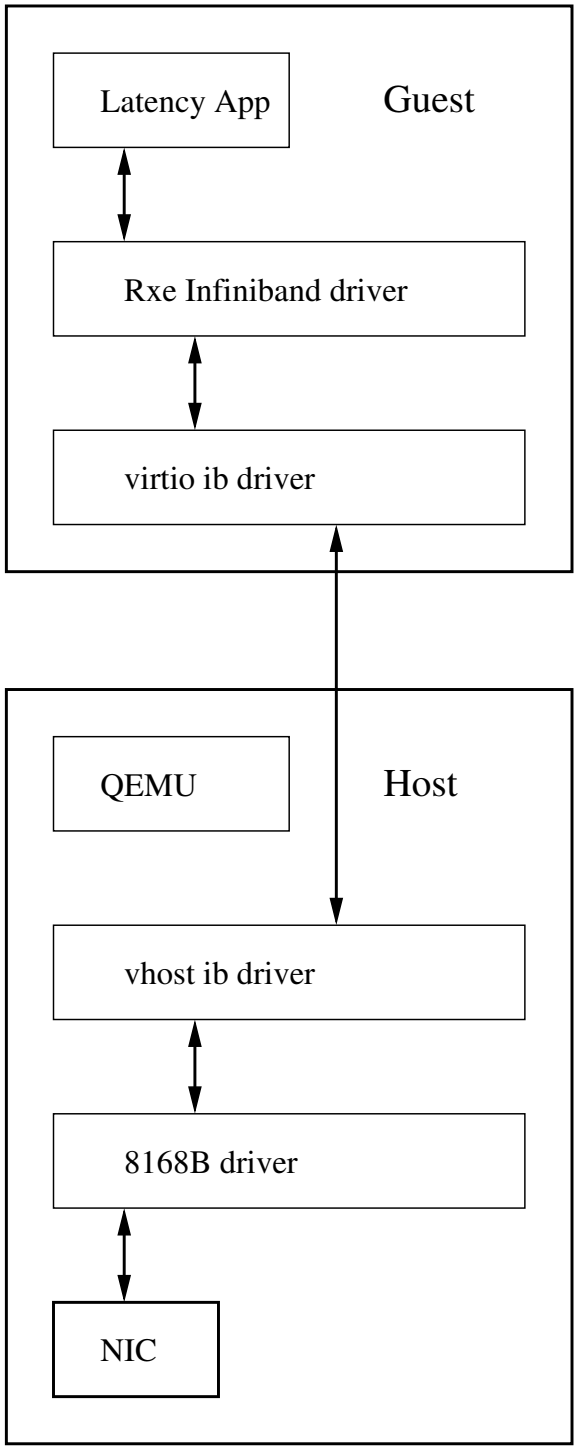

Figure 4.1: Virtio Infiniband implementation graphic

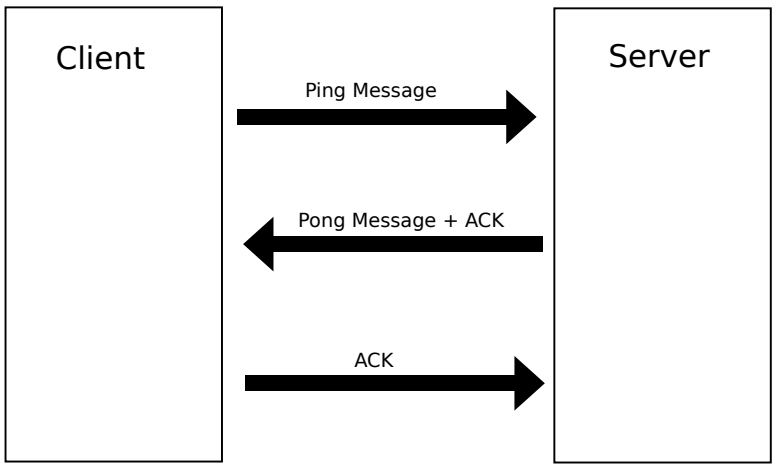

Figure 4.2: Latency Measuring application

3. Server sends the Pong message to the client.

4. Client sends ACK to Server.

Aside from overall delay that is measured by the ping-pong framework we are also interested in a fine-grained trace of the kernel functions involved in each configuration. The kernel trace tools were used to provide us with the information about how much time is spent in each function as the packet traverses the software stack in each configuration. The Linux kernel trace tools are perfect for this since they include a graph trace functionality that measures the time spent in each function called by a specified root function or functions. The graph trace then writes them to a Unix pipe as the kernel is run. The details of the use and analysis of this data are included later in chapter 5.

Chapter 5

Implementation Details

This chapter describes the details and challenges of implementing the para-virtualized IBoE driver and the methods used in the testing and analysis of the para-virtualized driver. The chapter is divided into three sections. The para-virtualized Infiniband driver is presented in two sections. Section 5.1 provides details on the host-side driver (vhost-ib) and section 5.2 provides details on the guest-side driver (virtio-ib). Section 5.3 describes the testing methods and implementation.

In the design of the para-virtualized IBoE implementation the first hurdle is where to put the processing of the Infiniband stack. This is essentially where to split the host driver from the guest driver. The nature of the Infiniband protocol gives userspace access to complete queues and send queues in the kernel. The goal is to have as few exits from the guest as possible so this access must be in the guest so there are not calls into the host to initialize the Infiniband queues on a connection basis. Once the queue processing is moved into the guest driver the next place for communication with the host is when the IBoE driver communicates with the network card. Choosing this point in the guest as the interface to the host driver minimizes the guest exits to the host and it simplifies the guest side implementation. Instead of making significant changes to the IBoE driver only the IBoE network module needs to be replaced with the para-virtualized version. This makes the vhost-ib driver a simple interface with the host network card that sends and receives IBoE packets to and from the proper guest based on the Infiniband global address.

5.1 Host Side Implementation

In Linux, host side virtio drivers come in two types the first most common type is through the userspace qemu application which is used to configure and launch KVM. These drivers are threads of the qemu process that communicate with the kernel drivers for the shared device and connect to the KVM process through virtio to relay messages between the devices. The other type of host side driver is implemented in the kernel using the vhost kernel module. The vhost kernel module is an implementation of virtio built into the Linux kernel. By locating the virtio communication in the kernel this allows the host kernel to communicate directly with the guest kernel driver and cuts out the host userspace application. Qemu is still used to configure the vhost driver but the actual exchange of data between the host and guest is done without involving a host userspace driver. The vhost-ib driver uses the vhost kernel module. By using the vhost method the path from the guest to the network is reduced by one copy on both the send and receive side. This is accomplished by the eliminating the involvement of qemu in the host side communication with the guest side driver.

The vhost-ib module is tied to the vhost module through qemu. To setup the kernelspace driver a new PCI hardware device driver is added to qemu. The qemu hardware driver is responsible for initializing, configuring, starting, and closing the vhost-ib module. These operations are done through the character device interfaces to the vhost and vhost-ib modules. A new PCI device type entry is added to the `virtio_info` list in `hw/virio-pci.c` within the qemu development tree. The virtio device entry can be seen in figure 5.1. This entry facilitates the initialization of the host driver when the guest registers a virtio-ib device with the virtual PCI bus. The `virtio_ib_init_pci`, `virtio_ib_exit_pci`, and `virtio_pci_reset` functions are implemented in the qemu virtio-ib driver. The vhost and vhost-ib modules are controlled by these functions. The `virtio_ib_init_pci` function is responsible for opening the character interfaces to the host kernel modules and registering the configuration functions with the vitio-pci userspace driver. The exit and reset functions perform the necessary clean up functions to return the driver to the required state.

Qemu arbitrates the virtio configuration process between the host and guest. Upon loading, the virtio drivers have four configuration operations reading and writing feature bits, reading and writing configuration space, reading and writing status bits, and the device reset [31]. All of these calls are passed through the qemu virtio driver. The configuration works in the following process:

1. The guest registers a vhost-ib device with virtual PCI bus.

```
{
    .qdev.name  = "virtio-ib-pci",
    .qdev.alias = "virtio-ib",
    .qdev.size  = sizeof(VirtIOPCIProxy),
    .init       = virtio_ib_init_pci,
    .exit       = virtio_ib_exit_pci,
    .vendor_id  = PCI_VENDOR_ID_REDHAT_QUMRANET,
    .device_id  = PCI_DEVICE_ID_VIRTIO_IB,
    .revision   = VIRTIO_PCI_ABI_VERSION,
    .class_id   = PCI_CLASS_COMMUNICATION_OTHER,
    .qdev.props = (Property[]) {
        DEFINE_PROP_BIT("ioeventfd", VirtIOPCIProxy, flags,
                        VIRTIO_PCI_FLAG_USE_IOEVENTFD_BIT, false),
        DEFINE_PROP_UINT32("vectors", VirtIOPCIProxy, nvectors, 2),
        DEFINE_VIRTIO_COMMON_FEATURES(VirtIOPCIProxy, host_features),
        DEFINE_PROP_END_OF_LIST(),
    },
    .qdev.reset = virtio_pci_reset,
}
```

Figure 5.1: PCI device entry in the qemu PCI device list

2. Qemu calls the virtio-ib init function which sets the virtio feature bits using an ioctl call to the vhost-ib module.

3. The guest reads these feature bits through the qemu `get_features` function and flips the bits of the features it does not support.

4. Once the guest probe is finished the VIRIO_CONFIG_S_DRIVER_OK bit is set to notify the host that it is ready.

5. The host then reads the features back through qemu `set_features` function.

6. The host then starts the vhost-ib module through an ioctl call and notifies the guest of a configuration change.

7. The guest then reads the configuration through the `get_config` call. Specifically, this gets the hardware address of the Ethernet card. So the packet can be initialized in the guest.

8. The Driver is now configured and the vhost-ib in the host driver and virtio-ib driver in the guest are connected via virtio.

