Master Thesis

vVerbs
a paravirtual subsystem for RDMA-capable network interfaces

Author(s):
Pfefferle, Jonas

Publication Date:
2014

Permanent Link:
https://doi.org/10.3929/ethz-a-010154461

Rights / License:
In Copyright - Non-Commercial Use Permitted

This page was generated automatically upon download from the ETH Zurich Research Collection. For more information please consult the Terms of use.
Master Thesis

**vVerbs**
A Paravirtual Subsystem for RDMA-capable Network Interfaces

Jonas Pfefferle

Prof. Thomas R. Gross
Laboratory for Software Technology
ETH Zürich

Dr. Patrick Stüdi
IBM Research Zürich
Responsible assistant

April 2014
Abstract

Recent development in data center applications puts extra pressure on the network stack. Unfortunately, traditional socket-based networking technologies are limited by copies on the data path, which not only increases latency, but also adds extra processing overhead. One decade ago, the remote direct memory access technology (RDMA) has been developed by the super computing community to provide ultra low-latency and high-throughput. RDMA allows to perform data path operations without the involvement of the operating system, thus, keeping CPU overhead to a minimum. Hence, recently there has been a growing interest in RDMA to be used in cloud environments. However, existing RDMA virtualization solutions, like single-root I/O virtualization (SR-IOV), are inherently static, e.g. supporting migration is complex, which is in contrast to flexibility and manageability requirements of cloud systems. In this work, we propose vVerbs, a paravirtual subsystem for RDMA-capable network interfaces. We believe that RDMA’s way of separating control and data plane is predestined to be used in virtualized environments. We leverage this property by allowing applications inside a VM to completely bypass the hypervisor on data path operations. Control path operations, e.g. used to setup resources, are performed by a paravirtual driver. We show that vVerbs can achieve bare-metal VM-to-VM latency and throughput without the need for I/O hardware virtualization support, such as SR-IOV. Furthermore, we believe that, with this approach, we can provide flexible and manageable, high-performance I/O virtualization which allows for large-scale deployments and enables new applications in the cloud.
Zusammenfassung

Acknowledgments

First and foremost, I would like to thank my supervisor Dr. Patrick Stüdi for his patient and continuous support during the thesis. His enthusiasm for new ideas and challenges has been inspiring, and I value his personal advice. I would also like to thank Dr. Bernard Metzler. His unsurpassed knowledge of RDMA has led to interesting technical discussions, and was a great input to this work. In addition, I would like to thank Animesh Trivedi for his advice on both technical and personal level. My appreciation also goes to Prof. Thomas Gross, who made this external master’s thesis possible. Last but not least, I would like to thank Clemens Lutz and Andreas Noever for proofreading this thesis.

Jonas Pfefferle

April, 2014
## Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDMA</td>
<td>Remote direct memory access</td>
</tr>
<tr>
<td>VM</td>
<td>Virtual machine</td>
</tr>
<tr>
<td>SR-IOV</td>
<td>Single-root I/O virtualization</td>
</tr>
<tr>
<td>OFED</td>
<td>OpenFabrics Enterprise Distribution</td>
</tr>
<tr>
<td>RNIC</td>
<td>RDMA-capable network interfaces</td>
</tr>
<tr>
<td>QP</td>
<td>Queue pair</td>
</tr>
<tr>
<td>CQ</td>
<td>Completion queue</td>
</tr>
<tr>
<td>MR</td>
<td>Memory region</td>
</tr>
<tr>
<td>MMU</td>
<td>Memory management unit</td>
</tr>
<tr>
<td>APIC</td>
<td>Advanced programmable interrupt controller</td>
</tr>
<tr>
<td>GPA</td>
<td>Guest physical address</td>
</tr>
<tr>
<td>HVA</td>
<td>Host virtual address</td>
</tr>
<tr>
<td>GVA</td>
<td>Guest virtual address</td>
</tr>
<tr>
<td>VA</td>
<td>Virtual address</td>
</tr>
<tr>
<td>HVM</td>
<td>Host virtual memory</td>
</tr>
</tbody>
</table>
## Contents

1 Introduction 1  

2 Background 3  
   2.1 RDMA . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3  
   2.2 Virtualization . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6  
      2.2.1 Advanced Programmable Interrupt Controller . . . . . . . . . . . . . . . . . . . 8  

3 Design 9  

4 Implementation 17  
   4.1 SoftiWARP . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18  
   4.2 Chelsio Terminator 4 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19  

5 Evaluation 21  
   5.1 Latency . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22  
   5.2 Throughput . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28  
   5.3 Interrupts . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31  
   5.4 Comparison of Network Virtualization Techniques . . . . . . . . . . . . . . . . . . . 32  

6 Related Work 37  
   6.1 High-performance VMM-bypass I/O in virtual machines[22] . . . . . . . . . . . . . . 37  
   6.2 vRDMA[2] . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38  

7 Future Work 41  
   7.1 Migration and Checkpointing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41  
   7.2 Connection Management . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42  

xi
1 Introduction

Every day, petabytes of data are processed by modern data center applications, like graph [23] and stream [37] processing, data stores [26] and sophisticated data analytics applications. Typically, thousands of compute nodes interact over a network to cope with this amount of data. However, meeting these application demands adds significant pressure on the network stack. On one hand, this requires high-bandwidth to allow distributing huge amounts of data in the network, on the other hand, growing demands for real-time processing imposes timing constraints which requires low-latency networking. Unfortunately, traditional socket-based networking technologies are limited by copies on the data path, which not only increases latency, but also adds considerable processing overhead. One decade ago, the remote direct memory access technology (RDMA) has been developed by the super computing community to provide ultra low-latency and high-throughput. RDMA removes copies on the data path and enables direct isolated application access to RDMA-capable network interfaces (RNIC). This allows to perform data path operations without the involvement of the operating system, thus, keeping network computation overhead to a minimum. While RDMA has been successfully used in super computing environments, there has been a growing interest to use it in cloud environments. However, these systems typically use virtual machines (VMs) to provide an easy to manage, flexible and scaleable IT infrastructure and therefore, require virtualization support for RDMA. Although, there are existing virtualization solutions for RDMA, we believe that they are not a good fit for cloud environments. For example, single-root I/O virtualzation (SR-IOV) is a virtualization technique enabling hardware to multiplex itself into several instances (virtual functions) which can be assigned to a VM [28]. However, besides adding complexity to the hardware their setup is inherently static, e.g. supporting migration is complex, which is in contrast to the flexibility and manageability requirements of cloud systems [20][11]. Therefore, we propose vVerbs, a paravirtual subsystem for remote direct memory access (RDMA). We believe that RDMA’s way of separating control and data plane is predestined to be used in virtualized environments. That is, RDMA already supports multiplexing between applications by providing direct, isolated access to its network resources. Together with direct data access capabilities this allows an application to completely bypass the operating system on data path operations. We leverage these properties by allowing user space applications inside a virtual machine to access their RDMA network resources directly and use their application memory for direct data access, thus, completely bypassing the hypervisor on data path operations. Control path operations, e.g. used to setup resources or for connection management, are performed by a paravirtual driver. We believe that vVerbs can provide flexible and manageable, cut-through, high-performance I/O
virtualization which allows for large-scale deployments and enables new applications in the cloud. In Section 2 we describe background information for vVerbs like RDMA, virtualization and interrupt processing on a virtualized x86 architecture. Then in Section 3 we discuss design choices and conclude the final design. Furthermore, we discuss in detail how memory registration from a virtual machine (VM) works and how RDMA network resources can be mapped to a VM to allow direct isolated access from applications. In Section 4 we go into implementation details of vVerbs. In the following Section 5 we show that vVerbs can deliver close to bare-metal latencies and throughput inside virtual machines. Moreover, we show that we can compete with pass-through device solutions while providing flexibility and manageability of VMs. Further, in Section 6 we discuss related work. Then in Section 7 we look at how vVerbs could be extended to support migration and better connection management abilities. Furthermore, we talk about possible extensions to the OFED RDMA stack to make it virtualization aware, thus allowing to reduce resource and memory mapping complexity in a paravirtualized driver.
2 Background

2.1 RDMA

Remote direct memory access (RDMA) is a network technology which enables low-latency and high-throughput data transfer between network nodes with minimal processing overhead[7]. This is achieved by allowing a RDMA-capable network interface (RNIC) to directly access application buffers, i.e. there are no copies on the data path like with socket-based networking. Additionally, applications can be given direct and isolated access to the RNIC, allowing to perform data transfers without the involvement of the operating system. Besides two-sided send and receive operations, RDMA also supports read and write operations. This allows an application to read and write from/to a remote memory buffer, and, because of direct data access capabilities of the RNIC, there is no involvement of the application or the operating system on the remote side. To support these operations, typically the whole network stack has to be offloaded onto the RNIC. The disadvantages of RDMA include the need for pinning physical memory to allow direct access from the RNIC, which stresses physical memory requirements and offloading the network stack by vendor-specific firmwares.

RNICs are typically programmed by RDMA Verbs, which are an abstract definition of the functionality provided by a RNIC, i.e. they describe the semantics of RDMA operations. Verbs includes work queues, completion queues, registration of memory, and how they interact with another. Among others, Verbs have been specified by the InfiniBand Trade Association (IBTA) and the RDMA Consortium [6][19]. We summarize the Verbs relevant for this thesis in Table 2.1.

RDMA operations specified by Verbs are always performed on memory buffers. To make them uniquely identifiable by a key and enable direct data access, they have to be registered once before they can be used in a data operations. Together with an offset, the key is used to identify a local or remote memory location in a data operation.

Verbs specifies the network resources, allowing to directly program the RNIC to execute data path operations. There are two types of resources: work queues, which are used to queue work requests from the applications, and completion queues, which are used to inform about completed work.

