InfiniBand RDMA over PCI Express Networks

Alve Vreim Elde

Thesis submitted for the degree of
Master in Programming and System Architecture
60 credits

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2020

InfiniBand RDMA over PCI Express Networks

Alve Vreim Elde

InfiniBand RDMA over PCI Express Networks

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Abstract

Remote Direct Memory Access (RDMA) is a data transmission technology featuring high bandwidth, low latency, and low computational overhead. RDMA can be used to transmit data between applications running on interconnected computers while bypassing the kernel network stack. The efficiency of bypassing the kernel makes RDMA ideal for High-Performance Computing (HPC) and cloud computing. RDMA works by offloading network transfers in the kernel to network adapters with RDMA capabilities. These network adapters can read and write virtual memory of an application with Direct Memory Access (DMA). DMA operations are executed safely with a virtual-to-physical memory map created through RDMA semantics. RDMA capable network adapters exist for InfiniBand and Ethernet network fabrics, and transport agnostic RDMA APIs enable applications to run on either fabric.

In this thesis, we have developed an RDMA transport named "RDMA over PCIe (RoPCIe)," intended to be used in computer clusters interconnected with PCIe Non-Transparent Bridges (NTB). Inspired by RDMA over Converged Ethernet (RoCE), this transport takes advantage of the established RDMA APIs to provide a new mode of transport for existing RDMA applications. Bringing a technology developed primarily for InfiniBand to PCIe interconnect, an esoteric transmission technology compared to InfiniBand, is one of the primary motivations for RoPCIe.

We have implemented the RoPCIe transport for Linux and made it available to applications through RDMA APIs in both kernel-space and user-space. The primary component of the RoPCIe implementation is a virtual RDMA device, registered with the kernel RDMA subsystem by a virtual device driver. This virtual device acts as an intermediary between the RDMA API and the NTB device driver, translating RDMA operations into NTB operations. To add support for RoPCIe transport in user-space, we have implemented a dynamically linked plugin for a library called "Libibverbs".

The result of our RoPCIe implementation is a working proof-of-concept RDMA transport. RoPCIe can run unmodified RDMA applications over PCIe interconnect and achieves performance comparable to InfiniBand transport.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Background

The emergence of technologies like machine learning and cloud computing has increased the need for high-performance computer intercommunication. Pooling the resources of multiple computers in a data-center is the basis of a large number of technologies we rely on today. However, slow intercommunication can create bottlenecks and overheads that limit overall system performance. Low bandwidth intercommunication can lead to difficulties with scaling interconnected systems, and high latency can result in ineffective systems that are more expensive and wasteful to operate. RDMA is an intercommunication technology that can enhance communication and data transfer between computers, and significantly improve the efficiency and performance of data-centers.

RDMA enables memory-to-memory data transfer between applications running on different computers. When RDMA-capable network adapter cards interconnect two or more computers, applications running on either computer can request RDMA operations through an RDMA interface. RDMA operations are then asynchronously performed by processors on the RDMA network adapter. Data transfer requests bypass the kernel entirely, eliminating the CPU overhead a data transfer would otherwise cause. Paired with a high-performance network fabric, RDMA can move data between nodes with low latency and high throughput. For suitable tasks, RDMA can bring significant improvements over standard networking technologies [1].

RDMA is closely associated with a technology called InfiniBand. In the context of this thesis, RDMA refers to a standard developed by the InfiniBand Trade Association (IBTA), which is rooted in a previous standard called Virtual Interface Architecture (VIA). InfiniBand has historically been the driving force behind the development of RDMA because nearly all nodes in an InfiniBand network communicate with RDMA semantics. Today, InfiniBand is a name used to describe the entirety of a high-performance networking ecosystem, with specialized hardware and software. Despite the two technologies having been developed in conjunction with each other, and often by the same developers, RDMA has not

become locked to InfiniBand. The transport independence of RDMA has enabled a more recent effort to bring RDMA semantics to other network fabrics, specifically Ethernet.

Ethernet is a universal standard, with the majority of LANs and Wide Area Networks (WAN) relying upon it. Several factors have sparked interest in bringing RDMA to Ethernet, one of which is an increase in the performance of Ethernet hardware [2]. The current result of these efforts is three distinct implementations of RDMA over Ethernet; RoCEv1, RoCEv2, and iWARP. The main difference between them is the layer of the OSI networking layers in which they operate. RoCEv1 operates at the Data Link Layer, switching data frames over an Ethernet switch fabric. This RDMA implementation is limited to a LAN topology, and while this can be efficient, it can also be too limiting for some applications. RoCEv2 operates at the Transport Layer, routing packets with UDP/IPv4 or UDP/IPv6. It expands the range of transport to a WAN while keeping communication overhead relatively low, due to UDP being unreliable. Similarly to RoCEv2, iWarp routes packets at the Transport Layer, opting to use TCP/IPv4 or TCP/IPv6 instead [3]. TCP adds inherent reliability to the transport at the cost of not-insignificant communication overhead.

There are many applications for RDMA over Ethernet, and one of them is cloud computing. A cloud is created by interconnecting large numbers of computers to provide a dynamic and elastic computing resource. Ethernet is the most commonly used technology for interconnecting the cloud [4]. One reason for choosing Ethernet over InfiniBand is hardware cost; Ethernet hardware is usually cheaper than InfiniBand hardware [5]. Another reason for choosing Ethernet for the cloud is that the cloud is inherently outward-facing. The primary function of a cloud is to elastically and transparently lend computing power to outside actors. Not only does the cloud need high-performance internal communication, but it also needs high-performance external communication. The extroverted nature of the cloud is a significant driving factor behind the development of RDMA over Ethernet. RDMA has the possibility of increasing the internal efficiency of the cloud while providing a faster on-demand service to the user [6].

High-Performance Computing (HPC) is where RDMA has historically seen the most use with InfiniBand. In the bleeding-edge environment of HPC, there has always been a demand for high-performance Local Area Networks (LAN), which is the core offering of InfiniBand. Today InfiniBand and other RDMA transports provide the interconnection backbone for some of the most powerful supercomputers in the world [7]. Low latency is especially important here, as each computer can do a large number of instructions in the time it takes an electrical signal to travel through a copper wire, or even light sent through an optical fiber, from one compute node to another. A low latency interconnection can reduce the amount of time a compute node spends in an idle state, thus reducing the time it takes for the system to do complex calculations.

Recent research suggests that RDMA can be used to accelerate deep learning. Training a Deep Neural Network (DNN) can be very computationally intensive, where one machine may not be able to train the network

in a reasonable time frame. The solution is often to throw more machines at the problem by distributing the computation over many computers. However, the computation is not entirely separable into chunks, and frequently requires a data exchange between the machines before the computation can continue. Most current distributed deep learning frameworks use Remote Procedure Calls (RPC) to perform this data exchange, but [8] suggests that an implementation optimized for RDMA can result in "up to 169% improvement against an RPC implementation optimized for RDMA, leading to faster convergence in the training process".

1.2 Problem definition

The goal of RDMA is to facilitate data transfers with high bandwidth, low latency, and low computational overhead. RDMA over Converged Ethernet (RoCE) has demonstrated the possibility of implementing RDMA for other transmission technologies than InfiniBand. This possibility leads us to investigate PCIe NTBs from Dolphin ICS as a candidate for a new RDMA transport. The performance characteristics of PCIe NTBs align well with the RDMA performance goals, and the transmission semantics are similar. Therefore we want to study the viability of RDMA over a PCIe NTBs by implementing a new RDMA transport named RDMA over PCIe (RoPCIe). The four main goals of this thesis are as follows:

- Research how exiting RDMA transports integrate with RDMA infrastructure in Linux. Create a design for a RoPCIe transport in Linux.
- Implement a proof of concept RoPCIe transport according to design. RoPCIe should support the minimum viable set of features and be available to applications in user-space.
- Measure the performance of the RoPCIe transport and evaluate it as an alternative to other RDMA transports. Compare functional and non-functional characteristics.
- Verify that existing software can use the RoPCIe transport without modifications to software or transport.

1.3 Limitations

The focus of this thesis has been to create a functional RDMA transport over PCIe interconnect, with the ability to send and receive data from user-space memory between two machines. We have not pursued a comprehensive implementation of the entire RDMA specification, nor have we targeted any other platforms than Linux.

The entirety of the RoPCIe transport exists on top of an API to a PCIe NTB. Integrating the RoPCIe transport with the PCIe NTB device driver is likely to improve RDMA performance due to API limitations. However, we consider modifications to the PCIe NTB device driver code out of scope for this thesis.

1.4 Main contributions

In this thesis, we show that proprietary data transmission technology without support for InfiniBand RDMA can be integrated with existing RDMA libraries, and used by existing RDMA software.

First, we design and implement a device driver capable of presenting a virtual RDMA device to the Linux kernel as a physical RDMA endpoint.

Next, we implement the RDMA operations needed to operate an RDMA transport between the kernel of two machines. The virtual RDMA device translates these RDMA operations into NTB operations, and asynchronous kernel workers initiate memory-to-memory data transfers through the NTB.

Finally, we make the RDMA transport available to applications in user-space by implementing a dynamically linked plugin for a user-space library called "Libibverbs". The plugin defers all RDMA operation requests from user-space to the virtual RDMA device in the kernel.

The result of this implementation is a new RDMA transport available to existing RDMA applications. We measure the end to end latency of the transport to be about 3.5 μs, and throughput to be about 82 Gb/sec.

1.5 Research method

While working on this thesis, we have designed and implemented an RDMA transport named RoPCIe. The RoPCIe transport works and can be used without modifying existing RDMA software or patching the underlying kernel. We have evaluated and tested our implementation. This corresponds to the *design paradigm* of the ACM classification [9].

Evaluations have been performed on hardware provided by Dolphin Interconnect Solutions.

1.6 Outline

In chapter 2, we introduce the mechanics, specification, and implementation of RDMA. We mainly focus on the concepts used in the implementation of RoPCIe transport, but we also cover some additional concepts that can be used to develop RoPCIe further.

In chapter 3 we discuss the design process of the RoPCIe transport. We define the requirements and scope of the transport and detail the different transport architectures that we considered during the design process.

In chapter 4 we describe the implementation of the RoPCIe transport. We detail the functionality of the virtual device driver, and how it integrates a virtual RDMA adapter interface with the kernel. We also detail the Libibverbs library plugin, and how it cooperates with the virtual RDMA adapter interface in the kernel.

In chapter 5 we measure the performance of RoPCIe transport and compare it to InfiniBand transport. We measure the transport latency and throughput for a variety of message sizes and examine the messages for

data integrity. We also do an end to end analysis of the RoPCIe transport, identifying and measuring sources of overhead.

In chapter 6, we provide a summary of the thesis and a conclusion of the result.

Chapter 2

RDMA

2.1 Mechanics

RDMA is a complex data transmission technology comprised of many interconnected systems. One can set up and use RDMA in several ways that result in distinctly different data transfer operations, and with different performance characteristics. Before we discuss the working internals of RDMA, an overview of the primary working mechanics of RDMA will be presented.

2.1.1 Kernel bypass

One of the main features of RDMA is kernel bypass on data transfer. An application operating in either kernel-space or user-space can request an RDMA operation through an RDMA interface. This interface requests an RDMA service in the kernel to set up and manage a connection to a remote computer. The remote computer is similarly configured to accept RDMA, and the two computers are linked with Host Channel Adapters (HCA). These HCAs are RDMA-enabled Network Interface Cards (NIC) mounted in PCI express (PCIe) slots on each machine and can use Direct Memory Access (DMA) to directly read from and write to each machines Random Access Memory (RAM). When two machines are linked and have established a connection, applications on either machine can request an RDMA operation through the RDMA interface. This request is passed from the RDMA service directly to the HCA. The HCA will then carry out the operation, directly accessing the RAM from the PCIe bus, bypassing the kernel entirely when sending and receiving data, as shown in section 2.1.

Fast path operation

User-space applications can bypass the kernel when requesting some RDMA operations by using a fast path to the HCA. Typically, this is only done for latency-sensitive operations, like initiating an RDMA transfer. Before the fast path can be used for such operations, an application needs to request the RDMA service in kernel-space to set up communication

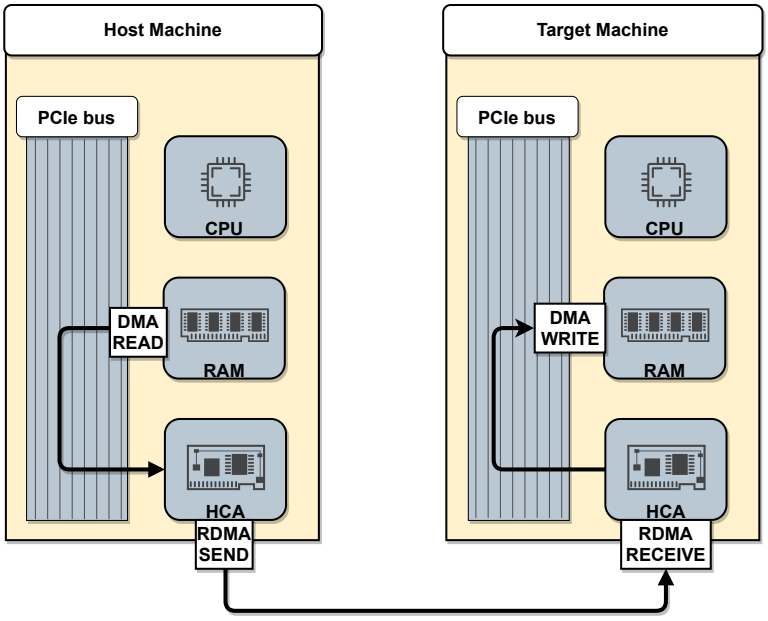

Figure 2.1: The path of a data transfer operation between a host machine and a remote machine.

with another machine. These requests must take the slow path from user-space to kernel-space through special files in the file system. After these requests have been fulfilled, some of the hardware registers of an HCA can be mapped into memory. Mapping the HCA into memory creates a fast path between the application in user-space and the HCA. With a fast path set up, the application can send RDMA transfer operation requests directly to the HCA, bypassing the kernel and reducing latency and computational overhead.