After this process the qemu application does not participate in any futher IBoE operations until the virtual PCI device is removed or reset from the guest. The vhost-ib kernel module now connects the virtio-ib guest driver to the network card. Before moving on to the vhost-ib driver it is important to note how the guest is instructed to setup an IBoE device on the PCI bus. KVM guests are started via running the qemu command line application. Below is an example of a qemu command line call to start a virtual guest:

```
qemu-system-x86_64 -enable-kvm -cpu kvm64 -m 512 -vnc :1 -net nic,model=virtio
   -net tap,ifname=tap0,script=no -ib eth2 -drive file=linux.img,if=virtio
```

The `-ib` option triggers the loading of the virtio-ib device on the guest and connects the guest to interface `eth2` on the host. In order for qemu to recognize this option it must be defined in the `qemu-options.hx` file and the processing must be added to the argument parsing loop in the qemu main function. For virtualized IBoE only the desired interface name is need as an argument and the virtio-ib device is then added to the guest device list. The vhost-ib kernel module is where the remaining porting of the host side IBoE implementation resides.

The function of the vhost-ib module is to connect the virtual guest with the host Ethernet card. In the module init function the vhost-ib module registers the IBoE packet type with the network stack. This directs all received IBoE packets to the vhost-ib `rxe_net_rcv` packet handler function. Once the host has an IBoE packet the first hurdle arises from the fact that this packet needs to be passed to the guest in userspace. Packet handlers are run in the hard_irq context. While in the hard_irq context a function cannot sleep since it is interrupting all other processes on the current CPU. Copies to userspace memory may require a sleep so they cannot be called from the hard_irq context. This forces the vhost-ib driver to use a bottom half to finish processing each IBoE packet. There are three options for bottom halves in the kernel tasklets, kernel threads, and workqueues. A tasklet cannot be used in this situation since it still runs in the hard_irq context. A workqueue is an API around kernel threads for the purpose of implementing bottom halves. Nothing outside of the workqueue API is needed so workqueues are used to implement the bottom half of the vhost-ib receive. Workqueues are allowed to sleep so they serve to facilitate userspace copies to the guest.

The workqueue runs the bottom half function as a kernel thread. In the vhost-ib module the workqueue is initialized with the `HIGH_PRI` and `WQ_UNBOUND` options which allow it to run at the highest priority and on

any available processor. Once within the kernel thread the kernel needs to switch to the virtual memory space that the receive virtual ring buffer resides in. The kernel has `use_mm`, and `unuse_mm` functions to facilitate this. The `vhost_virtqueue struct` holds this information and is set during the vhost-ib initialization. Once in the same virtual memory space as the virtual ring buffer the host copies the queue descriptor from the guest to get the userspace address of the guest receive `sk_buff`. The `sk_buff` from the network card is then copied to the userspace address in the descriptor. After the copy completes the descriptor is added to the guest used queue and an interrupt is sent to the guest via the `vhost_add_used_and_signal` function. The `sk_buff` is freed on the host side and the workqueue function returns. This is a complete receive on the host side of the para-virtualized IBoE implementation.

The vhost-ib module is also responsible for handling IBoE sends from the guest. The send process starts from the guest send interrupt handler. The send handler is run in the soft_irq context so it is safe to access userspace memory from it. It is also already in the right virtual memory space since it is called from the guest. The send handler loops on the send virtqueue until no send packets remain in the queue. Each time through the loop the next descriptor is copied from the guest and the `sk_buff` pointed to by the descriptor is copied from the guest into the host. The host then calls the Ethernet cards `ndo_start_xmit` function on the `sk_buff` which sends it do the network device. The host then adds the descriptor to the guest's used queue with the `vhost_add_used` function. Once the descriptor is added to the used queue the host side send is complete.

The host side driver is essentially a thin layer that connects the guest to the host Ethernet card. However, The need for the bottom half workqueue on receives, calls to the guest, and copies to and from the guest all add to the total communication latency. These added latencies are unavoidable without structural changes in the Linux kernel. The exact latency contribution of each part is discussed in chapter 6.

5.2 Guest Side Implementation

This section details the design and implantation of the virtio-ib guest side driver. The driver is implemented as a virtio device driver and is registered with the virtio module when the kernel is loaded. The probe function completes the initialization exchange that was described in section 5.1. Also, the virtio-ib driver adds itself as a network device interface to the IBoE rxe module. The virtio-ib module implements the

functions required by the rxe driver. There are three functions of interest in the virtio-ib module `send`, `init_packet`, and `ib_receive_done`. These functions are discussed in depth in this section.

The `ib_receive_done` is implemented as an interrupt handler that handles the receive interrupt from the host. Similar to the receive in the host this function is run in the hard_irq context. While this function does not make calls to userspace it still requires a bottom half. The reason behind this is it takes too much CPU time to complete the house keeping required to process the receive data and will cause the kernel to throw a timeout error. Since the receive processing code does not sleep a kernel tasklet is the best option for the receive bottom half. Tasklets have a lower latency than a workqueue since they are scheduled as an interrupt. The interrupt handler schedules a high priority tasklet via the `tasklet_hi_schedule` function. The tasklet then processes the receive virtqueue until no new packets remain. `Sk_buffs` are pulled off of the virtqueue with the `virtqueue_get_buf` function. Each buffer is then sent to the rxe driver through the `rxe_rcv` call. Then the next receive buffer is processed. Once no new buffers remain the tasklet returns. Once the `rxe_rcv` function is called the packet is in the IBoE rxe driver code and the stack is the same as the native IBoE driver at this point.

When the IBoE rxe driver sends a packet it first calls the `init_packet` fuction. This function allocates a `sk_buff` and copies in the Ethernet header source and destination information. The `sk_buff` is then returned to the IBoE rxe driver. The IBoE rxe driver then fills the `sk_buff` with the IBoE payload and calls the `send` function in the virtio-ib driver. The send function first frees all old `sk_buffs` on the used queue. The virtio-ib send function then converts the `sk_buff` to a scatter gather array which is then added to the send virtqueue via the `virtqueue_add_buf` function. An interrupt is sent to the host with the `virtqueue_kick` function call. After sending the interrupt to the host the send function returns.

Considerable effort was made to make the virtio-ib send and receive code as small as possible, but still resulted in the need for a tasklet to process the bottom half of the receive this tasklet results in added latency to the IBoE receive stack. The efficiency of the tasklet is improved by processing any packets that arrive while processing the first packet to avoid the added latency of scheduling the tasklet again. The latnecy added by the tasklet processing is discussed in chapter 6.

5.3 Testing Implementation

As stated in Chapter 4 the testing and analysis of the various virtualized network I/O solutions is broken into two parts. The first part is ping-pong test which calculates an overall latency for each method. The second is a trace of of software latency for each method. This section details the settings for both TCP/IP and IBoE, the hardware used in the tests, and the methodology behind the trace tests.
The hardware configuration of the two machines used in this study is:

- Intel Core i7 920 @ 2.67GHz
- ASRock X58 Extreme Motherboard
- Debian Host Operating System
- Debian Guest Operating System
- RealTek 8168B PCIe Gigabit Ethernet Card
- RealTek 8139C PCI 10-Mbit Ethernet Card

The details for the RealTek Gigabit Ethernet card are listed below:

- Ethernet controller: Realtek Semiconductor Co., Ltd. RTL8111/8168B PCI Express Gigabit Ethernet controller (rev 03)
- Subsystem: ASRock Incorporation Motherboard (one of many)
- Control: I/O+ Mem+ BusMaster+ SpecCycle- MemWINV- VGASnoop- ParErr- Stepping- SERR- FastB2B- DisINTx+
- Latency: 0, Cache Line Size: 256 bytes
- Interrupt: pin A routed to IRQ 46
- Region 0: I/O ports at b800 [size=256]
- Region 2: Memory at cffff000 (64-bit, prefetchable) [size=4K]

- Region 4: Memory at cfff8000 (64-bit, prefetchable) [size=16K]
- Kernel driver in use: r8169

The settings used for the TCP/IP tests are shown in Figure 5.2. These settings were applied to both native and virtualized tests using TCP/IP. The `wmem_max` and `rmem_max` settings set the maximum write and receive window sizes. They had to be adjusted up from the default of 110K to allow the larger payloads to be transmitted and received in direct succession. While 110K is smaller than the largest payload tested the TCP stack will automatically start throttling at a threshold of 16k for TCP/IP writes. In order to prevent this throttling the `wmem_max` and `rmem_max` values are adjusted up to 12MB. This is much higher than needed for the latency tests, but this allows the for greater bandwidth. The TCP throttling threshold is controlled by `tcp_rmem` and `tcp_wmem` these are adjusted to be safely above the maximum test payload size of 64k. The `tcp_low_latency` setting allows the TCP/IP stack to favor latency over throughput since latency is our primary goal this is enabled. `tcp_no_metrics_save` is turned off to prevent the saving of state from connection to connection for our analysis. Window scaling is required for sizes that cannot be addressed with a 16 bit value so that is turned on as well. Network devices allow the setting of the maximum receive queue length. This is done with the `netdev_max_backlog` option. The `netdev_max_backlog` option will not affect latency very much but it does improve bandwidth performance without negatively affecting latency.

```
net.core.wmem_max=12582912
net.core.rmem_max=12582912

net.ipv4.tcp_rmem=10240 87380 12582912
net.ipv4.tcp_wmem=10240 87380 12582912

net.ipv4.tcp_window_scaling=1
net.ipv4.tcp_no_metrics_save=1

net.core.netdev_max_backlog=5000
net.ipv4.tcp_low_latency=1
```