For each connection there is one work queue pair (QP): a send queue and a receive queue. To perform a send, write or read operation, a work request has to be posted on the send queue. Unlike other networking interfaces like sockets, receive requests of sufficient number and buffer size have
2. Background

Application

Buffer

Operating System

RNIC

Application

Buffer

Operating System

RNIC

Figure 2.1: RDMA

<table>
<thead>
<tr>
<th>Verbs call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create_QP</td>
<td>creates a send and receive queue</td>
</tr>
<tr>
<td>Modify_QP</td>
<td>changes internal state of QP</td>
</tr>
<tr>
<td>Destroy_QP</td>
<td>destroys QP</td>
</tr>
<tr>
<td>Create_CQ</td>
<td>creates completion queue</td>
</tr>
<tr>
<td>Poll_CQ</td>
<td>polls for work completions on a CQ</td>
</tr>
<tr>
<td>Req_notify_CQ</td>
<td>request completion event on new work completions (one shot)</td>
</tr>
<tr>
<td>Destroy_CQ</td>
<td>destroys CQ</td>
</tr>
<tr>
<td>Reg_User_MR</td>
<td>registers user memory</td>
</tr>
<tr>
<td>Dereg_User_MR</td>
<td>deregisters user memory</td>
</tr>
<tr>
<td>Post_Send</td>
<td>posts a work request on a send queue (send, write or read operation)</td>
</tr>
<tr>
<td>Post_Receive</td>
<td>posts a work request on a receive queue</td>
</tr>
</tbody>
</table>

Table 2.1: Verbs Calls

to be posted to the receive queue to allow sends from the remote site to be processed. Otherwise, data cannot be placed directly, which might require to abandon the connection. A work request includes all necessary information for the RNIC to perform the operation, but it does not include the data, except for a local and/or remote key as described above.

For each pair of work queues there can be an associated completion queue (CQ). A work request, if specified on creation, generates a work completion if successfully completed. A work completion can also indicate a failure of a work request. Work completions can be retrieved by polling the CQ. Additionally, there is an event mechanism which can be used to wait until a new work completion has been added.

The OpenFabrics Alliance (OFA) offers an open source implementation for the abstract Verbs interface called the OpenFabrics Enterprise Distribution (OFED). Their verbs API is the de-facto standard used to program RDMA applications on Windows and Linux operating systems. Figure 2.1 shows an overview of the OFED components described in the following.

OFED provides two APIs to program against, a kernel-level API (kverbs) and a user-level API (uverbs)\(^1\). They differ only slightly in functionality, however both depend on functionality provided

\(^1\)We also use these terms for their underlying implementation
2. Background

by the OFED verbs core running in the kernel.

Because most control operations depend on the underlying hardware, vendors have to register a
device-specific provider (kprovider) to kverbs. To allow user applications to bypass the kernel on
data path operations, e.g. when posting work requests, resources are shared with the RNIC and
directly mapped to the user space. The internal structure of these QPs and CQs is hardware specific
and thus requires the vendor to provide a user-level provider (uprovider). In contrast, control path
operations like registering memory can typically not be performed in user space and are forwarded
to the verbs core and eventually processed by the kprovider.

Work completions are stored in CQs to be retrieved by applications. OFED provides a verbs
call to poll a CQ for new completions. As described, this call can typically be performed from
uprovider without kernel involvement. However, polling for completions hogs the CPU, i.e. steals
other applications cycles or hinders the CPU to perform power saving, thus there is also an event
mechanism in OFED. It has one shot semantic, i.e. after requesting a notification the application is
notified once on a new work completion on the CQ. While waiting on a new event, the application
is put to sleep by kverbs and only woken up on an arriving completion. Notifying the kernel on
new completions is vendor-specific and is usually implemented with hardware interrupts. Despite
saving CPU cycles when using the event mechanism, its major drawback is that it significantly
adds latency, as shown in Section 5.

Although closely coupled, the verbs API does not offer connection management, therefore OFED
provides separate connection management components. Like the verbs API, it is split into a kernel
API (kcm) and a user API (ucm). It makes heavy use of the verbs API e.g. to get registered
providers or to manipulate the resources to reflect the connection state. In general, the network
stack is offloaded onto the RNIC and thus connection management is, like resource operations, ven-
dor specific. Therefore, a vendor has to specify connection management callbacks when registering
a provider to the kverbs. From a programmer’s perspective, the RDMA connection management
API is similar to the socket API.

Besides proprietary interconnects, RDMA is available on Infiniband (IB) and Ethernet [6][12]. IB
is a first order interconnect technology which provides point-to-point switching. The specification
includes, besides physical and link layer, its own network and (reliable) transport layer. RDMA
capability is built directly into the transport layer. In contrast, Ethernet only defines physical and
link layer and typically uses the Internet Protocol (IP) as network layer and the Transport Layer
Protocol (TCP) or User Datagram Protocol (UDP) as transport layer. Because these protocols
do not include RDMA support, a set of protocols called iWARP can be used to enable RDMA
on top of TCP/SCTP [29][33]. Besides TCP/IP with iWARP RDMA over Converged Ethernet
(RoCE) offers RDMA support by using IB network and transport layer on top of Ethernet aiming
to preserve IB transport services, protocol and software interfaces [16]. However, although not
specifically required, a loss-less Ethernet fabric is recommended.
2. Background

Application

Kernel Bypass
Kernel User

User

RNIC

ucm
uverbs
uprovider

kcm
kverbs
kprovider

Figure 2.2: OFED Stack

2.2 Virtualization

Virtualization is a layer of indirection between the hardware and software running on top of it [31]. This abstraction layer is called virtual machine monitor (VMM) or hypervisor and has been implemented in both software and hardware. A hypervisor can provide a uniform view of the underlying hardware, allowing to multiplex resources for guest operating systems (guest OS) running above. Further, it allows virtual machines to be encapsulated in such a way that they can be paused and resumed, replicated and migrated or moved to other hardware. It also provides isolation, making it possible to run untrusted software without compromising the underlying system.

There are two types of virtualization techniques: paravirtualization and full virtualization. While paravirtualization requires the guest OS to be virtualization aware, full virtualization requires no modification by providing a transparent view of the virtualized hardware. However, full virtualization typically requires software emulation in the hypervisor, imposing performance overhead, except where hardware virtualization support is available which enables full virtualization with partial or no emulation. Paravirtualization is typically used where hardware virtualization is not available or not feasible to gain the best performance and flexibility. In general, paravirtualized devices use a split driver model where a frontend driver is running in an (untrusted) guest OS and a backend driver is running in a (trusted) OS with direct hardware access[13][32]. Instead of emulating every access to the hardware, these drivers use a virtualization-aware communication channel.

CPU virtualization enables direct execution of CPU instructions. However, virtual machines are typically run in an unprivileged environment, thus not allowing to execute privileged instructions. One solution is paravirtualization using binary translation to replace unprivileged instructions. Another solution is to let the CPU exception handler catch unprivileged instructions and then emulate them in the hypervisor. However, as of today most modern processors support hardware virtualization, e.g. Intel VT-x, which enables execution of privileged instructions inside a virtual environment without trapping into the hypervisor or the need for translation [24]. Typically, the
2. Background

The hypervisor performs a VM-start instruction to bring the CPU in virtualization state. Whenever events occur that cannot be handled by the virtualized environment, like a non-virtualized instruction or interrupts, the CPU performs a VM exit, leaving the virtualization environment. These VM exits are typically costly (cf. Section 5) and should be avoided if possible.

Virtual memory systems have been used to allow applications to have their own isolated address spaces. Typically, the hypervisor uses virtual memory to encapsulate and isolate the virtual machine’s memory, i.e. guest physical memory. However, the virtual machine itself might want to use virtual memory by itself, i.e. the virtual memory system has to be virtualized. This can be implemented using a paravirtualized memory management unit (MMU) or by full virtualization using shadow page tables maintained by the hypervisor reflecting the mappings of the guest OS. That is, for every mapping in the guest OS page table there is a mapping in the shadow page table which is effectively used by the underlying hardware. However, this requires trapping into the hypervisor whenever a page table entry is changed or whenever page tables are switched. Therefore, CPU vendors like AMD and Intel have implemented hardware virtualization support for the virtual memory subsystem (AMD NPT [5], Intel EPT). This second level page table support allows guest OSs to have their own page tables without shadowing resulting in higher performance and without the need for guest OSs to be virtualization-aware [36].

Good I/O virtualization performance is essential to make virtualization environments feasible for I/O intensive workloads, e.g. high-performance networking or storage systems. Although, easy to install and flexible, emulation increases performance overhead significantly, as every access to the emulated hardware device has to go through the hypervisor’s software layer. Therefore, a common approach is to use paravirtualized I/O devices. For example, virtio-net is a paravirtualized network device used for the KVM hypervisor. Another option is to directly assign devices to a virtual machine, i.e. by allowing the VM to access the device’s registers or memory. However, these devices are part of the host physical addresses space, which poses a problem. For instance, direct memory access (DMA) capable I/O devices can perform memory operations on physical addresses by programming the DMA engine. If such a device were assigned to a VM, it would be programmed by a driver running in the guest OS. This driver would not be aware that the guest physical memory is host virtual memory and thus those DMA operations would not only write or read from the wrong location but compromise the system. This problem has been solved by introducing an I/O memory management unit (IOMMU) (Intel VT-d, AMD IOV), which resides between the device and the memory controller and allows translation of physical addresses[1]. The hypervisor programs the IOMMU to translate from guest physical memory to host physical memory for devices which are directly assigned. There are a few remaining problems with this approach: first, a device can only be assigned once, i.e. it cannot be multiplexed to be used by the host or any other guest OS. Second, migrating virtual machines is complex[20][11]. The first problem has been partially solved by introducing single-root I/O virtualization (SR-IOV) as an extension for PCIe [28]. This technology enables the hardware to virtualize itself by providing PCI virtual functions (VF) which can be assigned to a virtual machine. However, this requires not only the I/O hardware itself to be SR-IOV capable, but also the platform it is running on, cf. Intel VT-c. Furthermore, the number of
virtual functions that can be allocated is limited. Although virtualization-aware, SR-IOV devices do require an IOMMU for memory translation and protection.