Benefits of bypassing the kernel

There are several benefits gained from bypassing the kernel; lower latency, lower power usage, and lower computational load. These three attributes are highly attractive in resource-focused environments like data-centers and high-performance computing environments. The following explains how RDMA can improve each.

Latency: The number of CPUs in a computer system is generally smaller than the number of active tasks managed by the computer Operating System (OS). Linux is a multitasking OS, meaning it will preempt and switch tasks regularly according to a set of policies. Each task switch will increase the latency by the time it takes to perform, and can indirectly cause significant overhead [10]. Kernel bypass mitigates task-switching overhead by minimizing the number of task switches required. An OS managed task only needs to write a short RDMA operation request to mapped memory, and then the task can be freely switched out by the OS. The HCA will read

the request from mapped memory, and perform it with an uninterrupted processor. Task switching overhead is avoided on the remote machine, as the remote HCA also can read and write directly to pinned memory to complete the data transfer.

Power consumption: Every CPU has some electrical resistance that converts power into waste heat. The CPU generates some heat when idle, and heat generation increases with higher CPU utilization. Heat must be removed from the CPU by cooling equipment to avoid overheating it and damaging the internal circuits. Cooling equipment takes space, maintenance, and additional power to operate. CPU utilization over time directly affects the power consumption of a computer system, and indirectly its monetary cost of operation. Electricity is a substantial cost factor for all data-centres [11].

Kernel bypass can reduce total power consumption in a system by offloading work to lower-power hardware. A data transfer is generally not very compute-intensive, so a modern server CPU is overkill for the task. Smaller passively cooled processors, like the one found on an InfiniBand HCA, can reduce total power consumption for some system configurations and workloads. High processor utilization on the HCA over time indicates a high degree of CPU offload, where the CPU is free to do other more useful work. Therefore, HCAs can reduce both the power usage and the number of high power CPUs required in a data-center.

Computational load: As touched upon in the previous paragraph, data transfer is typically not compute-intensive for a modern CPU, but it takes CPU time. For some use-cases, like HPC, it is crucial to utilize every processor as much as possible. Offloading tasks with high CPU time, but low CPU utilization becomes very important to be able to exploit the computational power of a system.

2.1.2 Memory pinning

All processes in user-space operate in their own virtual address space, managed by the underlying operating system kernel. The virtual address space is segmented into pages of a fixed size, but this segmentation is transparent to the user. The kernel may swap any page out for another page for several reasons, by writing the old page to disk and reading a new page from disk to replace it. Page swapping makes the kernel bypass behavior of an HCA risky, as the kernel can swap out a page at any time from the perspective of the HCA. The solution is to "pin" the page and prevent the kernel from swapping it out.

There are multiple ways to pin memory, some of which do not guarantee the page stay at the same physical position in memory. The kernel may still move the page in memory as it sees fit, causing a soft page fault the next time the page is accessed. The page fault handler in the kernel quickly resolves a soft page fault without retrieving anything from disk.

This method of pinning is good enough for most applications, but not for RDMA.

We need a stricter form of memory pinning when doing RDMA, as the page needs to stay at the same physical position in memory. In Linux, this can be accomplished by a call **get_user_pages()**, or a variant of this function. Using this family of functions enables us to lock down pages in memory for an arbitrary amount of time, and a DMA to this memory will always find the same data. However, there are some implications of using this feature, with recent research showing that it can be problematic [12]. The issues with **get_user_pages()** are out of scope for this thesis.

Pinning memory is necessary for RDMA and has the benefit of somewhat reducing memory access latency, but there are also some drawbacks. As stated above, the CPU is free to do other tasks during RDMA transfers. However, the usefulness of doing so may be limited by the amount of system memory available after pinning. The available system memory may also be fragmented by pinned memory regions, limiting the ability to allocate large blocks of physically contiguous memory. Fragmentation is usually less of a problem, as most tasks only require a virtually contiguous memory.

2.1.3 Zero-copy data transfer

The concept of zero-copy data transfer is not exclusive to RDMA, but RDMA may be the technology that benefits the most from it. A standard network connection between two applications will copy the data down through the network stack on the first machine, and up through the network stack on the second. Copying the data consumes both CPU and memory resources, and increases latency. Many operating systems have ways to eliminate the sender-side copy by transmitting data directly from user-space buffers, but eliminating the receiver-side copy is non-trivial.

RDMA eliminates the receiver-side copy by pinning memory on the receiver machine, and telling the sender-side where it can write the data. DMA from the HCA enables zero-copy transfer form user space to user space on two machines, without the involvement of either kernel. While this can achieve lower latency and increased throughput, it gives rise to concerns about memory protection. As there is no government from either kernel, a malicious or byzantine sender could place data outside the buffer allocated by the receiver without the proper security measures. Memory protection is a fundamental security concern in RDMA systems, but it is not an unsolvable problem. InfiniBand, for example, addresses this with a key validation system built into silicone and driver code of each HCA.

2.2 Verbs

2.2.1 InfiniBand Architecture Specification

The InfiniBand Architecture Specification Volume 1 [13] published by InfiniBand Trade Association (IBTA), describes the RDMA technology in

detail. The conceptual overview of this document is stated as follows:

> The InfiniBand Architecture Specification describes a first order interconnect technology for interconnecting processor nodes and I/O nodes to form a system area network. The architecture is independent of the host operating system (OS) and processor platform.

The InfiniBand Architecture Specification often referred to as the IBTA specification, is extensive and detailed. Although it covers some topics that are currently exclusive to InfiniBand, the majority of its chapters apply to all RDMA transports, including the RoPCIe transport. The implementation of RDMA transports do sometimes deviate from the IBTA specification, but the content of the IBTA specification is typically considered the RDMA transport standard.

Chapter 11 of the IBTA specification describes several semantic behaviors called verbs. These verbs collectively define the behavior of an RDMA transport endpoint, which is described in the specification as a Host Channel Adapter (HCA). The IBTA specification does **not** define any concrete Application Programming Interface (API) or specific software constructs. It only describes behaviors that an HCA must exhibit in order to be compliant with the specification. The term HCA is sometimes used to describe the role of an InfiniBand adapter specifically, but it is also used as a general term to describe any adapter operating an RDMA transport. Going forward, we use the latter definition unless otherwise stated.

In total, the IBTA specification defines 59 verbs. Of these verbs, 36 are classified as mandatory, meaning all HCAs should support them in order to be compliant with the IBTA specification. The set of mandatory verbs is larger than the minimal set of verbs required for the successful operation of RDMA between nodes, so an RDMA transport under development can opt to target a smaller number of verbs for proof of concept. The remaining verbs are grouped into various features, where an HCA must support all verbs of a feature in order to claim support for that feature. Most of the verbs can be mapped one-to-one with functions in Verbs APIs, but not all.

The following describes how verbs can be used to create, manage, and destroy verbs resources. We do not cover the functionality of all the verbs, only the core verbs involved in typical usage patterns. Figure 2.2 shows the relationship between resources created through verbs. The figure includes the primary verbs resources: HCA contexts, protection domains, queue pairs, completion queues, and memory regions.

2.2.2 Host channel adapter

Discover HCA

The first step in the process of setting up an RDMA transport is discovering an HCA and obtaining an identifier from it. This identifier is unique to the HCA and will later be used to open it. The IBTA specification does not specify the discovery of an HCA as a verb. Consequently, the completion

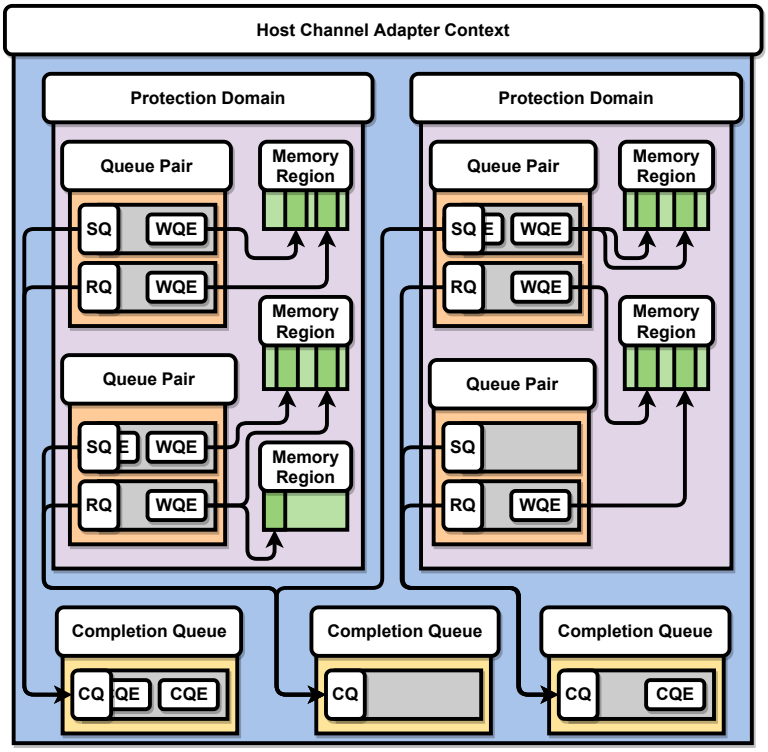

Figure 2.2: Example of relationships between verbs resources after initialization.

of this step is entirely dictated by the user. As long as the identifier is valid, it can be obtained in any appropriate manner, and the identifier can even be hard-coded in a program or entered manually by a user. However, users will typically opt to take advantage of device management facilities in a verbs API. A user can request a list of HCAs from the verbs API, and the verbs API will probe each HCA registered with the system, and return a list of all active HCAs. The user is then able to choose which HCA to open from the returned list.

Open HCA

When a user has chosen an HCA identifier, the HCA can be opened with a verb named "Open HCA." This verb is the first verb every RDMA user is required to call, and a handle to the HCA will be returned to the user upon successful completion. Opening an HCA will also associate it with the user, and create a line of communication between the two. The HCA handle acts as a device context for following verbs operations, where any new resources created by the user through verbs operations are associated with the HCA context. Here we can observe the emergence of a context-based design pattern in the IBTA specification, and by extension, the verbs API. The device context acts as the top-level context, and scoped parts are passed as input to the other verbs.

Query and modify HCA

After opening an HCA, the handle can be used to retrieve and modify specific attributes of the device. A verb named "Query HCA" can be used to obtain a list of all attributes relevant to the HCA. The attribute list includes attributes such as; the number of physical ports on the HCA, the Globally Unique IDentifier (GUID) of the HCA, and the number of protection domains, memory regions, and queue pairs the HCA supports. Some of the attributes in this list can be changed by the user with a verb named "Modify HCA." If the user calls this verb to request that one or more attribute be changed, and the changes requested are to modifiable attributes, and within modifier limits, the attributes will be changed for one or more ports on that HCA. This verb can be used to influence the operating parameters of RDMA transport through individual ports on an HCA.

2.2.3 Verbs resources

Protection domains

A Protection Domain (PD) is a sub-context of the top-level HCA device context. It can be allocated with a verb named "Allocate Protection Domain," which takes an HCA handle as input and returns an allocated PD as output. The HCA dictates the number of PDs that can be allocated at any time, and the limit will be checked in the HCA attributes upon allocation. Once allocated, a PD will be linked to the HCA handle, and can not be moved to another HCA handle, or unlinked from the HCA without deallocating it. When a PD is no longer in use, it should be deallocated with a verb named "Deallocate Protection Domain." Both PD verbs are mandatory in the IBTA specification and must be supported by every HCA.

The primary purpose of the PD is to provide a separation of resources associated with an HCA. A PD can keep track of resources like queue pairs, memory regions, and shared receive queues and impose limits on the allocation of these resources. The resources linked to a PD cannot be created without a PD or moved from the PD once created. The PD creates a guarantee of resource ownership, which is vital for memory protection and execution safety. A user cannot use RDMA to transmit data without first creating at least one queue pair, so it follows that at least one PD is required for RDMA operations.

Memory regions

A Memory Region (MR) is the resource used to pin a range of memory for RDMA transfers. The most common way for users to register a new MR is to use a verb named "Register Memory Region." This verb takes an HCA handle, a PD handle, a virtual address and length, and some access control modifiers. It returns an MR with a map of pinned memory, starting at the specified virtual address start and ending at an address given by the sum of the start address and length.

The virtual address map may be used for Direct Memory Access (DMA) from the HCA, which bypasses the kernel. If an MR was not created, and the kernel was allowed to swap out any memory pages that intersect with some of the memory range, DMA from the HCA could result in reading or writing data from another virtual address space. DMA without a virtual address map would be unsafe from an information security standpoint, and would likely result in the application reading garbage data. The system responsible for memory pinning is the IB Uverbs module discussed in section 2.3.5.

Another way to register a Memory Region is to use a verb named "Register Physical Memory Region." The physically addressed MR handle returned from this verb is interchangeable with a virtually addressed MR handle to verbs that take an MR as input. However, there are some differences in how they are created. This verb takes a list of physical buffers as input in addition to the attributes from the previous verb. The virtual address of the MR will be associated with the start of the first physical buffer in the buffer list. All the physical buffers are assumed to be in reserved physical memory, so IB Uverbs is not needed to pin memory for these buffers.

Local and remote access keys

Access to an MR is regulated through the use of a local access key (l_key) and a remote access key (r_key). The two keys act as identifiers for an MR and are produced as output from the MR registration procedure. An l_key is always produced and regulates read and write access from local users. An r_key must be requested explicitly at MR registration, and regulates read and write access from remote users. The r_key plays an essential role in RDMA communication, and without it, remote users would be unable to perform RDMA.

Queue pairs

A Queue Pair (QP) is a pair of Work Queues (WQ) that handle Work Queue Elements (WQE). A WQE represents a work request and describes a unit of work to be done by the HCA. Every QP is created as a pair of two separate WQs, a Send Queue (SQ) and a Receive Queue (RQ). A user can "post" a WQE to the end of one of these WQs, and the HCA will process the WQEs added to the WQ in a First In First Out (FIFO) order. No individual handles for the WQs are exposed to the user, so operations involving either WQ must be carried out through the QP handle. If a user wants to post a WQE to the SQ, the QP handle is used as input to a verb named "Post Send Request," along with the WQE. For posting a WQE to the RQ, a verb named "Post Receive Request" would be used.