Figure 5.2: TCP/IP settings in sysctl.conf

The IBoE driver also has several dynamic settings the were set for each test using IBoE. These settings are shown in Figure 5.3. Since the maximum transfer unit options for Infiniband are powers of 2

the closest MTU to that of the TCP/IP default of 1500 is 1024. This is set though `default_mtu`. The `max_pkt_per_ack` option is the number of packets IBoE is allowed to ack at once. Since the max message size in this study is 64KB and the MTU is 1024 this would result in the maximum number of packets per message in our study is 64. The maximum number of `sk_buffs` per queue is set to 800. This number does not have much effect on the tests in this study since it would primarily help with bandwidth. The `nsec_per_kbyte` and `nsec_per_packet` settings allow the IBoE driver to throttle the send packets to prevent overflowing the Ethernet hardware send buffers. Tests were done at each packet size and the lowest values that did not result in packet loss were chosen. The options appended with fast are triggers in the IBoE code that allow it to chose between processing steps in a kernel thread or in line. When these options are set to '2' the processing is done in line and when set to '0' the processing is done in a kernel thread. In the kernel trace test all of these values were set to '2' to ensure all processing was seen in the trace. It was determined that having the response processed in a kernel thread resulted in a lowered overall latency. So `fast_resp` is set to 0 for the ping-pong latency tests.

```
default_mtu=1024
max_pkt_per_ack=64
max_skb_per_qp=800
nsec_per_kbyte=700
nsec_per_packet=200
fast_arb=2
fast_req=2
fast_resp=0
fast_comp=2
```

Figure 5.3: IBoE settings set using sysfs

The details of the ping-pong test are described in Chapter 4 The setup and implementation for these tests is straight forward, aside from the settings discussed above there are no extra configuration steps. The software latency tests using the Linux trace tools are more complex. The goal of using the trace tools is to trace each packet through the kernel from start to finish. The ftrace functionality in the kernel is controlled using sysfs. There are several types of trace functionality available in the Linux kernel. The tests in this study use the `function_graph` tracer. This tracer records the start and finish time of a function and does the same for each function that function calls. The tracer can be filtered by function. The key to using the `function_graph` tracer is finding the earliest function exclusive to each network protocol to set the filter

for the tracer.

The root function for a native IBoE send is `rxe_post_send` this is the function called from userspace to initiate a send. All of the functions used in sending an IBoE packet extend from this function. Sends have the advantage of being called directly from userspace which gives an easy entry point to trace from. Also, there are no tasklets or workqueues that are scheduled on a native IBoE send allowing a packet to be traced all the way through the Ethernet driver call by filtering on one function. Tracing on the receive is made more difficult since it is called from an interrupt belonging to the network driver so choosing the root function too early in the network stack results in non IBoE data entering the trace. The first function exclusive the IBoE is the packet handler that the driver registers for the IBoE packet type, `rxe_net_rcv`. This function can be traced all the way to where the kernel frees the `sk_buff` after putting the packet on the complete queue. To successfully trace a packet for native IBoE only two functions need to be added to the trace filter. Para-virtualized IBoE has several differences due to the traces being split between the host and the guest machines.

A para-virtualized IBoE send originates in the guest at the same point as the native IBoE send, `rxe_post_send`. The message then travels through the guest and is copied to the host in the `handle_tx_kick` function. Since there are no workqueues or tasklets in between the `handle_tx_kick` and the network driver no other functions are needed for the send trace filter. However, `rxe_post_send` is added to the trace filter on the guest machine, and `handle_tx_kick` is added to the send trace filter on the host machine. The receive trace path has a workqueue and a tasklet in the path so there are four functions in the trace filter for the para-virtualized IBoE receive. The receive starts in the host similar to the native IBoE receive the trace is started in the IBoE packet handler `handle_rx`. The `handle_rx` function schedules a workqueue to handle the notification of the guest. The function run by the workqueue is the `rcv_wk_func`. This function handles the packet up to the guest hand off. The guest picks up the packet in the `ib_recv_done` function. This function schedules a tasklet bottom half with the `ib_recv_tasklet_fn` function which handles the remainder of the receive. The trace filter on the host side para-virtualized receive contains the `handle_rx` and the `rcv_wk_func`. The guest side trace filter contains the `ib_recv_done` and `ib_recv_tasklet_fn` functions. When running the trace tests the host and the guest keep separate trace logs and both logs are parsed together after the tests are run.

A similar method is used to get the software latencies for TCP/IP native. When processing a TCP/IP native receive the trace is started from the receive interrupt on the realtek card. Since protocols like ARP and ICMP will also show up at this level when doing the analysis the numbers are measured after the nfhook call passes processing to the IP layer. This is also where the IBoE processing analysis is started so the two measurements are comparable. The send trace is started at the `tcp_sendmsg` fucntion. This function is called from userspace and is traced to the network driver transmit function. With the receive interrupt from the realtek card and the `tcp_sendmsg` function the full software path of a TCP/IP messages can be traced by the Linux trace tools.

Traces for PCI pass-through are done in the exact same way as the native trace. The only difference is that the traces are run of the virtualized guest instead of the native OS, which does not participate in PCI pass-through messaging. Several difficulties were encountered while using the Linux trace tools. The first of which is the trace ring buffer limitation. When tracing the network stack many trace events are generated and the trace ring buffer size must be set at the start of the trace using the `buffer_size_kb` sysfs file. The default is 4KB for many traces this number had to be increased to four times that value in order to catch all of the function traces. Another method to avoid overflowing the ftrace ring buffer is to put sufficient delay between message exchanges. Another issue with ftrace is that it requires extensive knowledge of the code that is being traced since as explained above the correct trace start points need to be identified to ensure the correct path is being traced. This proved very dificult when attempting to trace the Mellanox Infiniband cards used in this study. As a result the traces for this hardware did not prove useful.

Chapter 6

Performance Results

The experimental results are presented in 5 separate sections. The first section 6.1 discusses the results of the round trip latency measurements from the ping-pong application. The second section 6.2, dissects the results of running the Linux trace tools to find the exact software latencies of each protocol in software. Section 6.3 details buffer allocation changes made to improve the software latency of the para-virtualized IBoE driver. Section 6.4 compares the IBoE ping-pong results from the Intel and Realtek Ethernet hardware. The last section 6.5, presents the results of the bandwidth tests for each configuration.

6.1 Latency Measurements

The results of the non-virtualized and virtualized machine tests are shown in Figures 6.1 and 6.2. Figure 6.1 graphs the latency of each test in μs and Figure 6.2 graphs each test relative to native TCP/IP. Native Infiniband over Ethernet tracks closely to native TCP/IP and significantly outperforms it at message sizes of 4k, 8k, and 32k. It is interesting that the Infiniband protocol coupled with the `RealTek` hardware is so sensitive to message size. The native TCP/IP numbers are the baseline to which other methods will be compared. This comparison to native TCP/IP is made since the selected virtualized protocol needs to perform at least as well as native TCP/IP in order to allow the speed up from a custom operating system to be realized.

Figure 6.1 shows the results of the para-virtualized drivers as well. The virtualized Infiniband protocol consistently out performs virtualized TCP/IP. Para-virtualized Infiniband can be as much as 81% faster (4k

message size) than para-virtualized TCP/IP and is never below 14% faster (65k message size). However, it does not succeed in matching native TCP/IP for small messages. Para-virtualized Infiniband is 112% slower than native TCP/IP for small messages up to 32 bytes. This is due to the time taken to communicate with the guest to notify it of the arrival of new messages. Thus, while a para-virtualized version of Infiniband performs much better than para-virtualized TCP/IP it still does not perform well with small messages between 4k and 32k. The para-virtualized Infiniband can be up to 56% faster than native TCP/IP for a 4k message. The comparison between para-virtualized Infiniband and native TCP/IP can be seen in Figure 6.1 as well.

PCI pass-through for TCP/IP and for IBoE results can also be seen in Figure 6.1. When using PCI

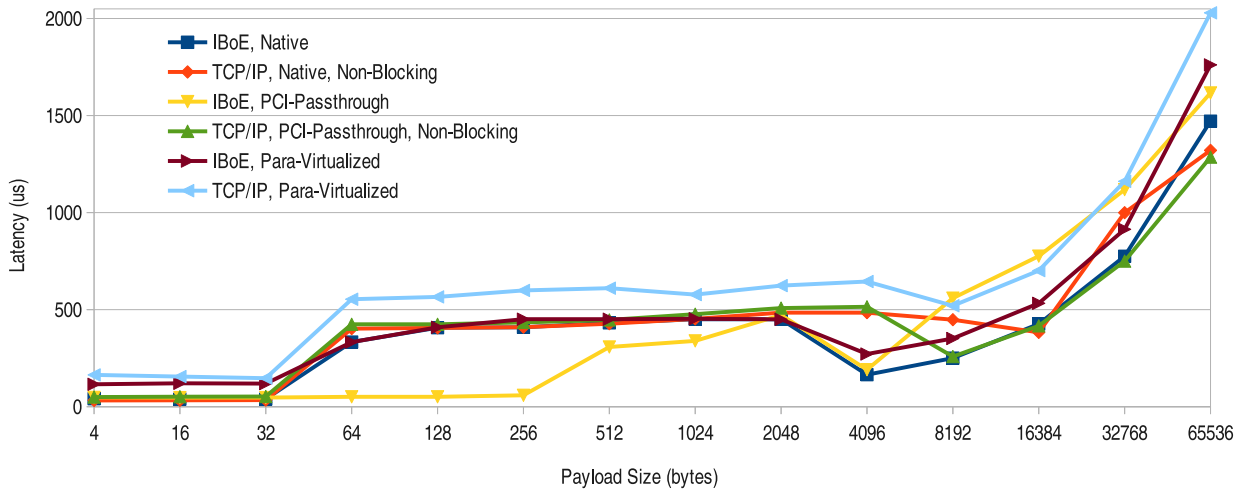

Figure 6.1: Latency results for Native, Para-Virtualized, and PCI-Passthrough for TCP/IP and IBoE