2.2.1 Advanced Programmable Interrupt Controller

Interrupts on x86 are delivered through the advanced programmable interrupt controller (APIC). There are two kinds of APICs present in a system: an I/O APIC and a processor core local APIC (lAPIC). The I/O APIC serves as a multiplexer for delivering hardware interrupts to a lAPIC. Every lAPIC has an interrupt descriptor table (IDT) which is used to look up the interrupt handler routine for a specific interrupt which is executed when this particular interrupt is raised. For backwards compatibility there are at least three programming interfaces available for the lAPIC. However, the newest and recommended is the x2APIC, which is programmed via model-specific registers (MSR)\[18\].

To allow transparent virtualization of x86 hardware, virtual CPUs (vCPUs) need a virtualized lAPIC which can be programmed by the guest OS via one of the standard programming interfaces, e.g. x2APIC. Although Intel’s and AMD’s x86 processors have a variety of hardware virtualization features like virtualized privileged instructions, second level page tables and IOMMU (cf. Section 2.2), which are widely available even on consumer hardware, APIC hardware virtualization has just been introduced by Intel in their newest Xeon processor series (APICv). If hardware virtualization is not available, the hypervisor has to fully virtualize the lAPIC. For example, the Linux kernel virtual machine (kvm) uses an emulated x2APIC. We described in Section 2.2 that emulation typically requires exiting the virtualized environment to perform operations on the emulated hardware which runs inside the hypervisor. For example, if we want to inject an interrupt into the VM, we need the exit to set the interrupt vector and force the CPU into interrupt state when continuing execution. Furthermore, the interrupt needs to be acknowledged when handled for the x2APIC this can be performed by writing to the end of interrupt (EOI) MSR. Thus, interrupt injection requires two VM exits with an emulated x2APIC. However, because VM exits are so expensive, Linux has built-in support for paravirtualized end-of-interrupt MSR writes, i.e. interrupt injection can be performed with only one VM exit. Note that because of the nature of paravirtualization, this requires guest OS support.

Besides Intel’s APICv, there has been other solutions to reduce or remove VM exits due to interrupts, because they are the main source for overhead in I/O hardware virtualization\[3\]. For example, a purely software-based solution is ELI (ExitLess Interrupts) implemented for kvm, which uses a shadow interrupt description table (IDT) similar to shadow page tables [14]. However, exitless interrupts are limited to one virtual core per CPU core, which does not allow over-provisioning of exit-less virtual cores.
3 Design

Our main design goal is to allow applications running inside VMs to leverage the full potential of RDMA, i.e. ultra low-latency and high-throughput with low cpu overhead. However, as described above, we do not want to rely on I/O hardware virtualization support like SR-IOV, because their setup is inherently static. For example, [20] and [11] show that direct assigned device migration is complex. Furthermore, (de-)multiplexing and scheduling of resources in hardware to support SR-IOV is complex and adds more layers of indirection. In this respect RNICs (without hardware virtualization support) are predestined to be used in virtualized environments, because they already support a secure way of multiplexing resources between applications. To leverage this fact, we conclude that a paravirtualized approach (cf. Section 2.2), is the best fit. Furthermore, we wanted to extend the existing RDMA stack in a non-intrusive way to allow vVerbs to be easily pluggable. Note that we focus on bringing RDMA to guest user-space applications although our proposed design is applicable to kernel verbs.

The key idea of RDMA is to directly expose application buffers to the remote side to enable direct data access operations. Thus, we believe it is a requisite element of the design to allow application buffers inside the VM to be registered and directly used by the RNIC. The setup process of a connection and its resources has to go through the host kernel, as it requires direct access to the hardware and dealing with the memory management subsystem. In vVerbs we use a paravirtualized driver to forward control path operations. However, once set up, RNICs are programmed via their resources (cf. Section 2.1), e.g. posting work requests on QPs, or polling for completion elements on CQs. In the following we discuss two different general design alternatives, that differ in the way these network resources are programmed.

**Design 1** (cf. Figure 3.1(a)) To achieve the best possible performance, the aim of the system should be to bring network resources as close as possible to the application, i.e. mapping the resources into the VM. This approach can completely bypass the hypervisor; we call it design 1. However, because of the way these resources are programmed, bringing the resources into the virtual environment exposes its underlying hardware. This might limit the system’s flexibility, e.g. support for migration.

**Design 2** (cf. Figure 3.1(b)) suggests to not expose the resources, but leave them in the host and make them uniformly accessible via the paravirtualized control path. Design 2 cannot completely
bypass the hypervisor on data pass operations, i.e. it does support direct data access, like design 1, but instructing the RNIC to do so via its resource means going through the slow control path into the host kernel, which introduces performance overhead.

We believe that to reach our performance goals and eventually to stand out from other solutions (see Section 6) design 1 is the best approach. We show in Section 7.1 how we can still provide flexibility, e.g. to support migration, with this solution. We choose design 1 as our general design, thus, all subsequent design alternatives, namely A, B and C, implement design 1.

For our second design goal we decided to extend the OFED stack, preferably in a non-intrusive way. A possible design could use unmodified user libraries and kernel modules. We choose the OFED stack because it is the de-facto standard API used to program RDMA applications (see Section 2.1). In the following, we discuss different designs of how to extend the OFED stack to a paravirtualization-enabled RDMA stack. We further assume that the paravirtualized driver uses a split driver model (see Section 2.2), and that its backend driver runs in privileged mode to gain the best performance [30]. We call the frontend driver 'gkvirtverbs' and the backend driver 'hkvhostverbs'. As described above, design 1 has the drawback of exposing the underlying hardware to the VM, therefore all subsequent designs need some hardware-specific part to program the RNIC’s resources. The connection management components are left aside and discussed later in more detail. An overview of design alternatives, that extend the OFED stack, is given in Figure

---

1To not get confused about which component runs where, every component that runs inside the guest OS is prefixed with a "g" and every component running on the host is prefixed with a "h". Further, "k" and "u" are used to differentiate between kernel- and user-space.
3. **Design**

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>hkvhostverbs</td>
<td>paravirtual backend driver</td>
<td>host kernel</td>
</tr>
<tr>
<td>gkvirtverbs</td>
<td>paravirtual frontend diver</td>
<td>guest kernel</td>
</tr>
<tr>
<td>hkverbs/gkverbs</td>
<td>OFED Verbs core and kernel Verbs</td>
<td>host kernel &amp; guest kernel</td>
</tr>
<tr>
<td>hkprovider</td>
<td>Verbs provider</td>
<td>host kernel</td>
</tr>
<tr>
<td>gkprovider</td>
<td>Virtualization enabling Verbs provider</td>
<td>guest kernel</td>
</tr>
<tr>
<td>guverbs</td>
<td>OFED user space Verbs library</td>
<td>guest user space</td>
</tr>
<tr>
<td>guprovider</td>
<td>user space component of hkprovider</td>
<td>guest user space</td>
</tr>
</tbody>
</table>

Table 3.1: Paravirtualized RDMA stack components

**Design A**  In design A (cf. Figure 3.2(a)) the guest OS uses unmodified verbs components, i.e. guverbs and gkverbs are used as is. Furthermore, the original, unmodified uprovider is used. However, we cannot use the original kprovider, as it assumes direct access to the underlying hardware, e.g. to the PCI-bus. Note that the uprovider is also capable of accessing the device directly, however, the interface used to program the RNIC in user space is restricted to mapped resources. We describe later how these resources can be mapped into the virtual environment. In contrast, the RNIC cannot be shared by multiple kproviders and thus allowing direct access from the guest OS is not possible or requires the host to give up ownership of the device (cf. device pass-through in Section 2.2). Therefore, in this design we need a “fake” gkprovider which mimics the original provider such that the original guprovider can be used. The gkprovider uses the gkvirtverbs’ API to forward its calls to the host. How these calls are executed on the host is discussed later in detail. The drawback of this approach is that migration to a host running different hardware is difficult, as we cannot control the loaded guprovider. We show in Section 7.1 how this could be solved.

**Design B**  (cf. Figure 3.2(b)) replaces all verbs components except the uprovider in the guest OS, i.e. guverbs is replaced by the virtguverbs component, which directly forwards the calls to gkvirtverbs. Furthermore, virtguverbs exposes the same interface as guverbs and thus the original guprovider can be used. In this case virtguverbs fakes exposed hardware interfaces to allow using the unmodified guprovider. We believe that the biggest drawback of this design is that user applications have to be modified\(^2\) to link against the replaced guverbs component. Another drawback of this approach is that we have to design a new user-to-kernel interface to forward the calls from the virtguverbs to the gkvirtverbs, while with design A we can leverage the existing interface by not replacing the gkverbs component and forwarding the calls via a virtual kprovider.

**Design C**  (cf. Figure 3.2(c)) uses the existing verbs components, but requires the kprovider and its corresponding uprovider inside the guest to be virtualization aware, i.e. they have to be rewritten or extended, which requires comprehensive understanding of existing provider code.

---

\(^2\)Modification might be overstated, typically the uverbs is loaded dynamically and thus can be replaced by modifying the library search path. Nevertheless, user interaction is required.
We believe that design A is the best of the described approaches. It allows us to use all verbs components and the uprovider without modification by adding a small layer of redirection in the kernel, mimicking the kprovider (see Section 4).