A QP can be created with a verb named "Create Queue Pair," which requires an HCA handle, a PD handle, as well as a set of attributes as input. The attributes of the QP can be retrieved at a later time with a verb named "Query Queue Pair," and a subset of these attributes can be modified with a

verb named "Modify Queue Pair." Some attributes can not be modified at a later time, for example, Completion Queue handles for the SQ and the RQ, which must be set and valid at QP creation time. It is normal for a user to create many QPs on one HCA, and the QP verbs mentioned are mandatory for every HCA to support.

A QP has a state that determines what action can be performed with it. When a QP is first created, its state is set to RESET. In this state, a QP cannot be connected to other QPs, and no work requests can be consumed from either WQ. A QP must be transitioned through a sequence of states in order to make it fully operational. To transition a QP to the next state, the user must call a verb named "Modify Queue Pair," and modify a specific set of attributes. The IBTA specification defines the sets of attributes that are required, as well as some optional attributes. If the requirements are met, the QP will transition from one state to the next. A QP can also be returned to the RESET from any other state.

If a user modifies a QP in the RESET state with the correct attributes, the QP will transition to the INIT state. A QP in the INIT state can still not do any useful work, so another transition to the Ready To Receive (RTR) state is required. Upon successful transition to the RTR state, a QP can process work requests in the RQ. If a QP is only intended to process receive requests, a user may choose to stop the state transitioning at RTR. But if a QP is intended to process any send requests, the user must transition the QP to the Ready to Send (RTS) state. A QP in the RTS state can process work requests in both the SQ and the RQ, and from this state it can only be transitioned back to the RESET state.

Completion Queues

A Completion Queue (CQ) is a single queue that handles Completion Queue Elements (CQE). A CQE represents a work completion, and is posted to CQ by the HCA. When an HCA completes a work request from a WQ, it will post one CQE to the CQ associated with that WQ. The work request may or may not have been completed successfully, which is indicated by the CQE. A QPs initial attributes require a CQ handle for the SQ and the RQ, and these CQ handles do can resolve to different CQs, or the same CQ. A CQ can be associated with multiple WQs in multiple QPs across multiple PDs, usually only limited by the number of CQEs the CQ can hold. The CQ the central resource in RDMA for confirming the completion of asynchronous work.

A CQ can be created with a verb named "Create Completion Queue." This verb takes an HCA handle, minimum CQE capacity, and an optional completion event handler as input. As mentioned previously, the CQ does not require a PD handle, meaning it can gather work completions across multiple PDs. Because every QP requires at least one CQ, this verb must be called at least once before creating a QP and is mandatory for HCAs to support. Unlike the QP, the CQ is stateless and ready to use immediately after creation.

A work request usually results in a work completion, but not always.

In some cases, a WQE posted to a WQ will not eventually result in a CQE being posted to the CQ. This behavior can be observed when processing elements in SQs or RQs, but the two WQ types require a different set of circumstances in order to omit the work completion. An SQ can omit work completions if the associated QP is configured for unsignaled completions and the WQE being processed requests not to generate a completion. If such a WQE is completed successfully, no CQE will be generated in the CQ associated with the SQ. Unsignaled completions can not be configured for WQEs in an RQ, but some modes of RDMA operations behave similarly. If a QP is configured to process reliable datagrams or is associated with a Send Receive Queue (SRQ), processing and completing WQEs in the RQ will not generate CQEs in the associated CQ.

2.2.4 Communication

After the HCA has been opened, and verbs resources have been created and configured, communication can be established between QPs. Communication in this context implies a negotiation of data transfer path and parameters, but not the RDMA data transfer itself. The transport service type of each QP determines how communication between QPs is established, and what information is exchanged. The following describes the process of configuring a QP for a transport service, establishing communication with one or more remote QPs, and readying the QPs for RDMA operation.

Transport service types

A transport service type is a term used to classify the behavior of a transport service used by a QP. The IBTA specification defines six distinct transport service types, and four of them are covered here; Reliable Connection (RC), Unreliable Connection (UC), Reliable Datagram (RD), Unreliable Datagram (UD). The two remaining transport service types, XRC and Raw Datagram, are considered out of scope for this thesis.

The transport service type of a QP is set by the QP attributes. Each QP associated with an HCA can be configured differently, and target different remote QPs. A QP can only target a remote QPs with the same service type, which means that not all QPs can work together without reconfiguration. The service type of a QP must specified at creation time, but it can later be changed by modifying the attributes. Modifying the service type while there are still uncompleted WQEs in one or both WQs may cause some WQEs to be completed differently, or not completed at all. This should in general be avoided. A user that has taken initiative to request communication between QPs is called a **requester**, and a user that has responded to an incoming communication establishment request is called a **responder**.

Reliable connected transport service: The RC transport service type promises to establish a one-to-one connection between a requester and a responder. It also promises to transport messages between them reliably.

The reliability promise guarantees that messages will be retransmitted until delivered without missing or corrupted data, that all messages are delivered in the correct order, and that each message is delivered exactly once. If the link is broken, the transport will attempt to find a new path to the target and reconnect to resume message transmission. The one-to-one connection promise restricts the requester and responder to only communicating with each other for the duration of the connection. Neither party can establish new connections to other QPs or accept incoming connection requests. Any WQE posted to an SQ is guaranteed to result in a message sent to the responder. The reliability promise guarantees that the message will be delivered (if possible) and processed by the responder.

Unreliable connected transport service: The UC transport service type promises to establish a one-to-one connection between a requester and a responder. It also promises to transport messages between them unreliably. The unreliability promise does not guarantee that messages are delivered, and lost messages are not retransmitted. Incoming messages are scanned for corrupt data, and this transport service type will detect packet loss, packet duplication, and wrong packet order. The connection promise imposes the same restriction on the requester and responder as in the RC transport service type. Connections are exclusive and must be explicitly terminated. Any WQE posted to the SQ is guaranteed to result in a message sent to the responder. However, the UC transport service type does not guarantee that the message will be delivered to the responder.

Reliable datagram transport service: The RD transport service type promises to reliably deliver messages from a requester to any reachable and receptive responder. The reliability promise in the RD transport service type makes the same guarantees as the reliability promise in the RC transport service type does, ensuring all messages are delivered correctly and intact. The datagram promise allows communication between one requester and multiple responders, where no connection needs to be established before messages can be sent between them. A requester can process single work requests that target multiple responders.

Requesters and responders with the RD transport service type both require End-To-End Contexts (EEC) to fulfill the reliability promise. An EEC will act as a middle layer between the RD QP and the HCA and keeps track of all incoming and outgoing messages from the QP. Only one EEC is required per HCA because one EEC can keep track of multiple RD QPs across multiple PDs. To create an EEC, a verb named "Create EE Context" is used. This verb takes an HCA handle and an RD domain handle as input. An RD domain is a mechanism used to associate RD QPs with EE Contexts, and is allocated with a verb named "Allocate Reliable Datagram Domain." This verb returns an RD domain handle, which can be associated with one or more QPs by modifying an attribute in the QP attribute list. If a QP is correctly configured for RD transport, all WQEs posted to that QP will be handled by the HCA through the EEC, and the EEC will be responsible for

reliably performing the work described by each WQE.

Unreliable datagram transport service: The UD transport service type promises to unreliably deliver messages from a requester to any other receptive responder that can be reached. The unreliability promise does not guarantee that messages are delivered, and lost messages are not retransmitted. Incoming messages are scanned for corrupt data, but this transport service type will not detect packet loss, packet duplication, or wrong packet order. The datagram promise allows communication between one requester and multiple responders, where no connection needs to be established before messages can be sent between them, and a requester can process single work requests that target multiple responders.

A UD transport service does not have the same reliability requirements as an RD transport service and consequently does not need an EEC. Instead, UD transport is accomplished with the help of Address Handles (AH). An AH is used to define a local or a global address and can be referenced by a WQE. It is created with a verb named "Create Address Handle," and is used by the HCA to resolve a path to the recipient of a message. A message over UD transport will not be acknowledged by the recipient, which can make it a non-viable choice of transport for some systems. However, some benefits can also be gained by using unreliable transport. UD transport is usually highly scalable, has excellent performance, and the maximum message size of datagrams is the Maximum Transmission Unit (MTU) of the fabric. The low computational overhead makes UD a good option in high-performance computing environments [14].

Communication manager

The Communication Manager (CM) is an entity defined in chapter 12 of the IBTA specification. A CM can be used to assist with the establishment and management of communication between requesters and responders over RC, UC, and RD transport. UD transport is supported through a separate system in the CM implementation, called the Service ID Resolution (SIDR) protocol. The CM establishes communication by special Management Datagrams (MAD), sent to and from the General Services Interface (GSI) on each HCA. The GSI is a well-known QP that the HCA can use for an assortment for tasks. As this QP can become heavily loaded, the CM can also create new QPs to use for communication with another HCA CM after initial communication has been established.

The CM is implemented as a separate library complementing the verbs library. It has a separate API from the main verbs API and internally makes use of the verbs API to accomplish some of its tasks. Device-specific functionality can also be provided to the CM, with an interface similar to the verbs library. While the CM can be useful in managing larger RDMA systems, it is not mandatory to use. However, communication and connections can be established without the use of the CM with relative ease. Therefore, the CM is considered out of scope for this thesis.

The SIDR Protocol is a system defined as a part of the CM. It is created to assist with the establishment and management of communication between requesters and responders over a UD Transport Service. SIDR is a relatively small protocol consisting of only two MADs, a request and a reply sent over the GSI. The details of this protocol are also considered out of scope for this thesis.

External coordination

The CM can be a useful tool, but it is not required to set up communication between a requester QP and a responder QP. Any external utilities that can be used to send information between two machines can be used to coordinate the RDMA services. Network sockets are often used for this purpose.

2.2.5 Transfer operations

In a data transfer between two machines, the machine initiating the transfer is the requester, and the machine accepting the transfer is the responder. A transfer operation defines the combined behavior of a requester and a responder in a data transfer. Not all HCAs support all transfer operations because some transfer operations require the requester and responder to have a reliable transport service type. There are in total four transfer operations, but only one is mandatory for every HCA to support. The following details three of the transfer operations: Send/Receive, RDMA Read, and RDMA write. The fourth, Atomic Read/Write, is out of scope for this thesis.

Each transfer operation is a complicated procedure with many sub-procedures. Some sub-procedures are optional and are mandatory. For a detailed description of each transfer operation, we refer to the IBTA specification. The configuration of a QP combined with the configuration of WQEs posted to it can lead to many different behaviors within the umbrella of one transfer operations, and this thesis will only cover the most common configurations. The role of some components, especially the MR, has been generalized for ease of explanation. A more detailed view of each component is provided later in the thesis when discussing the details of our implementation of RoPCIe.

Send/Receive

The Send/Receive transfer operation must be supported over every transport service type an HCA supports. This transfer operation assumes communication has previously been established between the requester and the responder, regardless of the transport service type. Any transport service type may be used to Send and Receive, but both sides must agree on which to use. In this operation, both the requester and the responder are active, and work requests are consumed on both sides. Data can be gathered from multiple ranges of memory on the requester side, and

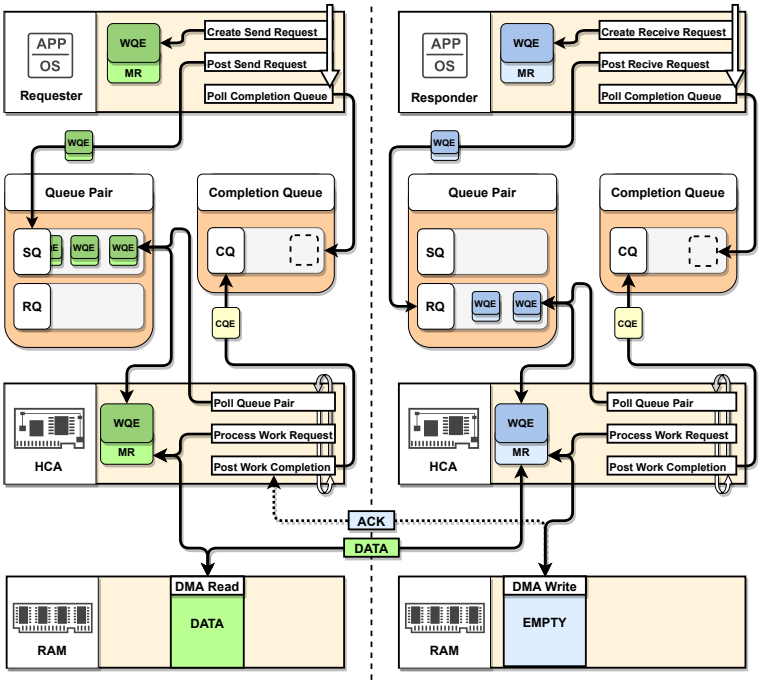

Figure 2.3: Send/Receive transfer operation.

scattered into multiple ranges of memory on the responder side. Figure 2.3 shows an overview of a Send/Receive transfer operation between a requester and a responder.

Requester: The requester in a Send/Receive transfer operation produces send requests in the form of WQEs posted to the SQ of a QP. The contents of a WQE are the following:

- **MR:** A handle to a local MR registered by the requester. This MR handle defines one or more ranges of memory to be read by the HCA and transmitted to the responder.

The requester HCA will process each WQE in the order they were posted. When a WQE is processed, the memory ranges defined by the WQE will be read by the HCA with DMA. The memory ranges are combined into one or more packets, depending on the total amount of data. The UC and RC transport service types can send 1000 MB of data per packet, while UD and RD can send the Maximum Transmission Unit (MTU) of the transport link. When the data is sent, a CQE can be posted to the CQ if the transport service is unreliable. If the transport service is reliable, the requester HCA will not post a CQE to the CQ before the responder sends an ACK message back. The ACK acknowledges reception of data, and that a receive request was posted in advance to receive the data.

Responder: The responder in a Send/Receive transfer operation produces receive requests in the form of WQEs posted to the RQ of a QP. The

contents of the WQE are the following:

- **MR:** A handle to a local MR registered by the responder. This MR handle defines one or more ranges of memory to be written to by the HCA when data is received from a requester.