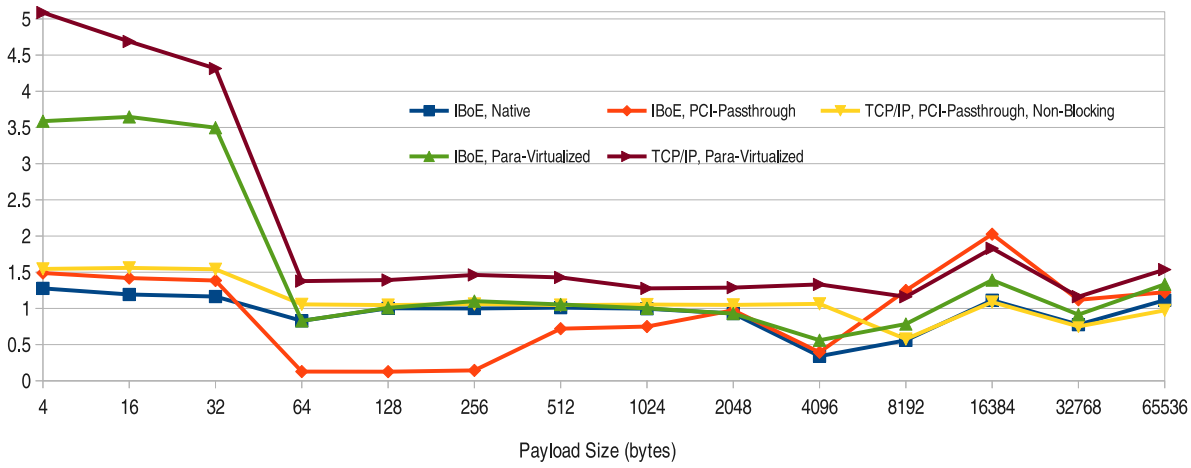

Figure 6.2: Latency results relative to TCP/IP

pass-through, IBoE outperforms native TCP/IP from 4 byte messages up to 4096 byte messages. On larger messages the latency of IBoE increases 66% faster than it did when running natively. The `Vt-d` implementation on the motherboards are most likely to blame for this since the IOMMU is not set up properly by the BIOS to handle the pass-through interrupts due to a bug in the proprietary BIOS code. Since Infiniband has a smaller MTU size this increase in interrupt latency begins to show as the number of received packets increases with Payload size.

6.2 Breakdown of Latency Contributors

Data from the Linux trace tools is compiled to show where the software latencies in each test occur. The latency data is broken down by function for send and receive of IBoE native, TCP/IP native, and the virtualized IBoE. Figure 6.3 shows the breakdown of for a native TCP/IP send message. Figure 6.4 shows the software latency breakdown as a percent of total computation time. When going from a 4 byte message to

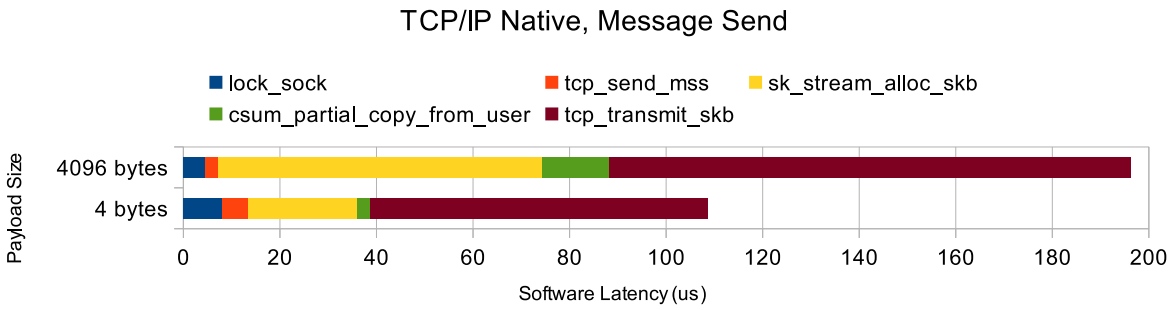

Figure 6.3: Breakdown of Kernel Latency for Native TCP/IP Send Message

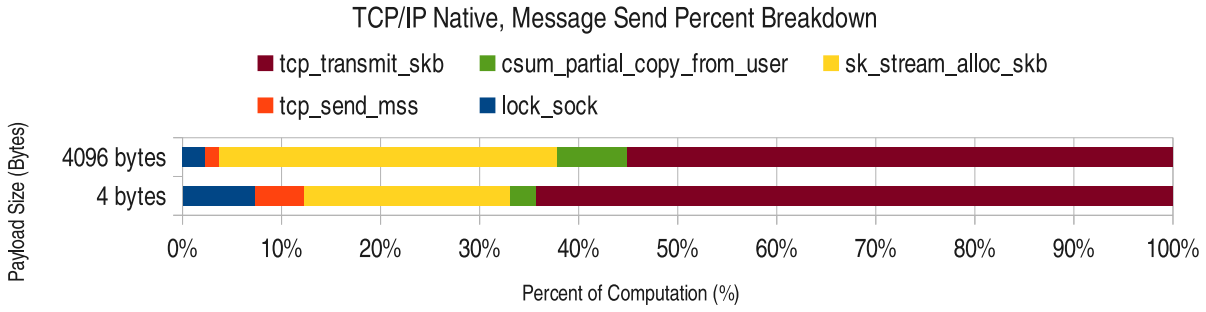

Figure 6.4: Percent Breakdown of Kernel Latency for Native TCP/IP Send Message

a 4096 byte message the software latency scales by 57%. The receive breakdown with respect to time can be seen in Figure 6.5 with the percent of total computation graph in Figure 6.6. On the receive the software latency increases by 98% when the payload size is increased from a 4 byte message to a 4k message. A detailed discussion of the contributors is given below.

The data for the TCP send stack (Figure 6.3) breaks down as follows. First, the system has to get the lock on the TCP socket using the function `lock_socket`. The `tcp_send_mss` function is responsible for setting the messages maximum segment size. Then the kernel allocates an `sk_buff` to copy the userspace data into the kernel. This happens in the `sk_stream_alloc_skb` function. The next function call is `csum_partial_copy_from_user` which checksums and copys the data from userspace into kernelspace. The last and longest function the send call travels through is `tcp_transmit_skb`. This function does the destination lookup, inserts the mac address into the packet header, and recalculates the checksum. It also inserts the IP layer information and then sends the `sk_buff` to the driver for transmission. While the `lock_sock` and `tcp_send_mss` functions stay roughly static as the messages size

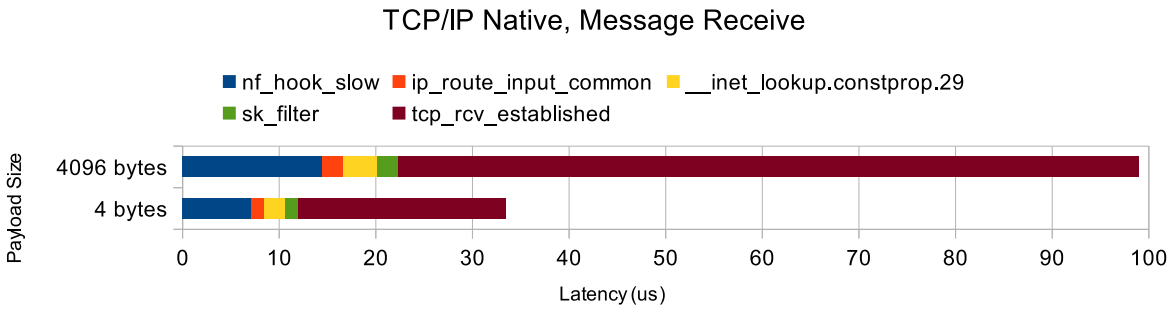

Figure 6.5: Breakdown of Kernel Latency for Native TCP/IP Receive Message

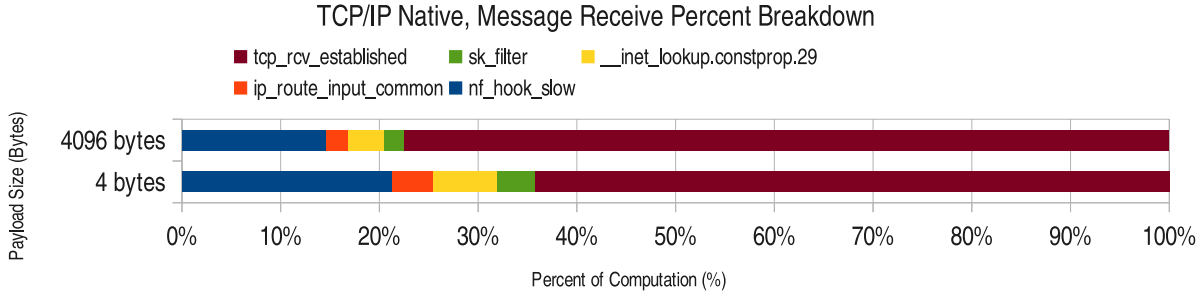

Figure 6.6: Percent Breakdown of Kernel Latency for Native TCP/IP Receive Message

grows `sk_stream_alloc_skb`, `csum_partial_copy_from_user`, and `tcp_transmit_skb` grow with messages size.

On the receive side for TCP/IP native (Figure 6.5) the trace starts when `nf_hook` detects a TCP_IP message from the network hardware driver. This detection occurs within the `nf_hook_slow` function call. The packet is then passed to the IP layer through the `ip_route_input_common` function which determines which interface the message is intended for and routes the message. The `inet_lookup` function then routes the message to the proper interface where the `sk_filter` function trims the `sk_buff` to the required size. `tcp_rcv_established` is then called which transfers the ownership of the `sk_buff` to userspace and acks the packet. Both `nf_hook_slow` and `tcp_rcv_established` grow with respect to packet size.