As described above, control operations are forwarded over the paravirtualized drivers, because they require direct access to the underlying hardware; this allows us to isolate VMs. On the host, the control operations are first processed by the hkvhostverbs backend driver, which makes the commands virtualization-aware. Eventually, these commands have to be processed by vendor specific code to program the RNIC. Keeping the second design goal in mind, we want to use the existing hkverbs and the unmodified kprovider. Therefore, control path calls are processed by the hkvhostverbs, altered if needed, and then forwarded into hkverbs.

We forward control path operations to hkverbs, as they come from a host user-space application using the uverbs API. Note that we cannot use the kverbs API together with a user space applications because, for example, resources might be different for uproviders. However, this requires altering the commands and additional processing to use the OFED core as-is. For example, when a user-space application wants to register memory, it passes a virtual address and size to uverbs, which forwards the command to kverbs. Registration has to be performed in the kernel because it requires pinning the physical memory such that it can be used in I/O operations (e.g. is not swapped out). Additionally, together with the virtual address, these physical pages are stored on the RNIC to support direct data access without kernel involvement and thus require programming the hardware directly.

In vVerbs we cannot use the virtual address provided by the guest user-space application as-is on the host, because the virtual address space does not exist on the host and thus pinning the memory
would fail (or lead to undetermined behavior). However, the guest physical address (GPA) space is
directly represented in host virtual memory and can easily be translated using an offset. Therefore,
when a memory registration is performed from the uverbs API in the guest user-space we pin its
underlying memory in the gkvirtverbs driver and pass the physical pages to the host. We can then
translate these GPAs into HVAs. However, there are two problems with this approach. First, as
described above, kverbs expects one VA but the pinned pages in the guest are most likely non-
contiguous and thus are non-contiguous in the HVA space, resulting in multiple memory regions.
Second, remote memory operations use the virtual address as an offset into the registered buffer
and are thus to be checked by the firmware of the RNIC when performing direct data access. The
application typically sends the memory buffer information, i.e. key and offset, required for access
over an arbitrary channel to the remote side, thus we cannot easily control which addresses it
advertises. Now the problem is that, even if we had a contiguous virtual memory region, its virtual
address would still not be equal to the one advertised respectively used by the guest application
and its remote side.

A solution addressing both problems is to have virtual address spaces reflect the registered buffers
in the guest, i.e. for every memory buffer registered with guest virtual address A and its guest
physical pages B, there has to be a host virtual address space where a virtual memory region
starting from address A points to physical pages C, which backs guest physical pages B. However,
this approach is cumbersome as is requires setting up new virtual address spaces, in worst case for
every user application in any guest. Furthermore, this not only degrades performance significantly,
but adds additional complexity for switching to and from those contexts. We are not satisfied with
this solution and will later look into different approaches that work without modification of the
verbs components.

Going back to our first problem, i.e. mapping the GPAs contiguous in HVM. This is achievable by first pinning host virtual memory regions representing these guest physical pages. We can then map the underlying host physical pages to a contiguous host virtual memory region and use its start address for registration by kverbs. We visualize this approach in Figure 3. For the second problem, there is an elegant solution when using the OFED stack, i.e. the verbs API internally already uses two kind of addresses. One address is used to pin the memory and get its underlying physical representation, the other address is used for the I/O operations, i.e. stored on the RNIC. Typically, these addresses are equal, but we can leverage their existence by providing an address to the contiguous host virtual memory representing the guest virtual memory buffer and its GVA.

As described in Section 2.1, resources are typically shared with the user-space application as it enables kernel bypass for performance critical data path operations. Sharing is achieved by having a memory region accessible from the user-space and the RNIC directly, e.g. DMA-enabled pages or PCI I/O memory, which are mapped into the application virtual memory space. We see two different approaches to setup resources for sharing in OFED providers: Allocate resource memory in user space by the uprovider, then pin and install them to the RNIC in the kprovider or allocate and setup in the kprovider and map them to user-space.

Because we want to leverage the full potential of RDMA, we want to have the resources directly accessible by an application in the guest (cf. final design). In the following we discuss how to share resource when set up with one of the two approaches described above. Allocation in the (unmodified) guprovider is similar to memory registration, the hkverbs expects a HVA of a contiguous memory region holding the resource. However, in this case we do not have to worry about which VA address is used because it is only used once to pin the underlying physical pages. Therefore, we can use the technique described for memory registration to contiguously remap the memory.

For allocation in the kprovider, resources are mapped into VA space. In vVerbs, that means
mapping it into guest virtual address space. To achieve this mapping without modification of the kprovider, we first map the resources into HVM (1) (cf. Figure 3). From this mapping we extract the underlying physical pages (2). When the upprovider asks for mapping a resource (3) we forward the command from the gkprovider to gkvirtverbs where we allocate and map (empty) physical pages (4). Furthermore, we forward the list of physical pages to the hkvhostverbs, where we translate their GPAs to HVAs. We then destroy the existing mapping to these HV pages (5) and map them to the extracted physical pages of the resources (6).

The RDMA connection management (cm) component in the OFED stack is, like verbs, split into a user API (ucm) and a kernel API (kcm). As described above, we focus on user applications and thus our first attempt to paravirtualize cm is to simply forward the ucm kernel calls from the guest to the host kernel. This forwarding is performed over a separate paravirtualized driver. The disadvantage of this approach is that cm takes place completely on the host, e.g. IP addresses for VMs have to be registered on the host, routes are determined by host routing tables, and port space of the host is used. While this works sufficiently well for simple setting, it is not flexible and does not give the VM a transparent view of its RNIC. Furthermore, having socket connections besides RDMA connections over the same IP address, e.g. when using iWARP or IP over Infiniband, is not directly supported by this approach, because we do not expose the RNIC’s socket network interface to the virtual machine. Thus, socket connections have to be performed over a virtualized NIC, which, if in the same subnet, cannot share the IP which is used for RDMA connections by the host interface. In Section 7.2, we propose a cm design which allows transparent cm in the VM.
4 Implementation

For our prototype implementation, we used the kernel virtual machine (kvm), which is a hypervisor running inside the Linux kernel[21]. As frontend to kvm we used Quick EMUlator (QEMU)[8].

QEMU/kvm offers a framework to build paravirtualized drivers called 'virtio'[32]. Similar to Xen’s paravirtual drivers they are split into a frontend and a backend driver [13]. The frontend driver runs in the guest OS and its backend driver on the host. In virtio, buffer queues are used to communicate between frontend and backend. For every communication channel there are two queues: an available queue and a used queue. New buffers can only be added by the guest on the available queue. The guest then typically notifies the host that new buffers are available. The host can retrieve those buffers from the available queue and place them on the used queue when done. Further, the host can then notify the client that the buffers have been consumed and put it on the used queue, where the guest can then retrieve them again. There is no copy performed in delivering the buffers to the host; instead buffers in the guest have to be allocated in contiguous guest physical memory. This allows the host to translate them into contiguous host virtual memory and directly access them. Because of the way virtio handles its buffers, there is no way to add new buffers on the host side. To ease setup and allow use of interrupts on notification virtio exposes a PCI device to the guest1, i.e. the frontend driver is a PCI device driver and thus running in the guest OS kernel. On the host the backend driver typically runs in user-space inside QEMU. However, to gain better performance a backend driver, called ‘vhost’, can be implemented in the kernel. We implemented a vhost backend called ‘vhostverbs’ and a frontend driver called ‘virtverbs’ for vVerbs.

kvm uses second level page tables if available (see Section 2.2). As described in Section 3, to allow mapping of resources to the guest, we have to modify the mappings on the host. But changing the page table on the host is not sufficient; the mapping has to eventually be reflected in the second-level page table. kvm uses memory mapping notifiers to automatically update the second-level page table. This allows us to change the mappings without bothering about virtualization mapping details.

We use the OFED RDMA stack for our prototype (see Section 2.1 and 3). Our idea is to forward control commands over the virtio framework to the host. Therefore, the virtverbs driver provides an API similar to the verbs API to forward the calls. Furthermore, the virtverbs API provides some additional commands for vendors to implement their specific features. The vendor only has to provide a kprovider for the guest, exposing its features in such a way that the (unmodified)

1There is a I/O memory implementation for guests that do not support PCI.
Implementation

The gkprovider running in the guest believes it is running on real hardware. All verbs calls ending in the gkprovider are simply forwarded to the virtverbs API, where the commands are enabled for virtualization. Thus, the gkprovider is only a thin layer, where each verbs call can be implemented typically in less than 60 lines of code. For example, it took us only 3 days to implement the cxgb4 gkprovider (cf. Section 4.2). This is all that needs to be done from a vendor perspective to enable their devices for paravirtualization. As discussed in Section 3, on the host side we mimic a user-space application in vhostverbs, i.e. we forward the commands to kverbs, adjusted with the necessary modifications for enabling virtualization.

In Section 2.1 we briefly discussed how the completion event mechanism works. Typically, it is used in a hybrid form together with polling to allow for small latencies and thus is a performance critical operation. In our virtualization enable RDMA stack, polling adds zero cost as it can be directly performed on a mapped CQ inside the guest. However, the events mechanism has to go through the whole stack from host to guest and thus is rather costly (see Section 5). To deliver the events as fast as possible, we implemented a dedicated event channel. On setup of the device, we install a event queue via a shared memory region between guest and host. Additionally we allocate a separate virtio channel for notification of the guest. We do not use virtio buffers to implement the event queue because it requires preposting the buffers from the guest, as adding buffers on the host is not possible. We override the event callback of the CQ objects in hkverbs with our own callback function (cf. Section 5.3). That way, on a completion event the RNIC’s driver directly calls into the vhostverbs driver instead of going through kverbs. This functions then adds the event information to the event queue and notifies the guest via the virtio notification mechanism, which is implemented by injecting interrupts into the virtual machine. When our event handler, which serves these interrupts, is called in the virtual machine, we retrieve the event information from the event queue and lookup the associated CQ in a hashmap, where we store all CQs created in the guest. We can then call the CQ’s event callback in the guest, which eventually wakes up the applications.