The responder HCA will process each WQE in the order they were posted. When the HCA receives an incoming data transmission, it pulls the next receive request from the QP, and places the incoming data into one or more memory ranges defined by the MR in the WQE. If the RQ is empty, the incoming data packets will be rejected. A CQE will be posted to the CQ if the transfer is completed successfully. If the QP transport service is reliable, an ACK message will be returned to the requester.

RDMA Read

The RDMA Read transfer operation is mandatory to support for any HCA that supports the RC and RD transport service types. RDMA Read can only be done over reliable transport, which makes it unviable for some applications. Only the requester is active in this operation, meaning the kernel at the responder side is completely bypassed. However, some setup is still required of the responder. The responder must declare memory to be remotely accessible with an MR before a requester can read it. Data can be scattered to multiple ranges of memory on the requester side, but can only be read from one contiguous range of memory on the responder side. Figure 2.4 shows an overview of an RDMA Read transfer operation between a requester and a responder.

Requester: The requester in an RDMA Read transfer operation produces send requests in the form of WQEs posted to the SQ of a QP. The contents of the WQE are the following:

- **MR:** A handle to a local MR registered by the requester. This MR handle defines one or more ranges of memory to be written to by the HCA when data is received from the responder.
- **r_key:** A key identifying an MR registered by the responder. This key was sent from the responder to the requester at some point before the WQE was created.
- **Address:** A virtual address specifying the start of the memory buffer to be read from the responder. This must be a valid address inside of one of the memory ranges spanned by the MR. It can either be absolute or zero-based (if supported by the HCA).
- **Length:** An integer specifying the number of bytes from the address to be read by the responder and sent back to the requester.

The requester HCA will process each WQE in the order they were posted. When a WQE is processed, the r_key and address is assembled into a packet

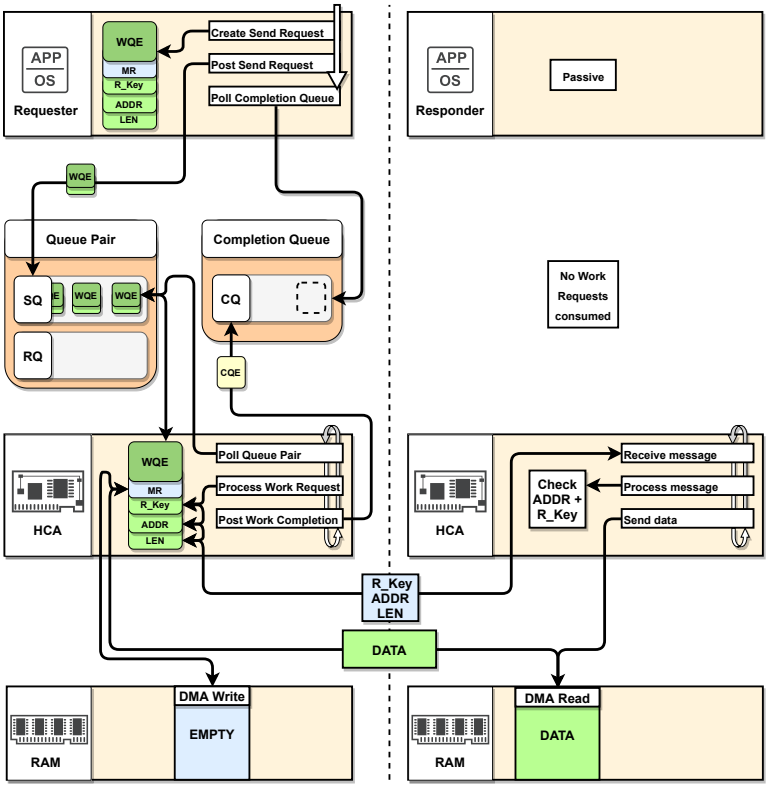

Figure 2.4: RDMA Read transfer operation.

and sent to the responder. If the r_key and address resolve to a valid range of memory at the responder side, one or more packets will be sent back by the responder containing the data at the address. The data will be written by the responder HCA, to memory previously declared in an MR. Now the WQE can be considered complete, and a CQE can be posted to the CQ.

Responder: The responder application in an RDMA Read transfer operation remains passive during the operation, but the responder HCA does not. When a packet containing an r_key, address, and length is received from the requester, the HCA will look up the corresponding MR and check the virtual address. The message is valid if the r_key resolves to an active MR, the virtual address resolved to valid pinned memory within that MR, and the length does not surpass the boundaries of the MR. If the message is valid, the HCA can read the requested memory with DMA and start transmitting it back to the requester. When the data transmission is complete, no WQEs on the responder side have been consumed from any QP, and no CQEs have been posted to any CQ. If the message from the requester is not valid, it will be discarded.

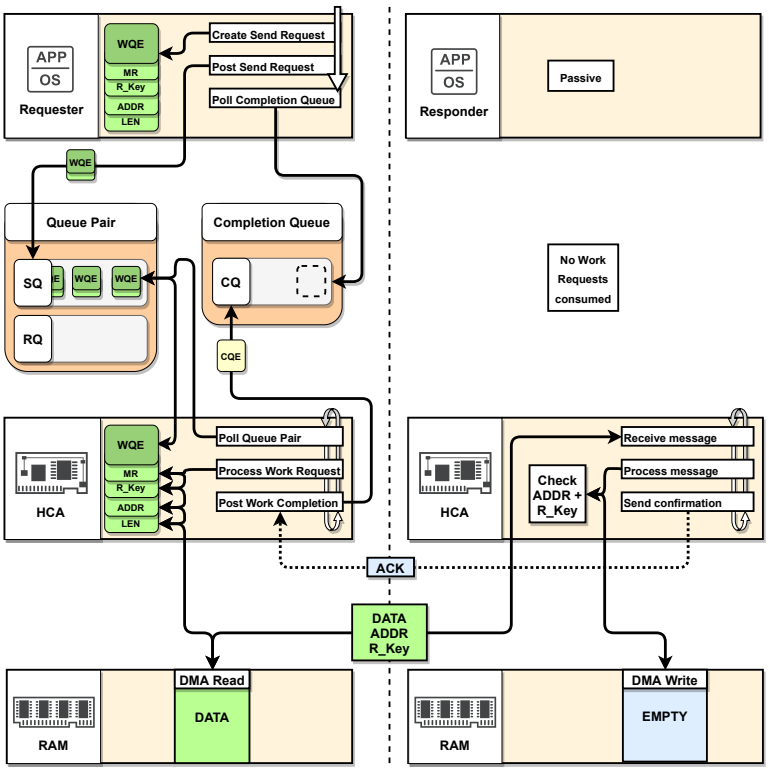

Figure 2.5: RDMA Write transfer operation.

RDMA Write

The RDMA Read transfer operation is mandatory to support over every transport service type an HCA supports. This transfer operation is similar to RDMA Read in that only the requester is active, and the responder kernel is completely bypassed. It is also similar in that data can be gathered from multiple ranges of memory on the requester side, and must be written to one contiguous range of memory on the responder side. Memory to be written to by the requester must previously have been defined as accessible by the responder. Figure 2.5 shows an overview of an RDMA Write Operation between a requester and a responder.

Requester: The requester in an RDMA Write transfer operation produces send requests in the form of WQEs posted to the SQ of a QP. The contents of the WQE are the following:

- **MR:** A handle to a local MR registered by the requester. This MR handle defines one or more ranges of memory to be read by the HCA and transmitted to the responder.
- **r_key:** A key identifying an MR registered by the responder. This key was sent from the responder to the requester at some point before the WQE was created.

- **Address:** A virtual address specifying the the start of the memory buffer to be written to at the responder side. This must be a valid address inside of one of the memory ranges spanned by the MR. It can be either absolute or zero-based (if supported by the HCA).
- **Length:** An integer specifying the number of bytes to write to the address at the responder.

The requester HCA will process each WQE in the order they were posted. When a WQE is processed, one or more packets are created, containing an r_key, an address, a length, and data. If an unreliable transport transmits the data, the operation is considered complete when all packets are transmitted. If the data is transmitted over a reliable transport, the requester HCA will not post a CQE to the CQ until the responder sends an ACK message back. The ACK acknowledges that the message was valid and that all data was received correctly.

Responder: The responder application in an RDMA Write transfer operation remains passive during the operation, but the responder HCA does not. When a transmission is received from the requester, the HCA will look up the MR corresponding to the r_key and check the virtual address. The message is valid if the r_key resolves to an active MR, the virtual address resolved to valid pinned memory within that MR, and the length does not surpass the boundaries of the MR. If the message is valid, the HCA can write the incoming data to the memory range with DMA. When data is written to memory, no WQEs on the responder side are consumed from any QP, and no CQEs are posted to any CQ. If the transport service is reliable, the responder will send an ACK message back to the requester.

2.3 Kernel RDMA subsystem

RDMA is implemented as a native part of the Linux kernel. The majority of RDMA related resources have been gathered into what is usually referred to as the RDMA subsystem. This subsystem has been a part of the Linux kernel since kernel version 2.6.11 [15], and its source code is located at **linux/drivers/infiniband** in the kernel source tree. It is among the most frequently patched subsystems in the Linux kernel and contains about 400 000 lines of code [16]. The subsystem has been divided into four categories: The core RDMA implementation, RDMA device drivers, software RDMA implementations, and implementations of other protocols on top of RDMA. Figure 2.6 shows the entire RDMA software stack, with the relevant contents of the RDMA subsystem in the kernel.

In this thesis, we are using version **5.5.8** of the Linux kernel [17]. At the time of writing, this is a very recent version of the kernel. We believe that using an up-to-date version of the kernel is likely to provide more valuable research than using an older and more stable version of the kernel. Research should, in our opinion, be at the forefront of technology, and not be outdated at the time of publishing. That being said, there is

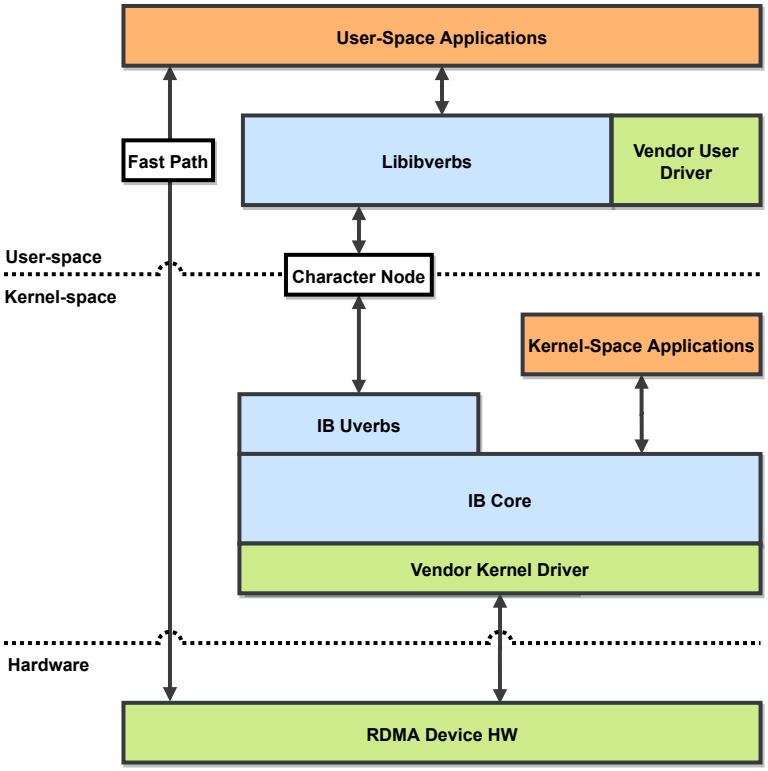

Figure 2.6: The RDMA component stack in a typical Linux system.

also a strong argument for using a version of the kernel that has Long Term Support (LTS). We recognize that using an LTS kernel would make probably makes for a better business case, and could benefit systems that are in deployment now. Using an LTS kernel would also have made parts of the RoPCIe implementation easier, as we ran into issues with software not yet supporting newer kernels.

2.3.1 IB Core

The core implementation of RDMA in the RDMA subsystem is located in a kernel module named IB Core. This kernel module provides the majority of RDMA logic, and is completely device agnostic. The naming of IB Core can can lead to confusion about what role it plays in the RDMA subsystem. IB is an acronym for InfiniBand, but IB Core is not exclusive to InfiniBand. Due to the history of RDMA in the Linux kernel, a lot of systems and functions in IB Core have names related to InfiniBand, or use other nomenclature from the IBTA specification. Some may assume that InfiniBand, RoCE, and iWARP are entirely separately implemented and more or less incompatible with each other, but the three actually share a good deal of code. The three transports can be surmised as follows: InfiniBand is RDMA over InfiniBand transport, RoCE is RDMA over Converged Ethernet, and iWARP is RDMA over TCP/IP. Each of the three share the same core RDMA implementation in IB Core.

The facilities in IB Core are made available to kernel-space applications through a collection of header files. These header files are located at **Linux/include/rdma/** in the kernel source tree, and define the majority of functions and data structures needed by kernel-space applications to do RDMA. One of the header files named **ib_verbs.h** acts as the IB Core main API. Any kernel-space application can include this header to get access to IB Core and use it to do RDMA over any RDMA adapter connected to the system.

Applications do not need to deal with device-specific implementations of RDMA functionality, because IB Core is implemented with another interface towards devices. IB Core has to run RDMA over a variety of transports, and each transport is run by a growing variety of devices like InfiniBand HCAs and RDMA-NICs. The code for running each device can not be tightly coupled with IB Core, as it would result in a massive growth in code and complexity. This issue is dealt with by using device drivers, which separate device-specific code from core code. Before exploring how the RDMA adapter drivers interface with IB Core, another kernel mechanism should first be explained; kernel modules.