The native IBoE latency breakdown for send and receive can be seen in Figure 6.7 and Figure 6.9

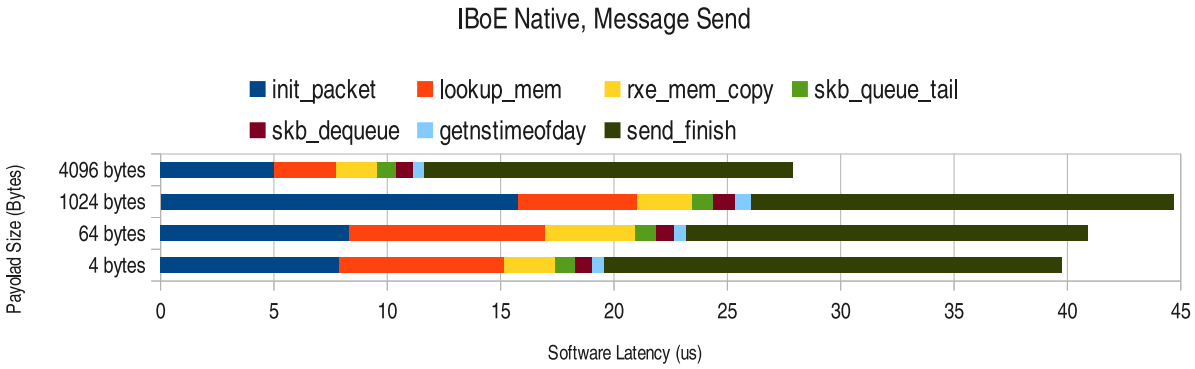

Figure 6.7: Breakdown of Kernel Latency for Native IBoE Send Message

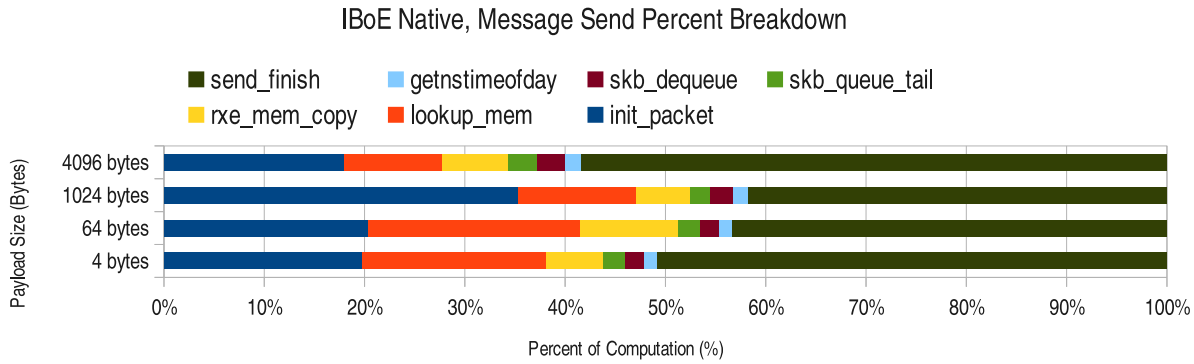

Figure 6.8: Percent Breakdown of Kernel Latency for Native IBoE Send Message

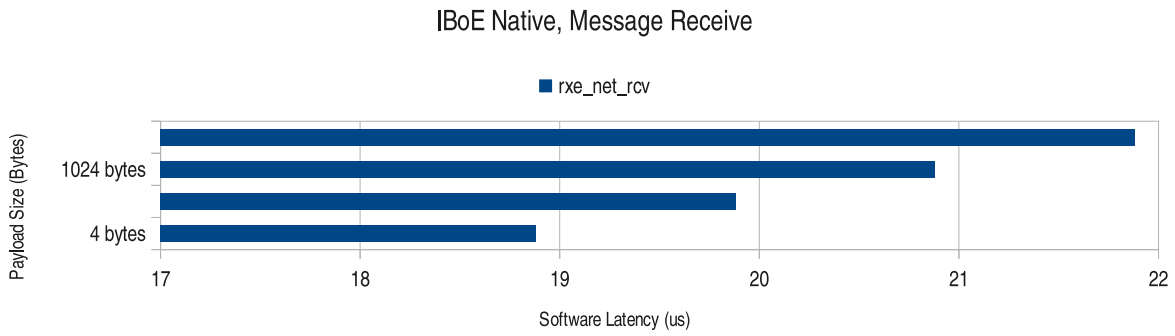

Figure 6.9: Breakdown of Kernel Latency for Native IBoE Receive Message

respectively. The percent of computation break down for an IBoE native send is shown in Figure 6.8. For native IBoE measurements were taken at 4, 64, 1024, and 4096 byte payload sizes. The send path for IBoE starts with the `init_packet` function. This function allocates the `sk_buff` to be used for the packet and initializes the device information and Ethernet header information in the `sk_buff`. The payload data is then located with `lookup_mem` and then copied from userspace into the `sk_buff` in `rxe_mem_copy`. The buffer is then attached to the `sk_buff` output queue with `skb_queue_tail` the head of the queue is then removed with `skb_dequeue` and `getnstimeofday` is then called to ensure that enough time has passed since the last send. This keeps the system from crashing the network driver since the system can send faster than the network card can deliver. The code is set to wait 700ns between packets. The `send_finish` function calls the network card driver to transmit the packet immediately. The data shows that the processing as a function of packet size is fairly constant.

The native IBoE receive trace starts when `nf_hook` calls the Infiniband receive handler `rxe_net_rcv`. From here the IBoE code puts the `sk_buff` in the complete queue and then posts that it has received a message. It then frees the `sk_buff` and returns. The messages in the complete queue can then be read from userspace. The software latency for this process is, as can be seen in Figure 6.9, fairly constant between message sizes. It is important to note the differences in the IBoE and TCP/IP software stacks that create these processing savings.

For IBoE the send and receive stacks are consistently faster than the TCP/IP send stack. The reason for this is there is less processing by definition on the IBoE stack than there is on the TCP/IP stack. While the copy from userspace for each is on similar magnitudes. The IBoE stack requires less time to initialize the

packet and less time to transmit the packet. On messages over 1024 bytes the IBoE trace data is deceiving. As can be seen from Figures 6.7 and 6.9, the 4096 byte trace has very low latency realized by savings in the `init_packet`, `lookup_mem`, and `copy_data` functions. The maximum transfer unit used for IBoE used in these experiments was 1024 bytes. Thus, for a 4096 byte message 4 packets are allocated and sent in direct succession. These savings are most likely from better cache performance as a result of the same instructions and data being used frequently. The real software latency for sending a 4096 byte IBoE message is probably closer to 4 times the value seen in Figure 6.7. As can be seen by the overall latency data in Figure 6.1 the compounded software latency of sending 4 messages for a 4096 byte payload does not directly translate into increased total latency. This is a result of processing being done on the first message on the receive while the next piece of the message is being sent. There is considerably less time spent processing IBoE messages at each messages size, but speed up is only seen at certain message sizes. These facts make a case for looking at the network card drivers and hardware for further decreased latency numbers. The virtualized IBoE traces give more insight into latency problems with the network card and driver.

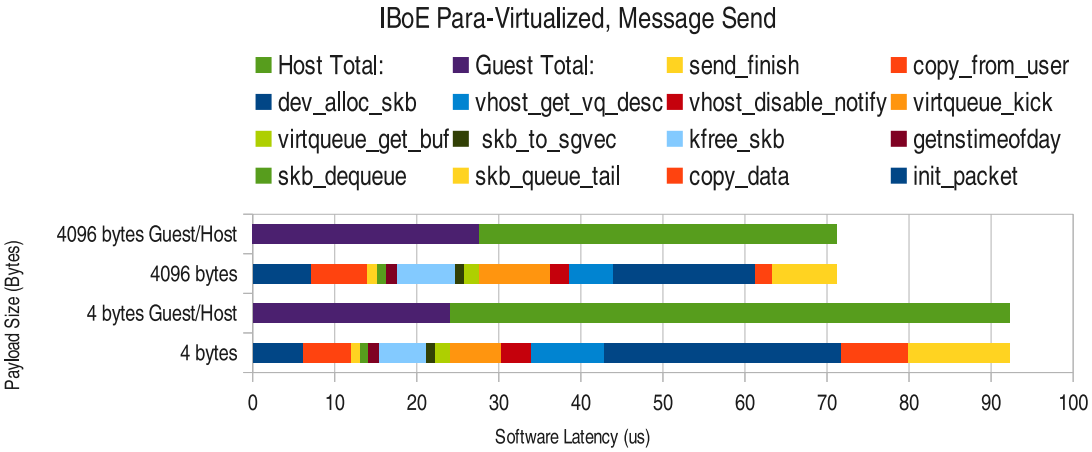

Figure 6.10: Breakdown of Kernel Latency for Para-Virtualized IBoE Send Message w/o SKB Ring Buffer

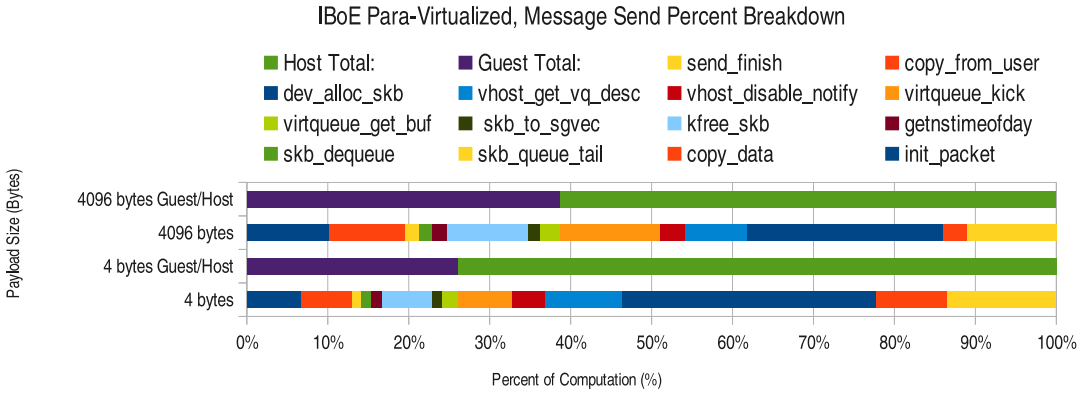

Figure 6.11: Percent Breakdown of Kernel Latency for Para-Virtualized IBoE Send Message w/o SKB Ring Buffer