4.1 SoftiWARP

SoftiWARP is a OFED provider which implements the iWARP protocol in software [34]. It runs on top of TCP kernel sockets. Like most hardware solutions, it shares its resources with user-space applications for best performance. However, because it relies on kernel sockets, it does not support zero copy on the receive side. The benefit is that it does not require special hardware to run on and thus enables RDMA in heterogeneous infrastructures.

For our first gkprovider implementation we choose SoftiWARP because it is easier to debug than hardware accelerated solutions and does not deal with hardware memory mappings, like PCI I/O memory. Note that one could run SoftiWARP without modification inside the virtual machine on top of a virtualized network interface. However, default VM setups typically use a paravirtualized network interface, like virtio-net, which forwards all its packets over virtio queues, adding significant overhead, e.g. because of additional copies. In contrast, paravirtualized SoftiWARP enables data
to be directly copied from the host kernel socket buffer into the guest’s application buffer.

In SoftiWARP the kernel has to be notified after posting a work request on a work queue to actually perform the operation. This notification is called 'doorbell' mechanism. In contrast to hardware solutions were the RNIC’s doorbell is typically implemented as an PCI I/O memory region, which is mapped to user-space, i.e. does not require kernel interaction on notification, SoftiWARP uses an otherwise unused command of the verbs API to notify the kernel. Although this works out of the box with vVerbs, it adds significant latency (cf. Section 5), as we have to go to an additional layer forwarding the command to the host.

### 4.2 Chelsio Terminator 4

The Chelsio Terminator 4 (T4) is a 10Gb/s Ethernet adapter which offers a wide variety of offloading features, like TCP offloading, iSCSI and iWARP (cf. Section 2.1). T4 allocates its shared network resources in the kernel, thus requires applying the remapping technique described in Section 4. Furthermore, T4 uses PCI I/O memory for its doorbell mechanism, which can be easily mapped like any other resource mapping through the gkvirtverbs API. Other than that, the T4 driver uses a shared status page, which is allocated in user memory and later pinned by the driver to expose hardware features like a direct mapped doorbell. To map this status page, we need a contiguous remapping, like for user-memory registration. Therefore the gkvirtverbs API exposes a `mmap_contiguous` function to map the status page. Implementation of the gkprovider to enable paravirtualization for T4 thus only requires calling gkvirtverbs framework functions. Thus, it took us only 3 days to implement the gkprovider.
5 Evaluation

We evaluated vVerbs by measuring latency and throughput. In these benchmarks we compare VM-to-VM performance to host-to-host performance. We also compared our solution against I/O hardware virtualization. All tests were performed on two nodes of an IBM Flex System with 2x Intel Xeon E5-2690 (20M Cache, 2.9Ghz, VT-x & VT-d enabled) CPU, 94GB RAM, Chelsio T420-CR 10GbE RNIC. The T4 RNICs were connected to an IBM G8264 Switch (According to its specification, this switch adds 880ns latency). Chelsio's T4 RNICs uses iWARP technology to enable RDMA capability (cf. Section 2.1). Although SR-IOV is supported by T4, RDMA on a virtual interface is not. To compare our solution against hardware virtualization, we used PCI pass-through which should in theory offer similar performance to an SR-IOV enabled RNIC. Furthermore, we compared vVerbs against typical virtual network stacks, i.e. paravirtualized NIC (virtio-net) and SR-IOV by measuring their TCP performance. Rsockets\(^1\), a socket implementation on top of RDMA was used to show how applications programmed against the socket API can benefit from virtualized RDMA \([15]\).

On the host we used Debian Linux running a 3.7.10 vanilla kernel with kvm provided as-is. As frontend to kvm we extended QEMU\(^2\) with our paravirtualized drivers with vhost backends (cf. Section 4). As guest operating system we also used a Debian Linux running a 3.7.10 vanilla kernel. The system described above has non-uniform memory access (NUMA), i.e. there are two NUMA nodes, one for each processor. We forced the interrupts and applications to run on one node only to avoid NUMA effects in our benchmarks. Intel Speedstep technology, which allows the processor to perform frequency and voltage scaling for power saving, was disabled for all tests. Furthermore, we changed the maximum transfer unit of all NICs (T4, pass-through T4, SR-IOV T4, virtio-net) involved in benchmarks to 9000 bytes to keep the protocol overhead low. Other than that we used default configurations to perform the tests.

We extended netperf to allow measuring RDMA latency and throughput for read, write, and send/receive operations. TCP measurements were performed with (unmodified) netperf. Furthermore, we used a 20 second test length for our measurements and every test was performed 5 times. The verbs API allows the user to specify whether a work request on the send queue should generate a work completion (signaled) or not (receives are always signaled). We performed all our tests with signaled work request because otherwise we would need tricks like zero-length read after the

\(^{1}\)git://git.openfabrics.org/~shefty/librdmacm.git commit: 984b1e3c189db9d156ea429c1726bd8739893247
\(^{2}\)git://git.qemu-project.org/qemu.git commit: 0706f7e85b3c0783f92d44b551f362884db0f4bd
operation to determine its actual completion. Nevertheless, when using signaled work requests, one has to decide whether to poll for completions or use the completion event mechanism described in Section 2.1 and 4 (or use a hybrid approach). To get an understanding of how this affects performance, every RDMA test was evaluated with the event mechanism and without, i.e. polling only.

The following configuration were used for measurements:

**Host-to-Host**
- t4 - Chelsio T4
- siw - SoftiWARP
- rst4 - T4 with rsockets

**VM-to-VM**
- pt4 - Chelsio T4 with PCI pass-through
- vt4 - vVerbs with T4
- vsiw - vVerbs with SoftiWARP
- virtiot4 - virtio-net paravirtualized network interface
- sriovt4 - T4 SR-IOV virtual device
- rstvt4 - vVerbs with T4 with rsockets

### 5.1 Latency

The RDMA read operation is the preferred operation to measure latencies because it has to wait until the data is read on the remote side, transferred back to the host and placed in the application buffer before adding a work completion element to the CQ.

In Figure 5.1 we show read latencies of a 4 byte and a 16kB read on a remote system with use of the event mechanism. Note that the read operation is a one-sided operation and thus the remote side’s kernel/application is typically not involved (with exception to SoftiWARP). Although we cannot achieve native T4 performance, we are able to compete against the PCI pass-through T4. With vVerbs we can even show slightly better performance for 16kB reads. For 4 byte transfer size, t4 achieves 17 usecs latency, pt4 23.8 usecs and vt4 23.6 usecs, i.e. adding approximately 40 percent to the latency. The reason that both our solution and PCI pass-through show similar performance is due to the cost of VM exits on work completion notifications. In the pass-through solution, these exits are introduced by forwarding device interrupts of the pass-through device to the virtual machine. It has been shown that these exits have a significant impact on the performance of pass-through devices (note that this also applies to SR-IOV solutions, as those use the same technique to assign their virtual functions to VMs) [10]. In vVerbs, these exits are introduced by the notification mechanism of virtio, which injects interrupt into the VM (see Section 4). For SoftiWARP there
are additional VM exits for notification of posted work requests (cf. Section 4). Thus, it has to go through the slow control path after adding the work request to the QP, i.e. processing only starts after notifying SoftiWARP running on the host. And this is why the performance difference compared to native SoftiWARP is much higher than t4 vs. vt4. For example, for 4 byte reads siw achieves 27 usecs latency and its virtualized counterpart 36.3 usecs, which is a 34 percent increase in latency. While we expected to see a constant overhead introduced by injecting interrupts, the numbers for 16kB reads show more than 2 times increase of virtualization-added overhead. For siw the increase is even more significant. After testing with different ways of injecting interrupts and scheduling of the virtual machine we were not able to show better results. However, initial tests with disabled CPU C-state power saving showed that the overhead introduced is indeed constant and estimates to what is shown for the 4 byte transfer size. Thus, the higher the transfer sizes, i.e. the higher the latencies, the less impact the virtualization should have. Nevertheless, we need to investigate further how exactly C-states affect latencies. We show in appendix A wake-up latencies for the sandy bridge architecture (micro-architecture of the Intel Xeon E5-2690 used for our benchmarks), that should give the reader an idea of how these can affect latencies. Keep in mind that these power saving mechanisms are typically enabled on systems, even if frequency scaling is disabled and cannot be disabled dynamically but requires a restart of the system. After all, the event mechanism might be used with the purpose to save power thus disabling this power saving mechanisms would not be acceptable. It would be interesting to evaluate how different C-states affect interrupt latencies, however, this is beyond the scope of this thesis. In Section 5.3 we look at interrupt injection in detail.

Figure 5.2 shows latency for 4 byte and 16kB RDMA read operations with polling for work completions. For T4 both vVerbs and the pass-through device show bare-metal performance, i.e. 9.7 usecs for 4 byte and 32 usecs for 16kB reads. This is due to the fact that all operations are performed only on shared resources, respectively involved application buffers and thus there is no interrupt or
kernel involvement. As described above, vsiw has to go through the kernel control path for ringing the doorbell and thus there is a performance penalty even when polling for completions. For 4 byte reads siw achieves 18 usecs and vsiw 22.4 usecs, which is a 24 percent increase in latency. However, the db ringing overhead is introduced only at the beginning of an operation and thus there is, especially for larger transfer sizes, enough time to continue VM execution, i.e. be in polling mode when the new work completion is added to the CQ. So for 16kB reads the performance penalty shrinks to 1 percent. Note that polling numbers are not affected by C-state power savings as the CPU is fully loaded while spinning, which prevents the Linux idle driver from going into power saving states.