2.3.2 Kernel modules

The entire RDMA subsystem is implemented as a collection of kernel modules. A kernel module is a compiled object file that can be loaded into a running kernel, extending its functionality. The dynamic nature of kernel modules makes developing software to run in kernel-space much more straightforward, as the code can be loaded and unloaded at will. It also helps keep the core kernel codebase from growing exponentially, and the number of bugs with it. Kernel modules can be developed in-tree or out-of-tree. In-tree kernel modules, like the ones in the RDMA subsystem, are part of the Linux codebase and reside somewhere in the kernels directory tree structure. These kernel modules are built at the same time as the kernel and loaded when needed. Out-of-tree kernel modules are developed outside the Linux codebase and built against a running kernel.

All kernel modules written for Linux are built, loaded, and unloaded with the **kmod** build system [18]. When a kernel module is built, the compiled code is output in a kernel object file(with the filename extension ".ko"). The kmod build system provides two programs for loading kernel modules: insmod and modprobe. These programs can be used by either the kernel or a user. Modprobe is based on insmod and is more powerful because it can resolve dependencies between kernel modules, and load multiple kernel modules in order of dependency. When a kernel module has served its purpose, the module can be unloaded by the programs rmmod or modprobe with the -r option. This will remove the functionality of the module from the kernel.

An example of a minimal kernel module can be seen in listing 2.1. The program includes the **linux/module.h** header, which is a part of the official Linux-header package. This header supplies two macros, **module_init** and **module_exit**, used to mark an init and an exit function for kmod.

Immediately after the kernel module is built and loaded into the kernel, the init function is called, and the kernel module must set itself up for operation. The load procedure usually includes registering itself with any kernel systems that desire to use its functionality. After the init function completes successfully, the functionality of the module is available to other systems in the kernel. The exit function will be called right before the kernel module is unloaded from the kernel. Here, the kernel module must perform clean-up and free any memory that has been allocated to it.

```
#include <linux/module.h>

static int __init init_func(void)
{
    return 0;
}

static void __exit exit_func(void)
{
}

module_init(init_func);
module_exit(exit_func);
```

Listing 2.1: Minimal kernel module example.

2.3.3 Device drivers

The kernel module mechanism proves its usefulness with device drivers. The nature of a device is that it may be inserted into and removed from the system at runtime, which fits the kernel module mechanism well. A device driver is a kernel module that, when loaded, enables the kernel to operate a specific device or a family of similar devices. [19] Taking the RDMA subsystem as an example, although each device provides similar outward functionality, the internal implementations are very different. These devices are created by different companies, maintained by different developers, and operate different RDMA transports. Some drivers are self-contained, and others rely on secondary drivers and other resources in the kernel. The RDMA device drivers are located at **linux/drivers/infiniband/hw/** in the Linux kernel source tree.

2.3.4 RDMA device context

IB core uses an RDMA device context data structure to represent an RDMA device in RDMA operations. The responsibility of creating and initializing the device context lies with the RDMA device driver. The context is usually created when a device is probed with the device driver, but each RDMA device driver has a different and often complex probe procedure. There are many different approaches to writing a device driver, and this variety can be observed among the RDMA device drivers in the RDMA subsystem. While some RDMA device drivers register a new device context with IB Core when the device is probed, others set up infrastructure (often

device files) to register a new device context when an event is triggered. However, the device context initialization and registration procedure must be executed at some point, and how the context is set up determines how IB Core will handle the device.

An RDMA device driver must have the same device context definition as IB Core. If the two parties had different definitions, bugs would occur when data structures are passed from one party to the other. It could lead to inconsistent evaluations of data structure members, writing garbage values to valid memory, or reading to memory out of bounds. This problem is solved by IB Core defining all its shared data structures in header files located at **linux/include/rdma/** in the kernel source tree. These headers are used in both the IB Core and the device drivers, providing a consistent definition of all shared data structures.

The device context is implemented as one of these shared definition data structures. It has the type **struct ib_device** and is defined in the **ib_verbs.h** header. As the device driver is the "owner" of the device context data structure, it is responsible for both allocating and deallocating it. Memory for the device context can be dynamically allocated with the **_ib_alloc_device** function in IB Core, but this function is typically not used directly. Instead, device drivers use a macro wrapper function **ib_alloc_device** in the **ib_verbs.h** header. The macro wrapper function can allocate memory to any data structure with a member of type **ib_device**. Device drivers use this to implement a device-specific parent device context, extending the base device context with device-specific fields. In summary, **ib_alloc_device** allocates memory for a parent device context data structure, of which **ib_device** is a member.

The **ib_device** context is not ready for registration with IB Core right after its allocation. The device context would not be accepted in its current state, as IB Core requires RDMA devices to support a minimum set of operations. To communicate which operations the device supports, the **ib_device** data structure has a member of type **struct ib_device_ops**. This data structure is defined in the same header as **struct ib_device** and defines function pointers for all possible RDMA operations. IB Core expects the device driver to set function pointers to each of the RDMA operations the device supports. A helper function in IB Core named **ib_set_device_ops** can simplify this step. The device driver can supply the helper function with the device context and a static data structure of type **struct ib_device_ops** where any supported device operations are set, and the helper function will set all the operations in the main device context. This data structure is how RDMA device drivers communicate what operations the device supports.

Now the **ib_device** context is technically ready to be registered with IB Core. The RDMA device drivers will typically configure a good deal more parameters in the device context, but the formal steps to register the device context are complete. The function **ib_register_device** will take the device context and a name for the device as input, and register it with IB Core. Assuming nothing fails in the registration procedure, the device driver has now registered a device with IB Core. From this point on, the device should

be ready to perform its supported RDMA operations.

2.3.5 IB Uverbs

So far, we have discussed a scenario where an RDMA device is being accessed through the IB Core API, which is only available to programs running in kernel-space. Some programs in the kernel do use this API, but the majority of software that uses RDMA exists in user-space. To make user-space RDMA possible, IB Core utilizes a kernel module named IB Uverbs. This module is a character device driver, which means that it manages character device nodes in the file system to communicate between user-space and kernel space.

There are two types of drivers who enable access to hardware devices through special device files; block device drivers and character device drivers. The two types are similar, and both are used as mechanisms to pass information between a kernel module and a user-space application. They differ in how they read and write to the special device files, usually called device nodes. We only encounter character device drivers in the RDMA subsystem, so block device drivers will not be explored in more detail. The following will explain how the IB Uverbs acts as a character device driver.

At module initialization time, IB Uverbs creates a character device node. This node exists as a special file in the file system and can read and written by any user in kernel or user space with the right permissions. The character device node can be used as a bidirectional communication channel where data is written and read as a character stream. The uaccess API in Linux aids character device drivers by providing functionality for reading and writing to the character device node. To make sense of the data passing through the character device node, the character device driver employs an Application Binary Interface (ABI).

Application Binary Interfaces

An ABI can provide interface definitions on a binary level. When IB Uverbs receives a character stream through a character device node, it will use the ABI to interpret the data as commands. The ABI defines byte-aligned data structures to remove any ambiguity about the expected input format. ABIs can be very useful when applications are compiled at different points in time, and on different machines. Some systems may cause the compiler to pad the data structures in different ways, leading to a mismatch between the two sides of an interface. ABIs are also tools used to maintain compatibility between programs and systems as they change and evolve.

ABIs are not needed by the RDMA device drivers when they interface with IB Core, as they are all compiled with the same **ib_verbs.h** header for the same kernel. When multiple programs are compiled with one header file, they share the same definitions of functions and data structures. But user-space applications cannot be compiled with **ib_verbs.h**, which is why ABIs are used. User-space RDMA applications are compiled with a different header file, but the header file has the same definitions of

functions and data structures. Each device driver also provides its own device-specific ABI, where the same custom data structures can be defined for both kernel-space and user-space.

2.4 User-space RDMA libraries

Several libraries providing RDMA services in user-space have been gathered into a single open-source repository named rdma-core. This repository acts as the user-space counterpart to the RDMA subsystem in the kernel, and most of the libraries in rdma-core rely on systems in IB Core. The rdma-core repository includes an automated build system that can compile all the libraries into one package. This system makes the process of downloading the code, modifying it, and building it relatively easy. All the code is open-source, and there are no additional binaries required, making it possible to audit the code and modify any part of it. A pre-built package is also available through the RPM package manager on most major Linux-distributions. When the rdma-core package is installed on a system, RDMA is provided to user-space applications by a library called Libibverbs. The API is available by including **infiniband/verbs.h**. In this thesis, we are using version **27.0** of RDMA-core.

2.4.1 Libibverbs

A library called Libibverbs has become an integral part of user-space RDMA and now serves as the main user-space RDMA API. It was first developed by Roland Dreier as a stand-alone library and has later been integrated with rdma-core, where continued work on the library is being done. Libibverbs mirrors IB Core in many design aspects while being substantially smaller than its kernel counterpart. This is because Libibverbs relies on IB Core for the actual RDMA implementation, and cannot function by itself. The library is often used directly by applications but is also part of various software stacks implementing other protocols on top of RDMA. The popularity of this library, combined with the fact that it shares a device-agnostic design with IB Core, effectively makes Libibverbs an abstraction layer between user-space applications and the RDMA transport. If an RDMA transport integrates with Libibverbs, it can automatically be used by many existing applications and protocol stacks.

Device-specific functionality can be added to Libibverbs by HCA user-space drivers in much the same way as device-specific functionality is added to IB Core by HCA kernel-space device drivers. An HCA user-space driver for Libibverbs is usually called a "user verbs provider," as it does not directly drive the HCA. The user verbs provider can choose to be statically or dynamically linked with Libibverbs, but dynamic linking is strongly recommended. At linking time, the user verbs provider is registered with Libibverbs, providing several device-specific function handles for verbs operations. The user verbs provider will be processed by Libibverbs and matched with a device driver in the kernel, which we will call the "kernel

verbs provider." Like IB Core, Libibverbs does not require a user-space provider to support all verbs operations. The kernel space provider can choose what verbs operations can be supported in user-space, and there is a minimum set of functions required for RDMA to function in user-space. Most major vendors of RDMA-devices maintain a user verbs provider for Libibverbs to make their device available to user-space.

Libibverbs communicates with the RDMA subsystem in the kernel through character nodes managed by IB Uverbs, as described in section 2.3.5. This communication channel is made available to the user verbs provider with a simple kernel command/response API. For each user verbs provider function, there is a Libibverbs kernel command that can be used to reach the corresponding kernel verbs provider function, through a character device node. A provider ABI can be used to ensure that any provider custom data structures have the same definition in kernel-space and user-space. A user verbs provider can use a kernel command for every verbs operation, deferring all complex logic to the kernel verbs provider. The downside to this approach is that communication through a character device node is relatively slow, which is far from ideal in latency-sensitive verbs operations like "post send." The latency is why the kernel command/response path is usually referred to as the "slow path."

There is an alternative "fast path" communication channel used by many user verbs providers to perform latency-sensitive verbs operations. By using the **mmap()** function to memory map resources like WQs and CQs, user verbs providers can read from and write to memory shared with the HCA. This fast path bypasses the kernel entirely for HCAs that can read from and write to the memory mapped resources with DMA. Most verbs operations will still use the slow path, but for the minority of verbs operations that are latency-sensitive, the fast path can be used.

Chapter 3

Design

3.1 Components

The design of RoPCIe can be split into two components; the integration towards the Dolphin Software stack and the integration towards the RDMA software stack.

3.1.1 Dolphin software stack

We use the term "Dolphin software stack" to describe a collection of software layers provided to us by Dolphin ICS. This software stack contains everything needed to program and operate Dolphin PCIe interconnect cards, including APIs, middleware, and drivers. It can be obtained through the Dolphin eXpressWare software package and, once installed, spans user-space and kernel-space. We use the same Dolphin software stack that Dolphin provides to its customers.

3.1.2 RDMA software stack

We use the term "RDMA software stack" to describe a collection of software layers provided to us by the rdma-core software package and the Linux kernel. This software stack contains everything needed to program and operate an RDMA transport, including APIs, middleware, and drivers. It can be obtained by installing rdma-core on a Linux based operating system and, once installed, spans user-space and kernel-space.

3.2 Complexity

The first step we took in the design process of the RoPCIe transport was to identify each separate system that was going to be a part of the final software stack. We evaluated how complex each system was, as well as how critical each system was to create a minimum viable RDMA transport. The task of creating the RoPCIe transport was broken down into a set of sub-tasks. Each sub-task was then assigned a complexity and criticality.

More complex sub-tasks would require more development time, and thus needed to be prioritized by the product of its complexity and criticality.

We evaluated the RDMA software stack to be the largest source of complexity for the implementation of the RoPCIe transport. Writing an RDMA application using a "front end" Verbs API is a non-trivial, but well-documented endeavor. There are multiple open-source examples of such applications, and the verbs interface is thoroughly defined in the IBTA Specification [13]. On the other hand, writing a verbs provider for the "back end" of the RDMA software stack is not well defined or documented. Most information on the provider interface exists as sparse comments in the source code and patch notes in email chains. Additionally, the RDMA stack spans both user space and kernel space with a separate verbs provider interface at each level. The size of the RDMA stack has led to most development time being allocated to interfacing with it.

The Dolphin stack is also a source of complexity and is similar to the RDMA stack in that it spans both user-space and kernel-space. However, our usage of the two stacks differ. While the RoPCIe transport must interface with the "back end" of the RDMA stack, we can use the "front end" of the Dolphin stack. This results in much of the complexity being abstracted away by the Dolphin stack APIs and decreasing the total complexity added by the Dolphin stack. Using the front end of the Dolphin stack has led to less development time being allocated to interfacing with the Dolphin software stack.

3.3 Requirements

During the design of RoPCIe, we set some design requirements that should be fulfilled by the implementation. These requirements have guided the development of RoPCIe and influenced most decisions made regarding scope and architecture. The requirements are broad and few in numbers, due to the complexity of the software stacks obfuscating the path to a correct solution. Making the requirements too restrictive could impede the progress of the implementation, and even result in a failure to achieve a minimum viable RoPCIe transport implementation. However, making the requirements too lenient could lead to an incorrect implementation of little practical value.

The main design requirement is adopted from the verbs-based RDMA design philosophy: An application in user-space or kernel-space should be able to perform RDMA through the standard verbs interface without any knowledge of the transport carrying out data transmission. An application using RoPCIe through a verbs interface should not be aware of the Dolphin stack and experience the same transport behavior as an InfiniBand, RoCE, or iWARP transport would exhibit. Our justification for this requirement is as follows: If an application is required to know about the Dolphin stack in order to perform RDMA over PCIe, it is better off directly using one of the native Dolphin interfaces. Transport-abstraction for high-performance intercommunication is an essential feature of RDMA, and a transport that

must break the abstraction in order to function will not be a lot more useful than bypassing the RDMA stack entirely.