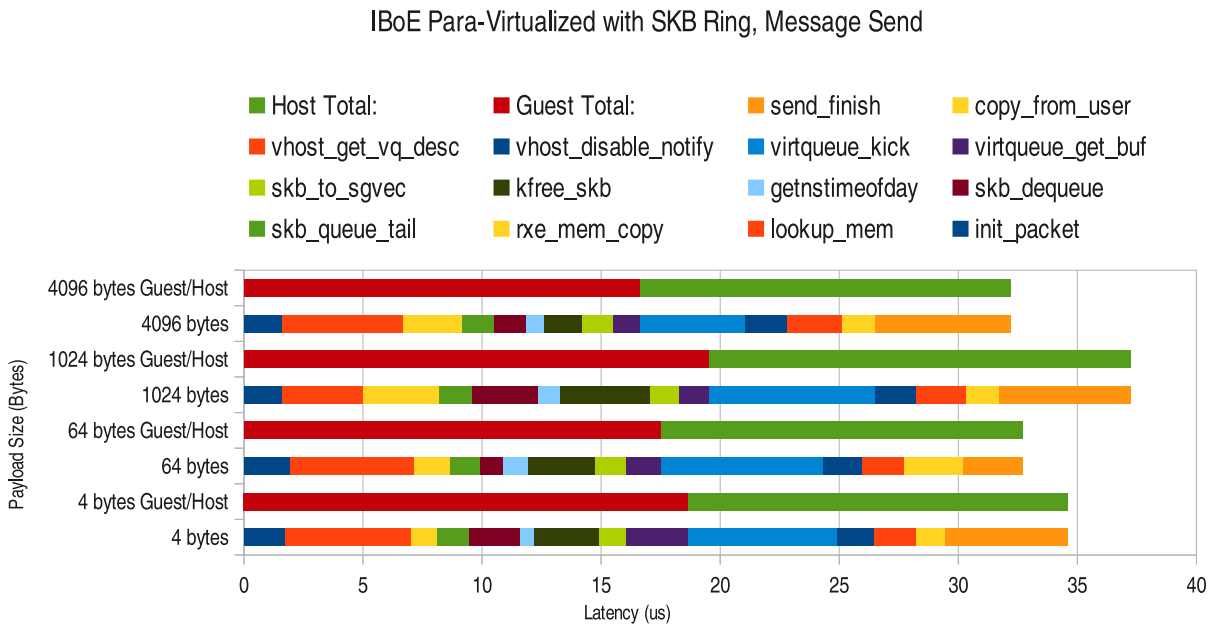

Figure 6.12: Breakdown of Kernel Latency for Para-Virtualized IBoE Send Message w/ SKB Ring Buffer

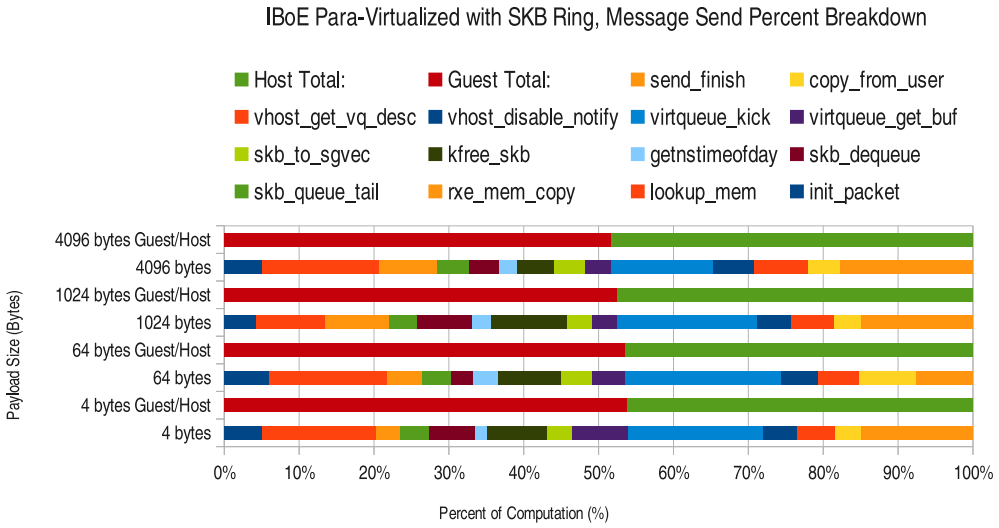

Figure 6.13: Percent Breakdown of Kernel Latency for Para-Virtualized IBoE Send Message w/ SKB Ring Buffer

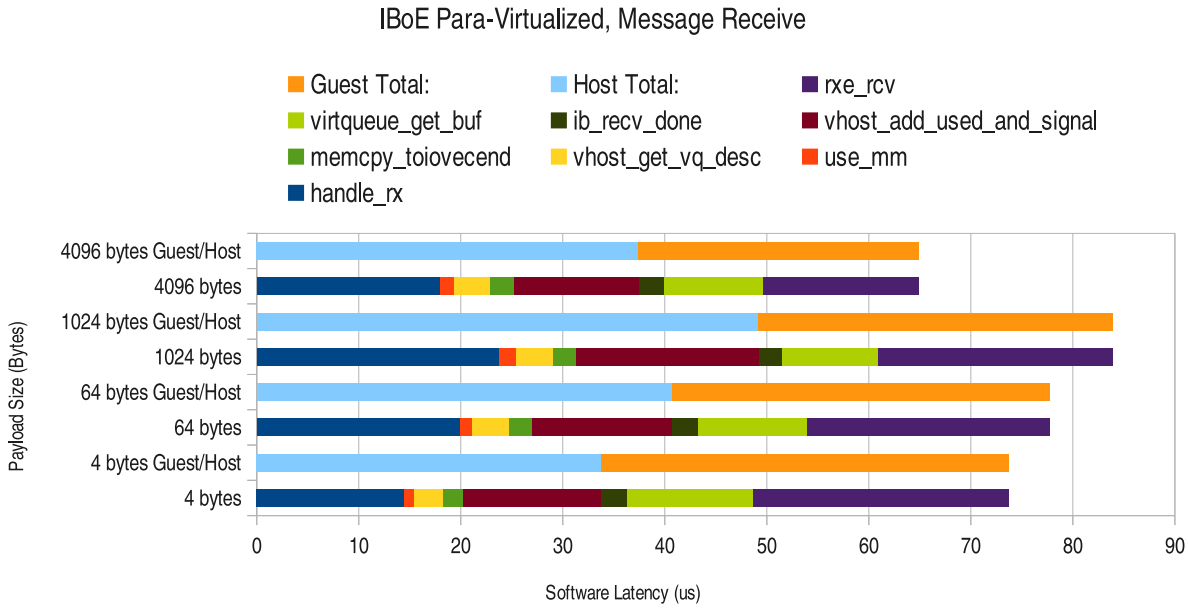

Figure 6.14: Breakdown of Kernel Latency for Para-Virtualized IBoE Receive Message

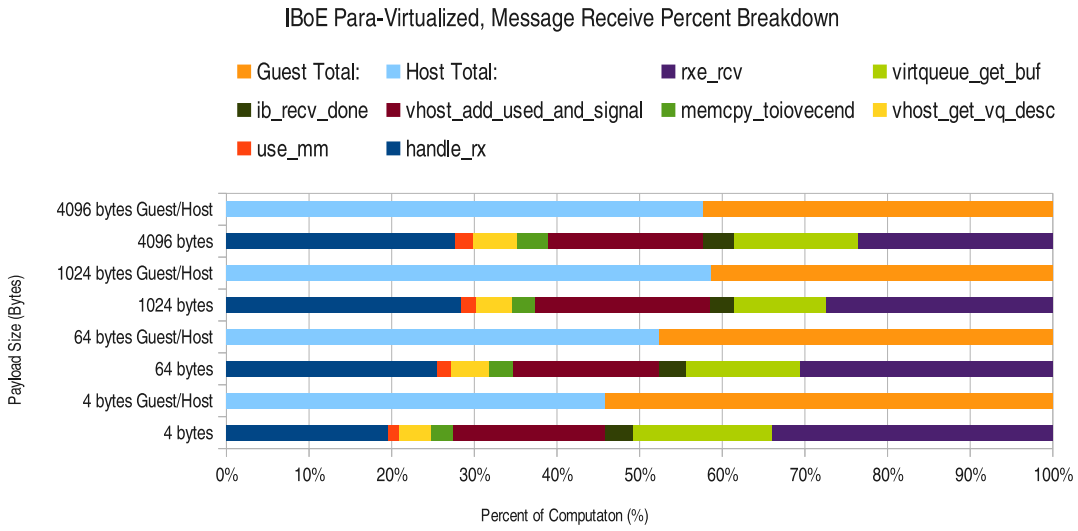

Figure 6.15: Percent Breakdown of Kernel Latency for Para-Virtualized IBoE Receive Message

6.3 Skbuff Study

The study of the para-virtualized IBoE send trace results revealed a large amount of time spent in allocating the sk_buff structure. The sk_buff data structure is used to store packet data in the Linux kernel. This section details decision to use a ring buffer of sk_buffers and documents the effect on the software latency and the over all latency of the virtualized IBoE implementation.

The original software latency and percent breakdowns for para-virtualized IBoE send are shown in Figure 6.10 and 6.11. In comparison with the native TCP/IP software latency it can be seen that for a 4 byte messages the para-virtualized IBoE data software latency is very high. We notice that large portions of time are spent allocating and freeing the `sk_buff`. To alleviate this problem the driver was changed to allocate a ring of `sk_buffs` upon initialization. This takes care of having to wait on allocation when sending messages. The buffers are allocated at the maximum transfer unit size and then the length of the packet is adjusted dynamically as messages are passed to the host. As shown in Figure 6.12 there is a drastic reduction in software latency from these changes. The percent computation for the original send path in Figure 6.11 and that of the new path in 6.13 show that the portion of time spent in the host is greatly reduced by the addition of the `sk_buff` ring.