The RDMA write operation reads data from the application buffer and stores it in an application buffer on the remote side. In contrast to the read operation, work completions can be added before the operation is performed on the remote side, i.e. directly after reading everything from the local buffer. The reason behind this is that the rest of the operation can be performed without the involvement of the application respectively its buffers and thus the operation is completed from an application perspective. However, this can lead to unexpected performance numbers for write latencies.

Figure 5.3 shows latency of 4 byte and 16kB writes with use of the event mechanism. SoftiWARP write latencies, 1.5 usecs for 4 byte and 13 usecs for 16kB, are quite low compared to read latencies, where they were considerably higher than T4. This is due to the fact that siw copies the local data to the tcp kernel socket before sending, which allows to directly add a work completion after copying. siw has to copy the data because it does not support zero copy write with signaled writes (yet), which would require siw to wait for the corresponding TCP acknowledgement before releasing the buffer, i.e. adding a work completion. In contrast, t4 has to wait until all local data is read via DMA and successfully sent before adding a work completion. Therefore, write latencies for t4 are 7.6 usecs for 4 byte and 19.1 usecs for 16kB. In contrast to read latencies the overhead introduced
by virtualization of t4 is similar between 4 byte and 16kB transfer sizes, e.g. for vt4 7.9 usecs versus 6.5 usecs. Again vt4 performs slightly better than pt4, we discuss in Section 5.3 where this difference originates from. As for read latencies, vsiw suffers from VM exits on posting work requests (besides completion events), which adds significantly to the latency.

Figure 5.4 shows write latencies with polling. Again t4, vt4 and pt4 show equal performance results, 2.8 usecs for 4 byte and 13 usecs for 16kB, because they only operate on shared resources and application buffers without kernel involvement. The overhead of vsiw is still significant due to post exits.

The send/receive latency test is a ping/pong test, i.e. a RDMA send is performed and as soon as the remote side receives the send it initiates a send back. The initiator always waits until it receives
the response before it sends a new message. A RDMA send operation reads from a local buffer and writes to the next available buffer on the receive queue of the remote side. Remember that we use signaled sends which immediately notify the application after reading the local buffer. Thus, we have two interrupts on each side on every transaction, one for the send and one for the receive operation.

Figure 5.5 shows send/receive latency with use of events. For t4 the graphs show constant performance degradation of approximately 13-17 usecs in the virtualized settings up to 2kB. After that, it increases to 70-80 usecs for 8kB to 16kB. Again, initial tests showed that this is related to C-state power savings adding extra latencies after a certain threshold of idle time between interrupts.

Figure 5.6 shows send/receive latency with polling. Like read latency with polling, both pt4 and vt4 show equal performance compared to native t4, e.g. 13 usecs for 4 byte and 47.4 usecs for 32kB. Again, this is due to the fact that resources are mapped into the VM and thus there is no overhead introduced while posting sends or polling for completions. For vsiw we cannot achieve bare-metal performance because posting sends needs to go through the slow control path (cf. explanation above). In this scenario, one can see that the virtualization overhead is indeed constant, because the CQ polling forces Linux to not use power saving C-states.

To get an understanding how the traditional network stack performs against our solution we include TCP request/response latencies (see Figure 5.1). This benchmark is similar to RDMA send/receive shown above. Furthermore, we include rsockets performance for t4 and vt4. Rsockets uses a credit based control flow system to allow a socket interface on top of RDMA-enabled buffers. The default buffer sizes for send and receive is 128KB. To allow low-latencies, rsockets uses a hybrid polling approach to wait for work completions, i.e. rsockets polls for some time, if there were no completions during this period it waits for an event. Default polling time is 10 usecs before switching to events. This approach avoids using the event mechanism, at least for small transfer sizes. However, even for
small transfer sizes, sometimes the event mechanism is used, e.g. due to scheduling. We know from previous benchmarks that events add significant overhead for vt4, so we should try to avoid them. However, in this setting, once switched to events it is more likely that on the next wait we will again use events. To accommodate for this we increased the default polling time for both rst4 and rstvt4 by 10 usecs to 20 usecs. We believe that this is still an acceptable value not using too much CPU cycles when there are no completions for longer periods. Keep in mind that theoretically we should reach send/receive latencies, should we set the polling threshold to an infinite amount of time.

First, let us compare the default socket numbers only. T4 compared to send/receive (with events) is approximately 1.5x slower for 4 byte transfer sizes, 28.9 usecs versus 41.7 usecs. For one, this is because the offloaded network stack is highly integrated in hardware and thus considerably faster than Linux’s network stack. Further, the NIC issues an interrupt for every packet (if interrupt coalescing is disabled), thus at least for larger transfer sizes adding significant interrupt processing overhead. However, we explain in Section 5.3 how Linux tries to avoid that with NAPI by switching to a polling mode when interrupt load is high. The paravirtualized network interface, virtio-net, shows the highest latencies, e.g. 145.8 usecs for 4 byte up to 199.4 usecs for 8kB. SR-IOV t4 shows only slightly better performance than virtio-net. We explain this overhead by the fact that virtio-net has to go through the Linux network stack two times, i.e. once on the host because it has to be bridged to a real NIC and thus Linux has to decide to which interface the packet belongs, and once in the virtual machine. Remember that all these benchmarks are performed VM-to-VM, thus doubling the overhead when performing two-sided operations.

Rsockets with T4 gains lowest latencies for all transfer sizes, 12.7 usecs to 86.5 usecs. Up to 1kB rst4 manages to stay in polling mode most of the time, i.e. latencies are below 20 usecs. rstvt4 cannot achieve as low latencies as rst4 because of scheduling. Rsockets sometimes switches to events even if average latencies are below 20 usecs. And thus, for transfer sizes larger than 64 bytes, rsockets
reaches the polling threshold most of the time, switching to events in the virtualized environment. Nevertheless, we can show the lowest socket latencies with a purely software-based virtualization approach.

5.2 Throughput

Our netperf RDMA read and write throughput test posts work requests in a configurable, fixed size number of batches before collecting their work completions. The bandwidth delay product determines how much data we have to put on the wire to reach line speed. So for small transfer sizes this is limited by how fast work requests can be posted by the application and processed by the RNIC. Another limiting factor for small transfer sizes is the protocol overhead. So theoretically when the batch size is high enough, the throughput test becomes uninteresting because posting work requests has zero overhead (except for siw). Thus, we choose an unconventional throughput test by limiting the batch size to 10. With this test we can see how the event mechanism overhead affects throughput performance, i.e. the send queue is not filled at all time. Note that SoftiWARP is more affected by these small batch sizes than T4 because of its doorbell mechanism and its higher latencies and with higher batch sizes siw can reach line speed between 8kB and 16kB transfer sizes (cf. [34]).

Figure 5.2 shows read throughput with use of events. While T4 reaches line speed at 2kB transfer size, pt4 and vt4 reach line speed at 4kB. As with previous results, both virtualization solutions gain similar performance. SoftiWARP struggles with the small batch size, thus reaching line speed at 32kB. vsiw shows only slightly lower results.

Figure 5.2 shows read throughput with polling. Without events, performance is determined by how fast work requests can be posted on the send queue and how fast they can be retrieved from the
Figure 5.8: Read Throughput

Figure 5.9: Read Throughput with polling

completion queue by polling. Thus, t4, pt4 and vt4 show equal performance numbers reaching line speed at 2kB. SoftiWARP is able to gain better results starting from 1kB transfer size, however, it still struggles with small batch size because of its doorbell mechanism, which limits how fast work requests can be posted on the send queue. Here vsiw cannot follow closely, as interrupt injecting, because of its doorbell mechanisms, limits posting performance significantly.

Figure 5.2 shows write throughput with events. We showed above that write latencies are considerably faster than read latencies. This is reflected by these numbers showing that t4 can reach line speed between 1kB and 2kB. Also, siw performs much better gaining line speed at 16kB. In contrast, vsiw is not able to deliver line speed even with 32kB message sizes. At this point we do not know were this originates from, because posting work requests for both read and write should
have the same overhead. It cannot be explained with low write latencies delivering completion events so fast that posting is disrupted and thus taking longer, because looking at Figure 5.2 shows similar numbers with polling. Nevertheless, vt4 and pt4 show line speed at 2kB.

Figure 5.2 shows write throughput with polling. Again t4, pt4 and vt4 show equal performance reaching line speed between 1kB and 2kB. siw performance is slightly better, reaching line speed between 8kB and 16kB. However, vsiw shows similar behavior as with events, thus we conclude that the overhead must come from its doorbell mechanism.
5.3 Interrupts

We showed in the previous sections how interrupts affect VM I/O performance. In this Section we show the detailed Linux interrupt stack and look at how interrupts are injected into the virtual machine.

As described in Section 2.2.1, in case there is no hardware APIC virtualization available, like on our system, the APIC is emulated in kvm and requires VM exits on interrupt injection. Our host kvm version and the guest OS support the described paravirtualized end-of-interrupt write and thus we can inject an interrupt with one VM exit. Nevertheless, we believe that exit-less interrupts will be available soon in processors from all vendors and thus both vVerbs and pass-through will deliver bare-metal performance inside the VM when using interrupts.