The secondary design requirement is related to the performance of the RoPCIe transport. RDMA is designed for low latency and high throughput, and the transport is essential to achieving this performance. The design of RoPCIe should not significantly sacrifice performance, because inefficiencies in the implementation would quickly limit the applications of this transport service. Making an effort to design and implement the transport to be fast and resource-efficient is a good use of development time, even if a slower and more inefficient implementation would be faster to develop. However, spending significant amounts of time on smaller optimizations not fundamental to the architecture is not a good use of time, and we consider this a subject for future work.

The tertiary design requirement is related to kernel bypass on data transfer. As section 2.1.1 elaborates on, bypassing the kernel is essential to lowering latency, power consumption, and computational load. RoPCIe must be implemented in a way that exhibits the same behavior, using the DMA engine on the RDMA interface adapter to transfer data with as little kernel interaction as possible. This requirement is highly dependent on the transport endpoint hardware and associated firmware and hardware drivers. We have to work within the constraints of the hardware and should attempt to use it to its fullest potential.

3.4 Scope

The scope of this design is mainly focused on a robust architecture for the RoPCIe transport, and implementing a functional prototype. The intention is not to create a direct competitor to InfiniBand, RoCE, or iWARP, but to explore an alternative transport for a PCIe fabric. The scope is limited to functionality facilitating the Send/Receive transfer operation through a verbs interface. The design should allow for expansion to RDMA Read, RDMA Write, and Atomic transfer operations, but the implementations of these are out of scope for this thesis. Finally, the scope also encompasses testing facilities for the transport, and additional miscellaneous tools required to develop, test, and maintain the transport as an open-source project.

3.5 Architecture

3.5.1 Dolphin stack API

The architecture of the first design component, the integration towards the Dolphin software stack, was continually evolved to fit the integration towards the RDMA software stack. We set the limitation of writing all code outside the Dolphin Stack, but the Dolphin stack has several APIs. The following explains which APIS we considered for the RoPCIe transport.

SISCI

The SISCI Library provides a relatively extensive user-space API. The SISCI API gives the programmer control over setting up and performing data transfers with Dolphin hardware. While a low level of API abstraction can be beneficial to achieve excellent performance for a wide range of use-cases, it also introduces a fair amount of code and complexity. For this reason, and because we end up not implementing the verbs provider in user-space, we did not choose to use the SISCI API for the RoPCIe transport.

3.5.2 GENIF

The GENIF Library is very similar to the SISCI Library, but the GENIF API exists in kernel-space. When we decided to implement the RoPCIe transport in kernel-space, this was the library we first considered. However, we got a strong recommendation from Dolphin for using another library instead; the SCI Library.

SCI Library

The SCI Library provides a simple API in both user-space and kernel-space. It revolves around message queues and abstracts away most of the communication setup and management. The low complexity of this API was an attractive feature, and the message queues looked like a good fit for implementing RDMA semantics. This is the reason why we chose the SCI API.

3.5.3 Verbs provider

The architecture of the second design component, the integration towards the RDMA software stack, went through some significant changes before the design converged on a solution. The following describes the architecture of three verbs provider designs we considered, and why we considered them. We use the term "Dolphin software stack" to describe a collection of software layers provided to us by Dolphin ICS. This software stack contains everything needed to program and operate Dolphin PCIe interconnect cards, including APIs, middleware, and drivers. It can be obtained through the Dolphin eXpressWare software package and, once installed, spans user-space and kernel-space. We use the same Dolphin software stack that Dolphin provides to its customers, and make no modifications to any of its software layers.

Libibverbs only provider

The first verbs provider option we considered was a user-space only Libibverbs provider. This provider would only exist in user-space, with no kernel component. Where other verbs providers would perform comparatively little work in the Libibverbs provider, and defer many of the verb operations to the IB Core provider counterpart, we would be using the

Dolphin software stack to operate the transport directly from user-space. We theorized that we could use the SCI Library to bridge the gap between user-space and hardware, which would abstract much complexity away from IB Core in the kernel.

There are several reasons why we investigated this design first. During our research, we found that the majority of RDMA applications exist in user-space and use the user-space verbs interface in Libibverbs (section 2.4.1). This design would result in RoPCIe transport not being available to kernel applications. However, we saw this as an acceptable tradeoff for the simplicity a user-space only verbs provider would bring. In our experience, working in user-space offers significantly less resistance than working in kernel-space, and user-space stuff is, in general, more comfortable to deal with. Creating a user-space only verbs provider was a compelling proposition, as it would make most aspects of the implementation simpler while being usable by the majority of RDMA applications.

Further research of the internal workings of Libibverbs revealed that a user-space only verbs provider is currently infeasible. Any application using Libibverbs must perform some initialization by opening a verbs device through the Libibverbs API to obtain a device context. The user-space provider does not create a verbs device. It is created by Libibverbs iterating over device directories at **/sys/class/infiniband_verbs** in the file system, reading the contents of the files within to initialize the device. These device directories are created and initialized by IB Core in the kernel when a verbs device registers with a verbs device driver. While creating a valid verbs device directory structure from user-space may be technically possible, we determined this course of action to be unnecessarily time consuming and hacky.

Libibverbs/IB Core hybrid provider

When the user-space only verbs provider was shelved, we considered a hybrid implementation spanning both user-space and kernel-space. This hybrid provider would register a verbs device driver with IB Core in the kernel, and when a verbs device was matched with the driver, IB Core could generate the file system infrastructure required by Libibverbs. It may then be possible then implement the majority of the verbs in the Libibverbs provider, with the SCI libraries from the Dolphin stack.

This provider design option was compelling for many of the same reasons as the user-space only driver. Most of the implementation would still exist in user-space, and we may gain some performance by letting the Dolphin stack handle the bridge from user-space to hardware. Having put a considerable amount of time into researching Libibverbs and its inner workings, the hybrid provider was considered mainly due to the sunk cost fallacy, as the smallest alteration to make the original plan feasible.

This hybrid provider design also ended up being shelved, but not because the implementation looked like it was going to be a giant hack. We think that the hybrid provider is feasible to implement, but we realized that it is also unnecessary. When the creation of a device driver to register

with IB Core is a requirement, there is not much point in trying to force the verbs implementation up to user-space. We further reasoned that, at this point, a potential verbs implementation in kernel-space at would look very similar to the hybrid provider verbs implementation in user-space. It is pointless mainly due to the SCI Library working in kernel-space as well, with most of the same functionality. The tools available in the kernel to create performant asynchronous work request processing may also be better, with less overhead. Moving the entire implementation down to kernel-space just made the most sense, and we would gain compatibility with kernel applications like NVMEoF in the process.

IB Core and Libibverbs providers

Finally, we settled on a verbs provider design, where we are integrating with the RDMA stack the "proper" way. This design requires both an IB Core provider and a Libibverbs provider. However, the Libibverbs provider can defer most of the work to the IB Core provider through the internal character node command/response system in IB Uverbs. Because we are more or less integrating with the RDMA stack in an intended way, we believed this design would result in the best fundaments for the RoPCIe transport. Creating a good fundament takes time, but it should make it easier to expand the transport feature set, ensure excellent performance, and operate the transport in compliance with the IBTA specification.

IB Core provider: The IB Core provider is by far the most complex of the two providers. It should be able to operate standalone, meaning it should be able to provide verbs to kernel applications without the presence of a Libibverbs provider. It should also operate more or less like any other IB Core provider within the feature set we choose to support. Stability is a high priority in the design of this provider because trivial bugs in the implementation can hang the users' program indefinitely, or even incite a kernel panic. While dereferencing a null pointer in user-space will segfault the program, in kernel-space, it will lead to an undefined state in the kernel itself. The undefined state can only be fixed by a reboot of the entire system and may lead to loss of data. For these reasons, we think it is wise to keep a tight scope and limit the initial feature set supported by the RoPCIe transport.

When designing the IB Core provider, we looked at existing providers for guidance. Open-source examples of IB Core providers can be found in the Linux kernel source tree, implemented as fully-fledged device drivers. Most of these providers drive the RDMA adapter hardware directly or rely on a secondary driver in the kernel source tree to drive the hardware. We did a surface-level analysis of the Lines Of Code (LOC) of each driver, to get an idea of the size of each driver. The results are presented in table 3.1. We determined that creating a new driver to drive the Dolphin adapter directly would be firmly out of scope for this thesis. The alternative was to create a virtual device driver that would act as a bridge between IB Core and the actual Dolphin NTB device driver.

Driver Name	HW Vendor	Transport	LOC
bnxt_re	Broadcom	RoCE	16 003
cxgb4	Chelsio	iWARP	15 628
efa	Amazon	SRD	5 859
hfi1	Intel	Intel OPA	78 907
hns	Hisilicon	RoCE	24 104
i40iw	Intel	iWARP	28 896
mlx4	Mellanox	InfiniBand	17 235
mlx5	Mellanox	InfiniBand	28 826
mtcha	Mellanox	InfiniBand	15 264
ocrdma	Emulex	RoCE	11 069
qedr	QLogic	RoCE	8 527
qib	Intel	InfiniBand	48 668
usnic	Cisco	RoCE	5 789
vmw_pvrdma	VMware	RoCE	6 187
rdmavt	-	-	9 061
rxe	-	RoCE	12 941
siw	-	iWARP	10 648

Table 3.1: Statistics for RDMA verbs provider drivers in the Linux kernel source tree (v. 5.5.0). Numbers in the Lines Of Code (LOC) column are generated by executing *$ find | xargs wc –lines* in the driver directory, counting every line in every file towards the sum.

Libibverbs provider: The Libibverbs provider is more forgiving than the IB Core provider because it is not required to operate standalone. Its initial role is to relay all user commands to the IB Core provider but should be designed to allow for expansion of responsibility in performance-sensitive operations. These performance-sensitive operations are typically the verbs concerned with posting work requests work to work queues and retrieving work completions from completion queues. The Libibverbs provider should allow for direct fast path operations through memory mapped data structures, similar to existing Libibverbs providers that bypass the kernel entirely in these operations. Fast path operations fall under the category of optimizations we design for but do not prioritize in implementation. The primary function of the Libibverbs provider is to expose the IB Core provider to user-space.

Chapter 4

Implementation

4.1 Source code

The implementation is comprised of two code repositories, a kernel-space provider and a user-space provider. There are also two code repositories with RDMA test programs, one for each provider. While the test programs are not part of the provider implementations, they can aid the understanding of how end-user applications can use these providers. Figure 4.1 places the implementation in each of the four repositories in the RDMA software stack. The following is a description of the contents of each of the four repositories.

The term "dis" or "DIS" appears as a prefix to identifiers throughout the source code of the four code repositories. DIS is an abbreviation of Dolphin Interconnect Solutions and is meant to distinguish RoPCIe code from the verbs API. IB Core prefixes most functions and data structures with "ib," the RDMA Communication Manager uses the prefix "rdma," and Libibverbs uses the prefix "ibv." We observed all other verbs providers using this prefix-based identifier naming scheme and decided to adopt it for our verbs provider code as well.

4.1.1 dis-kverbs

The dis-kverbs repository contains source code for a RoPCIe kernel verbs provider that registers a virtual RDMA adapter interface with IB core and operates the RoPCIe transport. It consists of four kernel modules and a Makefile-based build system to link, build, load, and unload the kernel modules. The four kernel modules are; the bus module, the device driver module, the device module, and the SCI Interface module. The device driver module and the SCI Interface module are located in the same directory, as they are tightly coupled. When loaded in the correct order, the four kernel modules produce a valid RoPCIe kernel verbs provider registered with IB Core. The RoPCIe kernel verbs provider can be used by kernel-space verbs applications or by user-space verbs applications through the user-space provider. When the modules are unloaded in the correct order, the RoPCIe verbs provider will tear down its context,

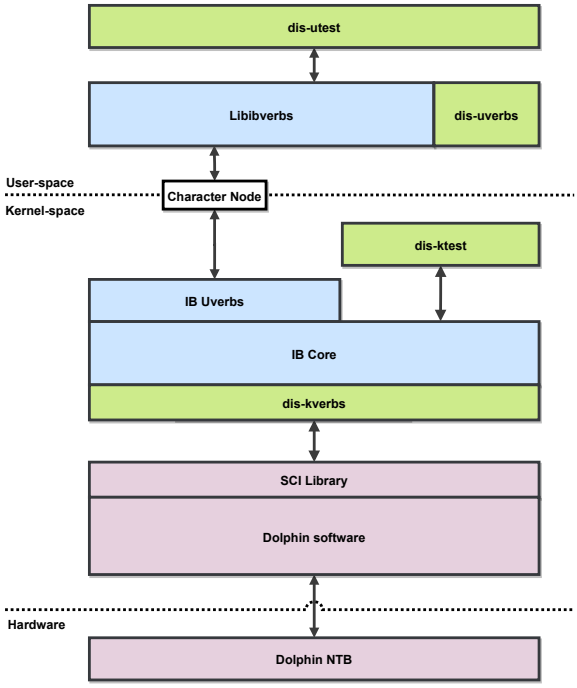

Figure 4.1: The two verbs providers and two test programs produced during the implementation of RoPCIe placed in the RDMA software stack.

unregister from IB Core, and free any allocated memory.

4.1.2 dis-ktest

The dis-ktest repository contains a test program implemented to test the RoPCIe kernel verbs provider through the IB Core API. The test can also be used to test any other kernel verbs provider that supports the same feature-set. It consists of one kernel module that, when loaded on two machines linked with RDMA adapters, will exchange requests and responses between the machines. All messages transmitted between the two machines will be checked byte for byte, to test whether the transport is working correctly. The entire RDMA infrastructure is created in one function, managed by contexts in a header file, and the infrastructure is torn down before the module is unloaded. The test does not currently do any external coordination of keys and remote Queue Pair numbers before creating the infrastructure.