The virtualized IBoE send (Figure 6.12) starts in the guest with the `init_packet` function just like in the native application. The first difference comes when the packet is to be transmitted after the `getnstimeofday` call. First, all old `sk_buffs` that the host has finished processing are freed. This consists of decrementing the use counter now since the buffers are preallocated. Then, the `sk_buff` is converted to a scatter-gather array in `skb_to_sgvec`. The next available `virtio` ring buffer is fetched by `virtqueue_get_buf` and is set to point to the address of the scatter-gather array. The guest then calls the `virtqueue_kick` function to alert the host there is a new message in the queue. The host then calls `vhost_get_vq_desc` and copies the data in the `sk_buff` to the host kernelspace in the `copy_from_user` function. The host then calls `send_finish` which calls the network driver to transmit the packet to the hardware destination address.

The receive trace for virtualized IBoE, shown in Figure 6.14, starts in the host when the `handle_rx` function is called by the Infiniband message handler registered with the network device. `Handle_rx` schedules a kernel workqueue to copy the received message to the guest. This has to be done in a workqueue

since copies to userspace memory are blocking. There is an average 5μs latency associated with scheduling a workqueue. This would not be shown in the trace data. In the kernel workqueue the host gets the next available ring buffer address from the guest with `vhost_get_vq_desc`. The host then copies the `sk_buff` into the buffer with `memcpy_toiovcend`. To signal the guest that new data is available the host calls `ib_recv_done`. The `ib_recv_done` function schedules a high priority tasklet to finish the rest of the receive processing. Within the tasklet the guest then reads the new data with `virtqueue_get_buf` and calls `rxe_rcv` on the new packet. It is important to note that the tasklet function scheduling takes an average of 2.5μs and this would not be seen in the trace data.

On the para-virtualized send There is not a significant per packet cost reduction for 4k messages at the software latency level like there was on the native trace. It is important to note again that the messages are broken up into 1k payload sizes so a single 4096 byte send results in 4 packets being sent. The code has some optimizations that aid this process. If the host is currently sending a packet and it sees a new packet has arrived after sending the first it will send the next one without waiting for the signal from the guest. This would mean that the host may be processing the packet before the virtqueue kick function returns. For the receive code it is important to note that the `handle_rx` function schedules a kernel workqueue. Since the host is a multi core system is is plausible that while this workqueue is executing a new packet can be received from the network. It is also important to note that before exiting the processing of the receive the guest checks for new receive messages in the queue so it is also possible for the guest to start processing the message before the vhost signaling function returns. The most promising result from the virtualized IBoE data is that it exhibits the same slow scaling with respect to message size on the receive as the native IBoE.

The trace data shows that on both the send and receive size of the para-virtualized IBoE there is significant time spent sending interrupts from host to guest and guest to host. The host interrupt of the guest averages 6.5μs and the guest interrupt to the host averages 2.5μs. These two interrupts combined with the workqueue latency of 5μs on the host receive and the tasklet latency on the guest bottom half receive of 2.5μs add up to 14.5μs of delay. This is in large part the reason that the virtualized Infiniband cannot beat native TCP/IP for small messages.

After making changes to the `sk_buff` allocation the overall latency tests were run again. Very little gain was realized from these modifications. The improvement on average was 1.5μs. Any savings these

software modifications made must be getting consumed else where in the IBoE stack. Also, the drop in latency at the 4096 byte messages size seems to be counter intuitive since this is the point where multiple messages begin to be sent. The only things not measured with the trace tools were the driver code and the hardware itself. Hardware coalescing makes sense in the both cases. If the Ethernet hardware is coalescing messages then reducing the software latency will have no effect unless the latency was already over the coalescing time. Also, the 4096k message size may fill the buffer with enough messages fast enough that the card stops coalescing the messages. The realtek driver offers no low latency configuration settings. So it is likely that the realtek hardware is responsible for unrealized latency gains.

6.4 Realtek vs. Intel Gigabit Ethernet Cards

Due to the inconsistencies detailed in the previous section the Native IBoE ping-pong tests need to be run on different Ethernet hardware. The tests are run on Intel 82574L Gigabit Ethernet cards. The 82574L card was chosen for the ability to set the coalesce options within the Intel driver. The desire was to eliminate any coalescing the Realtek hardware may have been doing.

The results can be seen in Figure 6.16 and differences are surprising. For the test both TX and RX coalesce settings are set to 0, This prevents the card from throttling messages and waiting for more data from the driver. While, for 4, 16, and 32 byte messages the Intel cards perform 2 times worse than the Realtek cards. For payloads from 64 to 2048 bytes the Intel card has a quarter of the latency of the Realtek cards. After 2048 byte payload the latencies of the two cards are within error. The type of gigabit Ethernet hardware in use can have a large effect on network latency. However, Significant improvement was not seen on the Intel cards. While it is not looked at in this study it is possible that there are also improvements that can be made in the Ethernet drivers of each card. Without intimate knowledge of each card making improvements in the drivers is a large task. This coupled with the loss in code portability lead to driver modifications not being included in this thesis.

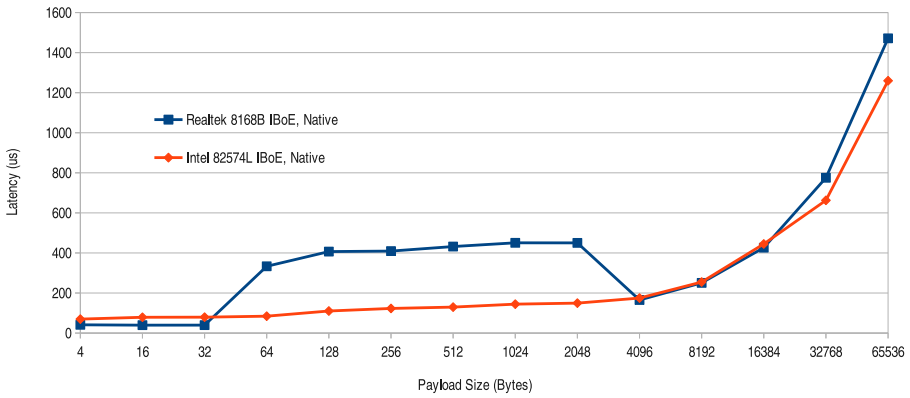

Figure 6.16: IBoE Native Ping-pong Tests results on Intel and Realtek Cards

6.5 Bandwidth Measurements

The results of the bandwidth test can be seen in Figure 6.17. The bandwidth results relative to native TCP/IP are shown in Figure 6.18. These bandwidth tests were run on the Realtek 8168B cards. The para-virtualized IBoE performed surprisingly well, within error or better than TCP/IP between 64 byte and 2 kilobyte message sizes. As expected the bandwidth of the Infiniband protocol is worse than that of TCP/IP native for smaller and larger messages. Sending messages right away, without combining or coalescing, allows the Infiniband driver to get lower latency, but it hurts the bandwidth performance. It is encouraging to see that virtualization does not hurt bandwidth any more than the protocol itself. The reason the para-virtual TCP/IP gets much better bandwidth performance is because it automatically combines messages as they come in. This is within the `virtio` driver so even though TCP/IP is set not to combine messages the `virtio` driver does this internally.