In the following we analyze the Linux interrupt performance and interrupt injection for completion events on T4 in detail. The T4 main driver is `cxgb4`, which sets up the hardware and provides the standard NIC features without offloading. `iw_cxgb4` is used to enable RDMA. It heavily relies on `cxgb4`. For example, interrupts of completion events are processed by `cxgb4` first and eventually forwarded to `iw_cxgb4`. For all interrupts, Linux’s New API (NAPI) is used, which implements a hybrid polling approach to mitigate interrupt overhead for network I/O, i.e. when interrupt load is high it switches to polling mode. Therefore, the NIC driver has to provide a poll function to check for new packets/completions without using interrupts (and allow to disable receive interrupts). In Figure 5.12 we show how completion interrupts are forward in Linux to `cxgb4` and `iw_cxgb4` and eventually end up in user space as a CQ event. Note that the Figure only shows relevant functions in the call stack. NAPI uses software interrupts (softirq) to balance interrupt load on all CPUs, i.e. typically the I/O APIC is programmed once to deliver an interrupt to a specific core, with NAPI the hardware interrupt handler (1) raises a software interrupt (2) which can be delivered to any core depending on the load. The softirq handler eventually calls the poll function `napi_rx_handler` of `cxgb4` and forwards the event to the `c4iw_ev_handler` of `iw_cxgb4`. This function calls the CQ’s completion handler, which was previously installed by the kverbs. In the non-virtualized environment the callback function `ibuverbs_comp_handler` adds the event to an event queue and wakes up all threads waiting for new events. In this example we assume that an application is already waiting (3) with the `ibv_get_cq_event` call (4). After being woken up, the kernel thread (3) retrieves the first element on the event queue and returns to user space (4) where the application can then poll for the completion.

In the virtualized environment we have to deliver the interrupt into the virtual machine. We achieve this by overriding the CQ’s completion handler on the host with our own `vhost_verbs_comp_handler` (a), which adds the event to the shared event queue discussed in Section 4, and schedules work to signal the guest. The vhost worker thread executes the worker function `vhost_verbs_send_event` (b), which uses the irqfd mechanism of kvm to inject an interrupt into the VM (c). As described above, this process (c) to (d) requires a VM exit. After continuing the VM execution, the interrupt handler `vring_interrupt` of virtio is called (d), which eventually calls the callback `virtverbs_ack_verbs` of our virtio frontend driver. This function retrieves the event from the shared event queue and
calls the completion handler of the CQ installed by the gkverbs. Further processing continues like on the host with (3) and (4).

Figure 5.13 shows how long it takes to deliver an interrupt on the host compared to the guest. Numbers and letters of the labels refer to Figure 5.12. Theoretically, the host numbers (left), e.g. added to polling latency numbers should add up to event based latencies. We showed in Section 5.1 slightly higher numbers due to the fact that netperf latency measurements include posting work requests and retrieving work completions. Furthermore, it takes the hardware some time to actually raise the interrupt and the APICs to process it.

In the virtualized environment (right), injecting the interrupt together with walking the guest interrupt stack takes up approximately 10 usecs. This is the main source of overhead introduced on the event path.

We believe that there is still room for optimization in the event path. For example, like NAPI, we could use a polling approach in the guest if interrupt load is high, thus instead of injecting interrupts for every event we poll as long as we are over a certain threshold of interrupts.

5.4 Comparison of Network Virtualization Techniques

In this Section we compare different network virtualization techniques and solutions by their feature set. We include features which we believe are important to deliver good I/O performance or show the flexibility of the proposed solution. Table 5.1 shows the features and if they are available for
a specific technique or solution: **direct data access** allows a NIC to directly access application buffers, **direct application access** enables an application to get direct, isolated access to a NIC, **multiplexing** allows a NIC for sharing single hardware instances by multiple machines, **hardware virtualization** is required, **IOMMU** for isolated physical memory access, **nested virtualization** is possible, **nested multiplexing** allows multiplexing inside a guest without involvement of the host, and **migration** is supported.

Let us first discuss virtualization of NICs without RDMA. In general, these NICs do not support direct data access or direct access to network resources by applications. The easiest to manage and most flexible approach is an emulated or paravirtualized NIC like virtio-net. It does allow multiplexing without any hardware virtualization support or IOMMU. Furthermore, it supports nested virtualization and nested multiplexing, however, as direct data access is not supported, i.e. data and control path are not separated like in RDMA, every operation and its data has to go through all layers. Migration of these NICs is supported and can even keep connections alive. In contrast the advantage of pass-through NICs is that they can deliver bare-metal performance inside a VM. However, they do not support multiplexing, i.e. can only be used by one virtual machine at once, and they require an IOMMU. Furthermore, nested virtualization is supported if the guest operating system supports a virtualized IOMMU (cf. the turtles project [9]). This allows data access on the NIC’s receive queues by the kernel without going through several hypervisor levels. However, pass-through device cannot be multiplexed. Migration is possible (cf. [20] and [11] and further discussion in Section 7.1). SR-IOV NICs have the same features as pass-through NICs expect that they can virtualize themselves in hardware, thus allowing multiplexing. Although they do support multiplexing and nested virtualization, they do not allow nested multiplexing. That is, single SR-IOV devices cannot be multiplexed, i.e. the number of devices that are used in the nested environment have to be allocated on the host with direct hardware access to the PCI device to create virtual functions. Although the turtles project allows direct access of physical memory through a paravirtualized IOMMU, interrupts have to go through all layers.
Pass-through and SR-IOV RNICs share similar features to their non-RDMA-capable counterparts except they allow direct data access and direct application access. Like NICs they require virtualized IOMMUs for nested virtualization. We included three paravirtualized RDMA solutions: vVerbs, VMware’s vRDMA and High-Performance VMM-bypass I/O in Virtual Machines (cf. Section 6). All solutions do not require an IOMMU or hardware virtualization support of the RNIC. Furthermore, migration is possible. The difference is that vVerbs and VMM-bypass both allow direct isolated access to RNICs from applications inside VMs, whereas VMware’s vRDMA does not allow direct access thus adding latency on data path operations (cf. Section 6). In theory, nested virtualization should be possible with all proposed solutions. However, for vRDMA adding additional layer adds latency to data path operations. In contrast, both vVerbs and VMM-bypass do not have this limitation.
### Network Virtualization Techniques

<table>
<thead>
<tr>
<th>Network Virtualization Technique</th>
<th>Direct data placement</th>
<th>Direct application access</th>
<th>Multiplexing</th>
<th>Hardware virtualization</th>
<th>IOMMU</th>
<th>Nested virtualization</th>
<th>Nested multiplexing</th>
<th>Migration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fully virtualized NIC</td>
<td>-</td>
<td>-</td>
<td>X</td>
<td>-</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Pass-through NIC</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>X</td>
<td>X</td>
<td>-</td>
<td>X</td>
</tr>
<tr>
<td>SR-IOV NIC</td>
<td>-</td>
<td>-</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>-</td>
<td>X</td>
</tr>
<tr>
<td>Pass-through RNIC</td>
<td>X</td>
<td>X</td>
<td>-</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>-</td>
<td>X</td>
</tr>
<tr>
<td>SR-IOV RNIC</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>-</td>
<td>X</td>
</tr>
<tr>
<td>vVerbs</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>VMware vRDMA</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>VMM bypassing</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Table 5.1: Comparison of different network virtualization techniques

1. Requires fully virtualized IOMMU inside guests
2. See Section 7.1
6 Related Work

6.1 High-performance VMM-bypass I/O in virtual machines[22]

High-performance VMM-bypass I/O in VMs uses paravirtualization to support RDMA inside VMs. Figure 6.1 is taken from their paper showing the prototype called Xen-IB based on the OpenIB Gen2 stack and Xen as hypervisor. Their proposed design is very similar to vVerbs in that includes support for direct hardware access to RNIC network resources to completely bypass the hypervisor. Also for initialization, resource management, and memory registration they use a paravirtual communication channel like we do. However, memory registration in Xen is fundamentally different than in kvm, because Xen itself is a hypervisor operating system which only deals with VM memory mappings and resource multiplexing like CPU scheduling. For that reason device drivers cannot run inside the hypervisor directly, but run inside a special VM, called domain 0, which can be given direct device access. Thus, the paravirtual backend driver also has to run inside domain 0 to communicate with the native device driver. Furthermore, this allows Xen to directly expose physical addresses to user domains, i.e. when registering memory for RDMA, pages can be pinned in the guest kernel and directly used by the backend driver without translation. However, validity checks have to be performed to ensure that a user domain does not access other domain’s memory. In vVerbs we need to translate the addresses (cf. Section 3). However, physical guest to host virtual memory translation adds very little complexity as it is only a offset calculation. The resource allocation, as described in their paper, allocates buffers inside the guest domain and registers their physical pages in domain 0. Again, there is no need for translation in this scenario, only validation checks have to be performed. What is not explained in detail is if the user-level driver, to access these resources, can be used as-is. Furthermore, in the paper they argue that checkpointing and migration is difficult because of the state that is stored on the RNIC and has to be eventually transferred. They do not propose a solution to this problem, but consider it ongoing work. Their performance numbers match our experiences gathered with vVerbs, i.e. achieving bare-metal performance when polling for completion events and taking a small performance penalty for the event mechanism.
6.2 vRDMA[2]

VMware proposed a paravirtualized RDMA in 2012, called ‘vRDMA’. They do support direct data access, however, in contrast to *High performance VMM-bypass I/O in virtual machines* (cf. above) and vVerbs, they do not support direct, isolated hardware access inside the VM (cf. design 2 in Section 3). They implemented their solution for the VMware ESXi hypervisor and the OFED RDMA stack. VMware’s paravirtualization is enabled by a framework called Virtual Machine Communication Interface (VMCI) which is used by vRDMA. Thus, they implemented a vRDMA VMCI Endpoint (backend driver) which forwards operations from the vRDMA Driver (frontend driver) in the guest to kernel verbs API (cf. Figure 6.2). Inside the guest they use their own kernel- and user-provider, which exposes a virtual RDMA device to the guest. As described above, the main difference to our approach is that, besides control operations, data operations are forwarded over the VMCI Endpoint, e.g. posting work requests and polling for completion. Although, they claim there prototype is running and feature complete, they do not show performance results until today. Nevertheless, they expect 5 usecs half-round-trip latencies. While, their design choice allows for easier migration and checkpointing support, we believe that it has performance limitations, originating from the fact that no direct access to the RNIC is given to applications inside VMs. For example, read/write latency for SoftiWARP shown in Section 5, confirm that even ringing the doorbell can take up to 5 usecs and this does not include the overhead of retrieving work completions.
Figure 6.2: VMware vRDMA
7 Future Work

7.1 Migration and Checkpointing

Checkpointing allows saving a VM’s execution state by saving the VM’s physical memory and its virtualized hardware state in a way that allows continuing execution at this save point. Migration extends checkpointing by allowing to transfer the state to another host environment and continuing execution there. Live migration even allows moving a VM on the fly, i.e. transferring its state while the VM is running.