4.1.3 dis-uverbs

The dis-uverbs repository contains a fork of the rdma-core source code repository, where a RoPCIe user verbs provider has been added to a new directory at **providers/dis**. Minor additions have been made to the build system of rdma-core in order to have rdma-core build the RoPCIe user verbs provider as a dynamically linked library along with the existing user

verbs providers. The RoPCIe user verbs provider currently acts as a user-space interface to the RoPCIe kernel verbs provider, as discussed in the design. ’

4.1.4 dis-utest

The dis-utest repository contains a test program implemented to test the RoPCIe user verbs provider through the Libibverbs API. The test can also be used to test any other user verbs provider that supports the same feature-set. It consists of a program that, when executed, runs a test that is very similar to the test in dis-ktest, with the only significant difference being that it creates a user-space memory region. In order to take advantage of the RoPCIe user verbs provider, the program must use the new rdma-core libraries built by dis-uverbs. This can be accomplished by adding the libraries built by dis-uverbs to the system **LD_LIBRARY_PATH**. When multiple libraries with the same base name exist in the path, the newest one with the highest version number will be chosen, even if rdma-core is already installed on the machine. The test does not currently do any external coordination of keys and remote Queue Pair numbers before creating the infrastructure.

4.2 Virtual verbs device

4.2.1 Kernel device model

IB Core is designed to accept device drivers as verbs providers. As one device driver can drive multiple devices simultaneously, each separate device must be registered with IB Core in order to be used in RDMA operations. All registered devices are managed separately within IB Core as unique RDMA transport endpoints with unique sets of capabilities defined by the respective device drivers. Our provider must follow the same pattern of registration in order to operate the RoPCIe transport through IB Core. To accomplish this, we have created a virtual device driver that can register a virtual RDMA adapter interface with IB Core.

To create a valid virtual device, we need an understanding of the kernel device model. A data structure of type **struct device** is the base representation of any external device in the kernel. While the structure is ubiquitous, it is not common for device drivers to use it directly. The base structure is instead encapsulated in a superstructure with additional fields to map features specific to different the type of device targeted by the device driver. Looking at the existing IB Core providers, the device drivers typically represent their devices with device superstructures of types **struct pci_device** (for InfiniBand devices) or **struct net_device** (for RoPCIe and iWARP devices). Each device superstructure has a suite of functions and additional data structures to help the kernel manage the device.

The RoPCIe device driver we have implemented is not directly driving a device, and this is what makes it a virtual device driver. The virtual

device driver is acting as an intermediary between IB Core and a top-level API to the Dolphin NTB device, and the virtual device driver can drive one or more virtual devices as communication instances towards the Dolphin API. As such, the RoPCIe virtual device driver does not have any information about the internal state of the Dolphin NTB device, or its low-level driver machinery. Using a device superstructure like **struct pci_device** and the accompanying functions in the kernel would be unnecessary, so we have elected to use the **struct device** for our provider.

The specific reason we need to create a device is that IB Core requires a valid parent device in order to register as a verbs provider. The formal requirements for registration with IB Core are explained in detail in section 2.3.4, where the device driver must allocate and initialize a data structure of **struct ib_device**. This data structure has a field **dev** of type **struct device**, which in turn needs a reference to a parent structure of the same type. If the device driver does not set a reference to a valid parent structure, the registration with IB Core will fail and incite a kernel panic. This step in the device registration procedure is the reason it is more or less impossible to register a provider without a device driver to generate a valid parent **struct device**.

4.2.2 Dolphin device

Before creating a virtual device driver, we explored retrieving the base device structure from the Dolphin device driver. The Dolphin software stack has several layers, and the base device structure exists at the bottom layer. If we could retrieve this structure, it may be a viable alternative to the virtual device, and the Dolphin adapter could more directly integrate with IB Core. In the end, we could not find any reasonable way of extracting the base device structure from the Dolphin stack without making modifications to the multiple layers of the driver source code. As no tangible performance benefit would be gained from doing this modification, we decided to go with the virtual device option instead.

Exploring the Dolphin driver stack made it clear that the alternative to providing RoPCIe with a virtual device is to integrate the provider code into the dolphin driver itself. While modification of the Dolphin driver stack is out of scope for this thesis, we think this would be a reasonably accomplishable task in and of itself. Our primary concern is the relative immaturity of the RoPCIe provider code, which needs more development and testing before it is ready for a production environment. If the provider is developed further into a more mature state, we see no significant obstacles to integrating it with the Dolphin driver stack and doing away with the virtual device driver entirely.

4.2.3 Bus module

To create a virtual device, we need to load three kernel modules in sequence, and the first module to be loaded is a bus. In the kernel driver model, a bus is a top-level device construct used to match devices with

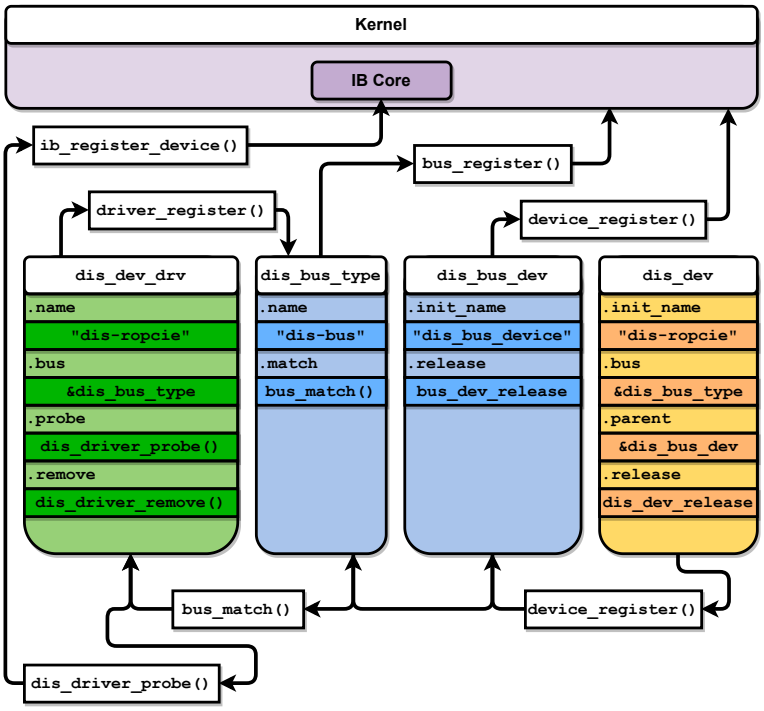

Figure 4.2: Interactions between bus, device driver, and device.

device drivers. When a device is registered with the kernel, it can specify a bus to register with. The internal kernel machinery will find the bus and use it to find the correct driver for the device. A bus can have many registered drivers, so a matching function is used to match driver and device. This matching function is bus-specific, and it is up to the programmer of the bus to create a matching function that determines if a driver is a match for a device. The matching function will be called for every driver registered with the bus until a match is found, or none of the drivers match the device.

When creating the bus for our virtual device, we have to initialize and register two different data structures with the kernel. Figure 4.2 shows an overview of the four main data structures involved in registering our virtual device, and the most important function calls between the structures and the kernel. Two of these data structures, **dis_bus_type** and **dis_bus_device**, represent the bus. When then bus module is loaded, the bus type is first registered with the kernel, and then the bus device is registered. The bus type contains a matching function and will be used to pair the RoPCIe virtual device driver with the RoPCIe virtual device. The bus device is needed to represent the bus as an instanced device in the kernel device hierarchy and will become the parent of our virtual device. When the bus module has finished loading, the bus is ready to be used in the kernel.

The RoPCIe bus matching function is currently straightforward and can be found in listing 4.1. It merely compares the name of the device with the name of the driver. If the driver name is a prefix of the device name, the

device and driver is considered a match. The bus does not need a more complex matching function, because no other devices or drivers can use this bus. We consider the bus to be more or less finished and do not expect its functionality and responsibility to be expanded in the future.

```
// Location: dis-kverbs/src/bus/dis_bus.c
static int bus_match(struct device *dev, struct device_driver *drv)
{
    return !strncmp(dev_name(dev), drv->name, strlen(drv->name));
}
```

Listing 4.1: RoPCIe bus: Matching function.

4.2.4 Device driver module

The second module we load into the kernel is the device driver module. This module contains most of the code necessary to operate the RoPCIe transport, but its loading is uneventful. As shown in figure 4.2, the only action performed during the loading of the device driver module is registration with the bus. The device driver is represented by a data structure of type **struct device_driver**, where the field **bus** is initialized with a reference to a bus type. The bus type reference is obtained in the device driver module by forward declaring an external data structure, which is exported as a symbol by the bus module. At compile time, the forward declaration will resolve to the structure declared in the bus module, and when the device driver module is loaded, the bus module has already initialized the bus type. The device driver can initialize the device driver structure and register with the kernel, which in turn will find the bus corresponding to the bus type and add the device driver to that bus. At this point, the RoPCIe device driver is loaded and will do nothing else until a device is registered with the device driver.

4.2.5 Device module

The third and last module we load into the kernel is the device module, and loading it causes a RoPCIe device to be registered with IB Core. As shown in figure 4.2, the device is initialized with the same name and bus type as the device driver. A device registration call goes to the kernel, which looks at the bus type and invokes a device registration with the corresponding bus. The bus will then match the device with the device driver based on the device name, and when the match is found, the drivers' probe function is called with a reference to the device. This probe functions will accept the device and register it with IB Core. If the probe function returns success, we have created a virtual device and registered it as a verbs transport endpoint.

4.2.6 Register device

The entirety of the IB Core device registration happens in the device driver probe function, and this function is mostly explained in section 2.3.4. The **struct ib_device_ops** data structure supplies IB Core with function pointers

to verb implementations, and it also includes some additional necessary information about the driver. A static initialization of this data structure in the RoPCIe provider is shown in listing 4.2. The driver ID and ABI version fields are used by Libibverbs to match a user verbs provider with the kernel space provider and choose the correct ABI. There are also some **INIT_RDMA_OBJ_SIZE** macros telling IB Core the size of some provider custom data structures, which will be explained later.

```
// Location: dis-kverbs/src/driver/dis_driver.c
static const struct ib_device_ops disdevops = {
    .owner = THIS_MODULE,
    .driver_id = RDMA_DRIVER_UNKNOWN,
    .uverbs_abi_ver = 1,

    .alloc_ucontext     = dis_alloc_ucontext,
    .dealloc_ucontext   = dis_dealloc_ucontext,
    .alloc_pd           = dis_alloc_pd,
    .create_cq          = dis_create_cq,
    .create_qp          = dis_create_qp,
    .dealloc_pd         = dis_dealloc_pd,
    .dereg_mr           = dis_dereg_mr,
    .destroy_cq         = dis_destroy_cq,
    .destroy_qp         = dis_destroy_qp,
    .get_dma_mr         = dis_get_dma_mr,
    .reg_user_mr        = dis_reg_user_mr,
    .get_port_immutable = dis_get_port_immutable,
    .modify_qp          = dis_modify_qp,
    .poll_cq            = dis_poll_cq,
    .post_recv          = dis_post_recv,
    .post_send          = dis_post_send,
    .query_pkey         = dis_query_pkey,
    .query_port         = dis_query_port,
    .query_qp           = dis_query_qp,
    .req_notify_cq      = dis_req_notify_cq,
    .query_device       = dis_query_device,
    .create_srq         = dis_create_srq,
    .modify_srq         = dis_modify_srq,
    .query_srq          = dis_query_srq,
    .post_srq_recv      = dis_post_srq_recv,
    .destroy_srq        = dis_destroy_srq,

    INIT_RDMA_OBJ_SIZE(ib_pd, dis_pd, ibpd),
    INIT_RDMA_OBJ_SIZE(ib_ah, dis_ah, ibah),
    INIT_RDMA_OBJ_SIZE(ib_cq, dis_cq, ibcq),
    INIT_RDMA_OBJ_SIZE(ib_srq, dis_srq, ibsrq),
    INIT_RDMA_OBJ_SIZE(ib_ucontext, dis_ucontext, ibucontext),
};
```

Listing 4.2: RoPCIe kernel verbs provider: Verbs provider structure.

4.3 RoPCIe kernel verbs provider

4.3.1 Provider super structures

Verbs resources are modeled as data structures with a shared definition between IB Core and the providers. An example of this can be found in

listing 4.3, which shows the definition of the CQ verbs resource. While this data structure includes several useful fields, a verbs provider usually wants to extend the structure with some fields specific to the provider.

```
// Location: Linux/include/rdma/ib_verbs.h
struct ib_cq {
    struct ib_device       *device;
    struct ib_uobject      *uobject;
    ib_comp_handler     comp_handler;
    void                  (*event_handler)(struct ib_event *, void *);
    void                   *cq_context;
    int                 cqe;
    atomic_t            usecnt; /* count number of WQs */
    enum ib_poll_context    poll_ctx;
    struct ib_wc        *wc;
    union {
        struct irq_poll     iop;
        struct work_struct  work;
    };
    struct workqueue_struct *comp_wq;
    struct dim *dim;
    /*
        * Implementation details of the RDMA core, don't use in drivers:
        */
    struct rdma_restrack_entry res;
};
```

Listing 4.3: IB Core: Verbs completion queue base structure.

Extending the base verbs data structures with additional fields is natively supported by IB Core, and has become a ubiquitous design pattern among the existing verbs providers. Memory for base verbs data structures like **ib_cq** is directly allocated inside IB Core. For structures like **ib_qp**, this memory allocation is left to the verbs provider. In both cases, we can ensure more memory is allocated immediately after the "end" of the base verbs structure, and the verbs provider can use the extra memory as additional fields in a superstructure of the base structure While it is trivial to extend **ib_qp**, extending **ib_cq** requires IB Core to know the size of the superstructure. We mention this in section 4.2.6, where the provider uses a macro **INIT_RDMA_OBJ_SIZE** to inform IB Core of the size of **dis_cq**, among other superstructures. The definition of the RoPCIe verbs provider superstructure **dis_cq** is shown in listing 4.4.

```
// Location: dis-kverbs/src/driver/dis_verbs.c
struct dis_cq {
    struct ib_cq    ibcq;
    struct circ_buf cqe_circ;
    spinlock_t      cqe_lock;
    u32             cqe_max;
};

...

static inline struct dis_cq *to_dis_cq(struct ib_cq *ibcq)
{
    return ibcq ? container_of(ibcq, struct dis_cq, ibcq) : NULL;
}
```

Listing 4.4: RoPCIe kernel verbs provider: Completion queue super structure and conversion function.