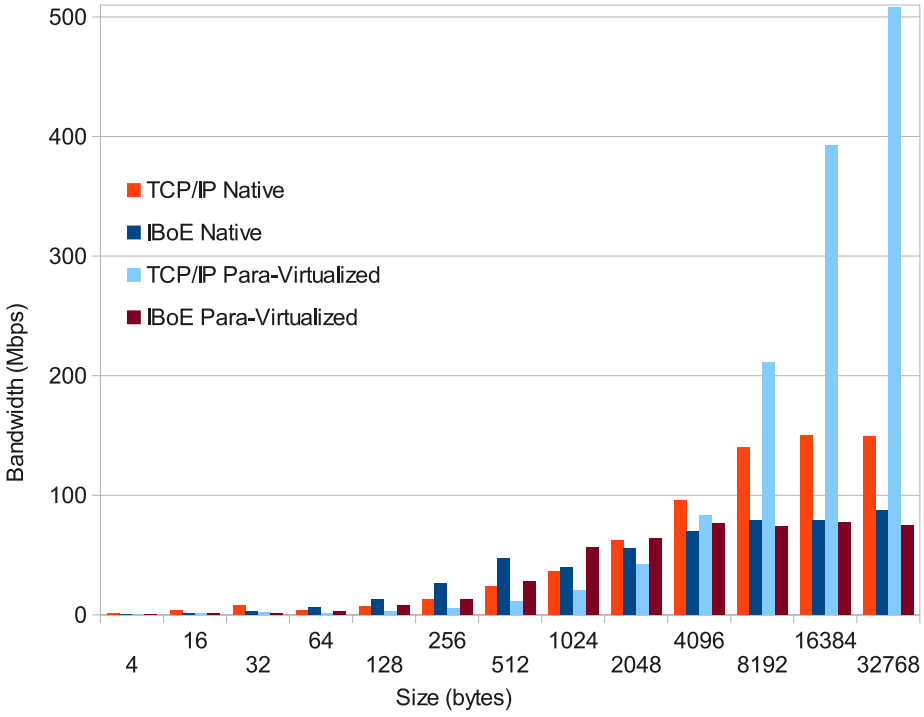

Figure 6.17: Performance Impact on Raw Network Bandwidth

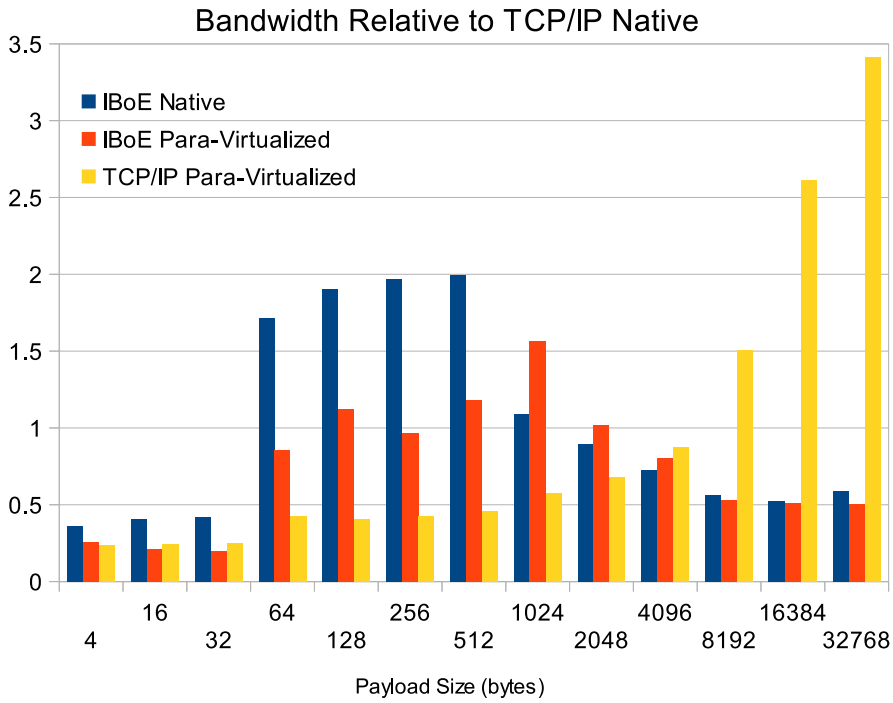

Figure 6.18: Performance Impact on Bandwidth Relative to TCP/IP

Chapter 7

Conclusions and Suggestions for Future Research

The para-virtualized IBoE drivers developed for this study show lightweight protocols as a viable option for reducing and, at a number of message sizes, eliminating the negative effects of virtualized network I/O latency. The para-virtualized IBoE driver significantly outperforms the para-virtualized TCP/IP driver at every message size. The para-virtualized IBoE driver also outperforms native TCP/IP in a significant number of cases.

The weakness of the para-virtualized IBoE driver lies in message sizes smaller than 32 bytes. Native TCP/IP has significantly better latency performance at these message sizes. The costs of host-to-guest and guest-to-host interrupts coupled with the latency costs of additional workqueues and tasklets are largely responsible for this short fall. As the message size increases the lower software latency of the Infiniband protocol hides these added latencies.

Despite the higher latency at small message sizes, para-virtualized IBoE is an attractive option for the use case of virtualized Beowulf clusters. As it can be used without the additional cost of upgrading every node on the network to a higher speed interconnect and has latency performance that is comparable to native TCP/IP. Another strength of IBoE is HPC applications that already support Infiniband are easy to configure for IBoE which makes the barrier for use very low. As 10G Ethernet systems become more prevalent para-virtualized IBoE will be effective on these as well. For these reasons the use of lightweight protocols in virtualized

operating systems has the potential to be a solution for the I/O latency problem in many applications.

7.1 Suggestions for Future Work

This study has shown that network latency performance is highly dependent on the Ethernet hardware being used. This is shown by the very different results achieved with the Intel cards in Chapter 6. It would be useful to experiment with driver modifications on several popular cards to find driver programming methods to support low latency operation. While this would require the changing of driver code it would not be a barrier to use, but an optimization of the system. Also, other Ethernet hardware need be tested with the IBoE para-virtualized driver this includes the newer 10G Ethernet cards.

The premiere sources of of latency within the IBoE para-virtualized driver are the host-to-guest and guest-to-host interrupts along with workqueue and tasklet latencies. Modifications that reduce the contributions of any or all of these functions would further improve the performance of the para-virtualized IBoE drivers.

The para-virtualized IBoE driver needs to be tested under real world workloads. Running IBoE on a Beowulf cluster with virtual nodes would be a viable next step in the process. The expectation would be that it would perform similarly to that of TCP/IP native. This would require that an HPC application be modified to use IBoE as its communication protocol and that test be run on varying numbers of nodes.

Bibliography

[1] "Interconnect analysis: 10gige and infiniband in high performance computing," tech. rep., HPC Advisory Council, 2009.

[2] D. Jefferson, B. Beckman, F. Wieland, L. Blume, and M. Diloreto, "Time warp operating system," *Proceedings of the eleventh ACM Symposium on Operating systems principle (SOSP 87)*, vol. 21, Nov. 1987.

[3] J. S. Steinman, "The warpiv simulation kernel," *Proceedings of the 19th Workshop on Principles of Advanced and Distributed Simulation (PADS 05)*, 2005.

[4] P. De, R. Kothari, and V. Mann, "Identifying sources of operating system jitter through fine-grained kernel instrumentation," *2007 IEEE International Conference on Cluster Computing*, pp. 332–340, Sept. 2007.

[5] *InfiniPath QHT7140*.

[6] S. Larson, P. Sarangam, R. Huggahalli, and S. Kulkarni, "Architectural breakdown of end-to-end latency in a tcp/ip network," *19th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2007)*, pp. 195–202, Oct. 2007.

[7] O. Pentakalos, *An Introduction to the InfiniBand Architecture*. O'Reilly, 2002.

[8] "Kvm kernel based virtual machine," tech. rep., Red Hat Inc., Jan. 2009.

[9] M. Fenn, M. A. Murphy, and S. Goasguen, "A study of a kvm-based cluster for grid computing," in *ACM-SE 47: Proceedings of the 47th Annual Southeast Regional Conference*, (New York, NY, USA), pp. 1–6, ACM, 2009.

[10] G. Motika and S. Weiss, "Virtio network paravirtualization driver: Implementation and performance of a de-facto standard," *Computer Standards and Interfaces*, no. 34, pp. 36–47, 2011.

[11] J. Hurwitz and W. chun Feng, "End-to-end performance of 10-gibit ehternet on commodity systems," *IEEE Computer Society*, pp. 10–22, Jan. 2004.

[12] K. Mansley, G. Law, D. Riddoch, G. Barzinz, N. Turton, and S. Pope, "Getting 10gb/s from xen: safe and fast device access from unprivlileged domains," tech. rep., Solarflare Communications, Inc, 2008.

[13] A. Barczyk, D. Bortolotti, A. Carbone, J. Dufey, D. Galli, B. Gaidioz, D. Gregori, B. Jost, U. Macoroni, N. Neufeld, G. Peco, and V. Vagnoni, "High rate transmission on ethernet lan using commodity hardware," *IEEE Transactions on Nuclear Science*, vol. 53, June 2006.

[14] L. Nussbaum, F. Anhalt, O. Mornard, and J.-P. Gelas, "Linux-based virtualization for hpc clusters," in *Linux Symposium 2009*, July 2009.

[15] J. N. Matthews, E. M. Dow, T. Deshane, W. Hu, J. Bongio, P. F. Wilbur, and B. Johnson, *Running Xen: A Hands-On Guide to the Art of Virtualization*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2008.

[16] K. Fraser, S. H, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson, "Safe hardware access with the xen virtual machine monitor," in *In 1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS)*, 2004.

[17] AMD, "Amd i/o virtualization technology (iommu) specification," tech. rep., Advanced Micro Devices, Inc., 2009.

[18] D. Abramson, J. Jackson, S. Muthrasanallur, G. Neiger, G. Regnier, R. Sankaran, I. Schoinas, R. Uhlig, B. Vembu, and J. Wiegert, "Intel virtualization technology for directed i/o," *Intel Technology Journal*, vol. 10, pp. 179–192, Aug. 2006.

[19] B.-A. Yassour, M. Ben-Yehuda, and O. Wasserman, "Direct device assignment for untrusted fully-virtualized virtual machines," tech. rep., IBM Research, 2008.

[20] G. Ciaccio, "Messaging on gigabit ethernet: Some experiments with gamma and other systems," *Cluster Computing*, vol. 6, no. 2, pp. 143–151, 2003.

[21] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active messages: a mechanism for integrated communication and computation," tech. rep., University of California, Berkeley, Berkeley, CA, USA, 1992.

[22] T. H. Von Eicken, *Active messages: an efficient communication architecture for multiprocessors*. PhD thesis, University of California, Berkeley, 1993. Co-Chair-Culler, David E. and Co-Chair-Wawrzynek, John.

[23] "The case for infiniband over ethernet," tech. rep., Mellanox Technologies, Apr. 2008.

[24] J. Liu, W. Huang, B. Abali, and D. K. Panda, "High performance vmm-bypass i/o in virtual machines," in *Proceedings of the annual conference on USENIX '06 Annual Technical Conference*, ATEC '06, (Berkeley, CA, USA), pp. 3–3, USENIX Association, 2006.

[25] M. Rashti and A. Afsahi, "10-gigabit iwarp ethernet: Comparative performance analysis with infiniband and myrinet-10g," *Parallel and Distributed Processing Symposium*, pp. 1–8, Mar. 2007.

[26] D. Dalessandro, P. Wyckoff, and G. Montry, "Initial performance evaluation of the neteffect 10 gigabit iwarp adapter," *Cluster Computing 2006 IEEE International*, pp. 1–7, Sept. 2006.

[27] H. Subramoni, P. Lai, M. Luo, and D. Panda, "Rdma over ethernet - a preliminary study," *Cluster Computing and Workshops*, pp. 1–9, Aug. 2009.

[28] N. Regola and J. Ducom, "Recommendations for virtualization technologies in high performance computing," *Cloud Computing Technology and Science*, pp. 409–416, Nov. 2010.

[29] L. Xia, Z. Cui, J. R. Lange, Y. Tang, P. A. Dinda, and P. G. Bridges, "Vnet/p: bridging the cloud and high performance computing through fast overlay networking," in *Proceedings of the 21st international*

symposium on High-Performance Parallel and Distributed Computing, HPDC '12, (New York, NY, USA), pp. 259–270, ACM, 2012.

[30] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. king Su, "Myrinet: A gigabit-per-second local area network," *IEEE Micro*, vol. 15, pp. 29–36, 1995.

[31] R. Russell, "virtio: towards a de-facto standard for virtual i/o devices," *SIGOPS Operating Systems Review*, vol. 42, pp. 95–103, July 2008.