Checkpointing and migration for RNICs is challenging because the network stack with all its state is offloaded in hardware. Therefore, to support checkpointing, we have to be able to extract the RNIC’s state, like network resources (CQs, QPs), memory registrations (MRs) and connection management in a way that allows restoring it later, possibly on other hardware. To our knowledge there is no RNIC that is virtualization-aware and supports this directly, i.e. state has to be captured by software. Thus, one solution is to log state creation and modification initiated by software and replay it when continuing execution. Among others, outstanding problems are: (1) direct resource access which enables a application to directly program the RNIC and is typically performed over a shared memory region, i.e. after recreation of the resource, this mapping has to be reassigned or recreated in such a way that the uprovider can continue accessing it. (2) resource identifiers can change on recreation; however, applications only have knowledge of the old identifiers which they use to program them. Moreover there are operations that require knowledge of the remote resource identifiers. (3) local and remote keys of memory buffers can change on recreation, but the applications use the original identifiers from the initial creation to perform any data operations. (4) connection management for live migration has to support handing over connections.

Recently there has been an effort by Oracle to support migration for SR-IOV Infiniband RNICs[25]. Their idea is to support migration/checkpointing by making the kprovider and uprovider virtualization-aware. That is, the uprovider uses a translation table to replace resource IDs and local/remote keys when performing direct RNIC access. However, this requires all hosts in the same subnet to be virtualization-aware, because whenever a remote host wants to perform a RDMA operation, it has to use a remote key to identify the memory buffer, which might have been changed in the meantime if the VM was migrated. So whenever this key changes, the state has to be distributed to all hosts with an open connection to this VM. Additionally, for unreliable connections, QP ids are used to identify remote end points which also requires the distribution of ids. For connection management,
they implemented a reconnection mechanism in the RDMA connection library. However, for this they require the connection manager to be tolerant of device removal, i.e. that a QP is not destroyed after the device is detached. Another problem they are facing is that SR-IOV migration is quite complex, i.e. they have to detach a virtual function while saving the state and use PCI hot-plug support to attach a new virtual function on continued execution.

Paravirtualized and emulated solutions are typically a good fit for migration and checkpointing because their state is held in software and can be stored or transferred. However, in vVerbs this is only partially true, because we inherently rely on the underlying hardware to support bare-metal performance inside VMs. In the following, we propose a possible design extension to our vVerbs prototype to support checkpointing and migration. State capturing and recreation can be performed in the paravirtual frontend driver, i.e. state is fully captured by saving/transferring the VM’s physical memory. Because we allow direct application access to real hardware, the uprovider has to be made virtualization-aware. However, we do not want to modify the actual uprovider, but instead propose a redirection layer, guvirtverbs, to forward calls between uverbs and the uprovider (cf. Figure 7.1). Additionally, guvirtverbs allows us to support migration/checkpointing to different underlying hardware by changing the uprovider on the fly. Nevertheless changing resource identifiers and memory keys remain an unsolved issue. We believe that distributing the keys/ids as described above is not a feasible solution. Instead, we propose to either keep the translation tables inside the switch environment of the network or to ultimately make the RNIC virtualization-aware. Note that vRDMA (cf. Section 6) faces similar problems when it comes to local/remote keys of memory regions and resource ids. Although they can change keys while going through the paravirtualized control path (which we can also do through the redirection layer), they cannot control distributed remote keys and ids like all other solutions discussed. Eventually they rely on the RNICs direct data access capabilities, thus there is no way around using the original remote key the underlying hardware created on memory registration. Nevertheless, they claim to support migration with their design, but it remains to be open how they will resolve these issues.

7.2 Connection Management

In the following we propose an overhauled design of the connection management used in vVerbs introduced in Section 3. The problem with the existing design is that cm is performed completely on the host and thus the VM has no transparent view of its connections and their related properties like IP addresses. To solve this issue, we would like to perform essential connection management functions of kcm in the VM but forward device specific calls to the kprovider on the host. Furthermore, to multiplex between VMs, the RNIC has to support multiple addresses, e.g. IP aliases for IP-based connections. The goal is to make this transparent for the VM, i.e. whenever the VM changes its address it has to eventually be reflected onto the RNIC. We are aware that these changes might require vendor-specific driver extensions on the host; however, we believe that like the gkprovider, we can provide an API which requires minimal effort from vendors.
7. Future Work

In this Section we propose possible extensions to the OFED stack to make it virtualization-aware. One issue we need to work around is the user space memory registration inside the VM, which eventually requires a contiguous mapping in host virtual memory (cf. Section 3). This comes from the fact that we use the unmodified kverbs underneath. To be able to do memory registration without remapping, we propose a new verbs function which allows registering memory by providing a list of physical pages and an virtual address. Although optional, a similar function is already available in the kverbs API, although typically, kverbs and uverbs API are not meant to be used in conjunction. Thus, there should be either a separate uverbs kernel injection API, or kverbs-specific functions should be made uverbs-aware (from our experience most of the functions already are, although this depends on the vendor).

Figure 7.1: vVerbs design extended with migration/checkpointing support

7.3 OFED extension
8 Conclusion

In this thesis we showed that we can deliver close to bare-metal performance for RDMA-capable network interfaces by using a paravirtual subsystem instead of I/O hardware virtualization. We reasoned that exiting solutions, like SR-IOV, do not provide flexibility and add considerable complexity to the hardware and the virtualization environment. Moreover, we showed that by allowing direct, isolated application access to the RNIC’s network resources, our prototype implementation vVerbs can deliver performance equal to existing hardware virtualization solutions. This direct resource access allows us to achieve bare-metal performance when polling for work completions. For the event mechanism of work completions, we showed that the main source of overhead is introduced by interrupt injection into the VM (both for vVerbs and pass-through devices) because of the emulated interrupt controller (APIC). Thus, once this overhead is eliminated, e.g. by hardware virtualization of the APIC, we should be able to deliver bare-metal performance even without polling. Nevertheless, compared to recent work by others, we believe that we can deliver lower latencies and high throughput with less CPU load. Furthermore, compared to traditional paravirtual network stacks without RDMA, we showed that we can deliver up to 6x lower latencies while using the existing socket interface.

Another advantage of our design is that vVerbs uses state-of-the-art OFED RDMA programming infrastructure without modifications, which is achieved by providing a virtualization-aware plug-in component (gkprovider) in the guest environment which forwards control path operations over a paravirtualized driver. However, because of the design decision to allow direct application access to network resources, the gkprovider has to be vendor specific. Therefore, we developed an API for our paravirtual frontend driver ‘gkvirtverbs’, which allows forwarding control operations. Thus, the gkprovider is only a shim layer which redirects Verbs calls and prepares resource mappings. For example, it took us only 3 days to develop the gkprovider for the Chelsio T4 RNIC, where each Verbs call is typically implemented in less than 60 lines of code and without inherent understanding of the original kprovider.
To save power, AMD and Intel processors support several power saving mechanisms [17]. In order to save power when idle the C-state mechanism can be used. Newer processors support C-state levels up to 7, where C7 saves the most power and C0 is active state. Linux uses the cpuidle driver to determine when to switch into deeper power saving modes [27]. It allows switching between profiles depending on the requirements of the system. From this driver we extracted the exit latency numbers of the Sandy Bridge Microarchitecture in table A.1.

### Table A.1: Intel Sandy Bridge Microarchitecture

<table>
<thead>
<tr>
<th>C-state</th>
<th>Exit latency in usecs</th>
</tr>
</thead>
<tbody>
<tr>
<td>C0</td>
<td>0</td>
</tr>
<tr>
<td>C1</td>
<td>2</td>
</tr>
<tr>
<td>C1E</td>
<td>10</td>
</tr>
<tr>
<td>C3</td>
<td>80</td>
</tr>
<tr>
<td>C6</td>
<td>104</td>
</tr>
<tr>
<td>C7</td>
<td>109</td>
</tr>
</tbody>
</table>

To save power, AMD and Intel processors support several power saving mechanisms [17]. In order to save power when idle the C-state mechanism can be used. Newer processors support C-state levels up to 7, where C7 saves the most power and C0 is active state. Linux uses the cpuidle driver to determine when to switch into deeper power saving modes [27]. It allows switching between profiles depending on the requirements of the system. From this driver we extracted the exit latency numbers of the Sandy Bridge Microarchitecture in table A.1.

---

Bibliography


[12] Ethernet Working Group. IEEE 802.3. Institute of Electrical and Electronics Engineers.


[27] V. Pallipadi. cpuidle - Do nothing, efficiently... 2007.