IB Core will not use the definition of the superstructure, and for subsequent operations on a verbs construct, the provider will receive a reference to the base structure. To retrieve the superstructure, the provider can use a conversion function **container_of()**. This function can "upgrade" a base structure reference to a superstructure reference. The RoPCIe verbs provider defines an inline function for each superstructure to make upgrading base structure references easier, as shown in listing 4.4. We use this design pattern throughout the verb functions, and for the CQ example, we use the extended fields to implement the work completion queue.

4.3.2 Protection domains

The PD implementation in the RoPCIe verbs provider more or less follows the PD model in the IBTA specification, explained in section 2.2.3. A PD superstructure **dis_pd** is allocated by IB Core and passed to the RoPCIe verbs provider for initialization. The **dis_pd** superstructure has lists of references to all QPs and MRs linked to that PD and a count of QPs and MRs. Currently, this count is incremented when a new QP or MR is created, and a static limit is defined to comply with the IBTA specification. The count should be decremented when a QP or MR is destroyed, but this is not yet implemented.

4.3.3 Memory regions

The MR implementation is one of the more complex elements of any verbs provider. Due to not mapping HCA registers into memory, the MR implementation in the RoPCIe verbs provider is a little different. A high-level description of what the verb "Register Memory Region" exits in section 2.2.3, but int does not go into the low-level details of implementing these verbs resources. In addition to being complex, the MR implementations differ significantly between verbs providers, because each provider targets different hardware. Herein lies a big difference between the existing hardware device drivers, and the RoPCIe virtual device driver. Hardware device drivers have to map the memory region to the hardware

device, while the RoPCIe virtual device driver leaves this part to the actual Dolphin NTB device driver. As a result, we can somewhat simplify the MR implementation in the RoPCIe verbs provider, and later take advantage of vectored messages in the Dolphin SCI Library.

Register memory region

The code for MR registration in the RoPCIe verbs provider is shown in listing 4.5, and the resulting MR is depicted in figure 4.3. A user initiates the procedure through the verbs interface, inputting a virtual start address, the length of memory to register in bytes, and some access control flags. IB Uverbs eventually call the **dis_reg_user_mr** function, and the RoPCIe provider starts by allocating memory for a new MR superstructure. A function **ib_umem _get** pins the pages containing the user-specified address range, and returns a map of the pinned pages as a data structure of type **ib_umem**. To ensure easy and fast work request processing, we create a list of the physical base addresses to each pinned page in the memory map. We use a list of **struct iovec** data structures to represent the pages, mainly because this is the data structure used by the SCI Library to represent buffers of physical memory. Finally, the MR is initialized and added to a PD with a new l_key. The procedure ends with the RoPCIe verbs provider returning an MR handle to IB Uverbs, and it is eventually passed on to the user.

```
// Location: dis-kverbs/src/driver/dis_verbs.c
    ...
    /* Allocate memory for MR structure */
    mr = kzalloc(sizeof(struct dis_mr), GFP_KERNEL);

    /* Pin the page(s) containing the user segment */
    mr->ibumem = ib_umem_get(udata, start, length, access);

    /* Retrieve the number of pages pinned in previous step */
    mr->page_count = ib_umem_page_count(mr->ibumem);

    /* Allocate memory for a list of physical page addresses */
    mr->pages = kzalloc(sizeof(struct iovec) * mr->page_count, GFP_KERNEL);

    /* Store the base physical address of each pinned page */
    page = mr->pages;
    for_each_sg_page (mr->ibumem->sg_head.sgl, &sg, mr->ibumem->nmap, 0) {
        page->iov_base = (void*)page_address(sg_page_iter_page(&sg));
        page->iov_len  = PAGE_SIZE;
        page++;
    }

    /* Store the MR base virtual address, length, and offset */
    mr->mr_va = va;
    mr->mr_length = length;
    mr->mr_va_offset = ib_umem_offset(mr->ibumem);

    /* Obtain an l_key and register this MR with PD */
    mr->ibmr.lkey = pd->mr_c;
    pd->mr_list[pd->mr_c] = mr;
    pd->mr_c++;

    return &mr->ibmr;
    ...
```

Listing 4.5: RoPCIe kernel verbs provider: MR registration procedure. Variable declaration and error handling is omitted.

Memory region l_key and r_key

The purpose of a memory region l_key and an r_key is explained in section 2.2.3. In the RoPCIe verbs provider, only l_keys are currently implemented, as an index into the PDs MR reference table. An r_key should also be implemented and returned when the user requests an MR with one or more **IBV_ACCESS_REMOTE_*** flags. The r_key should then be transmitted upon QP connection establishment and used to verify access rights of incoming RDMA Read/Write/Atomic requests. Because the RoPCIe transport currently only supports the Send/Receive transport operation, for which an r_key is not required, the current MR implementation is sufficient. For the other transport operations, the r_key is a critical security measure for preventing a program's access to physical memory (including the memory of other programs). It should be noted that it will not prevent a remote QP brute-forcing r_keys to obtain access to an MR which r_key has been registered, but not shared with that QP.

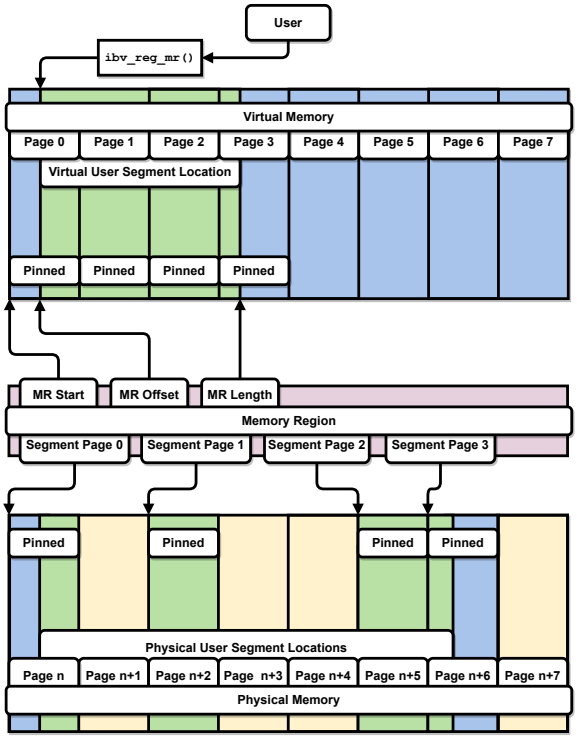

Figure 4.3: Result of RoPCIe MR registration procedure.

Deregister memory region

The MR deregistration procedure is straightforward, as shown in listing 4.6. A function **ib_umem_release** releases all pages pinned by the MR, and the MR memory is freed. This function is ideally called by the user as soon as an MR is no longer in use. It should at least be called before the end of the program. If an MR is not deregistered, the pinned pages will become unusable for any other process until the machine is rebooted.

```
// Location: dis-kverbs/src/driver/dis_verbs.c
    ...
    if (!mr->is_dma) {
        kfree(mr->pages);
        ib_umem_release(mr->ibumem);
    }
    kfree(mr);
    ...
```

Listing 4.6: RoPCIe kernel verbs provider: MR deregistration procedure.

4.3.4 Queue pairs

The QP implementation in RoPCIe behaves as described in section 2.2.3, from the perspective of the user. Users can post work requests to the SQ or the RQ, and the work requests will be processed asynchronously. When a work request is completed, a work completion is posted to a CQ, regardless of work success or failure. The goal for the RoPCIe QP implementation is

to comply with the IBTA specification and therefore behave the way users expect while being as fast and efficient as possible. There are several boxes left to tick before the RoPCIe QP implementation is fully compliant with the IBTA specification. However, this implementation covers the core usage pattern of posting work requests and receiving work completions.

Behind the scenes, the RoPCIe QP implementation is somewhat unique compared to the QP implementations in other verbs providers, especially in the way work requests are consumed asynchronously. Asynchronous work request processing is not only a nice feature of RDMA but an explicit requirement. If the work request processing were synchronous to the user program, verbs operations like "modify queue pair" and "post receive request" would block the user program until the operation was completed. Not only would this eliminate many of the significant benefits of RDMA, but it could deadlock many RDMA applications.

When researching RDMA applications, we found that almost all applications doing actual RDMA transfers contain at least one "QP handshake." A QP handshake is an information exchange between two connected machines where they both post one or more receive requests, then one or more send requests. This pattern is also present in any applications using the RDMA Communication Manager for connection establishment. If the QP implementation were to block on "post receive" during a QP handshake, meaning the "post receive request" would not return until the QP receives a transmission that fulfills that receive request, the two machines would be waiting for each other indefinitely. For this reason, we had to implement asynchronous work request processing in the RoPCIe verbs provider.

Create queue pair

The provider procedure for creating a QP is shown in listing 4.7. Like the MR, QPs are allocated by the provider and registered with a PD. This function initializes the majority of QP attributes, allocates work request queues, and initializes SQ and RQ attributes. When a QP is first created, it exists in the RESET state. The user can post work requests to this QP after creation, but the work requests can not be consumed before the QP is transitioned to the appropriate state. If the QP is transitioned correctly through its states, it will connect to a remote QP and begin to process work requests as they are posted, starting with any backlog from before the QP was transitioned. If no errors occur, a handle to the QP is returned to the user, and QP creation is complete.

```
// Location: dis-kverbs/src/driver/dis_verbs.c
    ...
    /* Allocate memory for QP structure */
    qp = kzalloc(sizeof(struct dis_qp), GFP_KERNEL);

    /* Set QP attributes */
    qp->dev             = to_dis_dev(ibpd->device);
    qp->sq_sig_type     = init_attr->sq_sig_type;
    qp->type            = init_attr->qp_type;
    qp->state           = IB_QPS_RESET;
    qp->mtu             = ib_mtu_enum_to_int(IB_MTU_4096);
    qp->l_qpn           = pd->qp_c;
    qp->event_handler   = init_attr->event_handler;

    qp->ibqp.pd         = ibpd;
    qp->ibqp.send_cq    = init_attr->send_cq;
    qp->ibqp.recv_cq    = init_attr->recv_cq;
    qp->ibqp.srq        = init_attr->srq;
    qp->ibqp.qp_type    = init_attr->qp_type;
    qp->ibqp.qp_num     = qp->l_qpn;

    /* Set SQ attributes */
    qp->sq.ibqp         = &qp->ibqp;
    qp->sq.cq           = to_dis_cq(init_attr->send_cq);
    qp->sq.sge_max      = init_attr->cap.max_send_sge;
    qp->sq.inline_max   = init_attr->cap.max_inline_data;
    qp->sq.wqe_max      = roundup_pow_of_two(init_attr->cap.max_send_wr + 1);
    qp->sq.wq_type      = DIS_SQ;

    /* Allocate memory for SQ */
    qp->sq.wqe_circ.buf  = kzalloc(sizeof(struct dis_wqe) * qp->sq.wqe_max,
                                   GFP_KERNEL);

    /* Set RQ attributes */
    qp->rq.ibqp         = &qp->ibqp;
    qp->rq.cq           = to_dis_cq(init_attr->recv_cq);
    qp->rq.sge_max      = init_attr->cap.max_recv_sge;
    qp->rq.inline_max   = init_attr->cap.max_inline_data;
    qp->rq.wqe_max      = roundup_pow_of_two(init_attr->cap.max_recv_wr + 1);
    qp->rq.wq_type      = DIS_RQ;

    /* Allocate memory for RQ */
    qp->rq.wqe_circ.buf = kzalloc(sizeof(struct dis_wqe) * qp->rq.wqe_max,
                                  GFP_KERNEL);

    /* Register QP with PD */
    pd->qp_list[pd->qp_c] = qp;
    pd->qp_c++;
    return &qp->ibqp;
    ...
```

Listing 4.7: RoPCIe kernel verbs provider: QP creation procedure. Variable declaration and error handling is omitted.

Work queues

There are many different implementations of WQs among the existing verbs providers. Some verbs providers memory map the WQ in user-space to kernel-space, or even to memory accessible by the HCA. In the RoPCIe verbs provider, the WQ is implemented as a circular buffer of work requests. Direct access from user space to the circular buffer is not yet implemented. However, we think this could be an essential future optimization, as it could lower the work request processing latency substantially for small message sizes. Direct access from the HCA would also be possible if a work consumer was run on the Dolphin hardware processor, and this is another possible future optimization. Currently, the work request takes the slow path from user-space through a character node, which can impact latency negatively.

When implementing the actual WQ that would contain work requests, we first examined the WQ implementations in existing verbs providers. Almost all of the implementations were much more complex than what the RoPCIe verbs provider needed, except the Software iWARP (SIW) verbs provider. This verbs provider implements a simple circular buffer, which can offer low latency IPC communication. The buffer can safely be used by one producer and one consumer at the same time, without employing mutual exclusion to serialize access to the buffer. Another critical aspect of the circular buffer is that it can indefinitely occupy the same location in memory at a fixed size because the buffer automatically reclaims memory when items in the buffer are consumed. Figure 4.4 illustrates the WQ as a circular buffer, with a user as a producer, and a work consumer as a consumer.

Doing some more research on circular buffers, we found documentation [20] for a Linux header with useful features for implementing circular buffers in Linux. The documentation outlines a circular buffer implementation with some substantial performance improvements over the SIW verbs provider, as well as some additional coverage of edge cases that should improve reliability. A particular improvement of the Linux circular buffer implementation is utilizing a power-of-two sized buffer. The following is explained in the circular buffer documentation:

> Calculation of the occupancy or the remaining capacity of an arbitrarily sized circular buffer would normally be a slow operation, requiring the use of a modulus (divide) instruction. However, if the buffer is of a power-of-2 size, then a much quicker bitwise-AND instruction can be used instead.

The SIW verbs provider implementation does round up the queue size to a power of two, but for no apparent reason, as they still use the modulus instruction. To lower the computational overhead, we chose the Linux style of circular buffers.

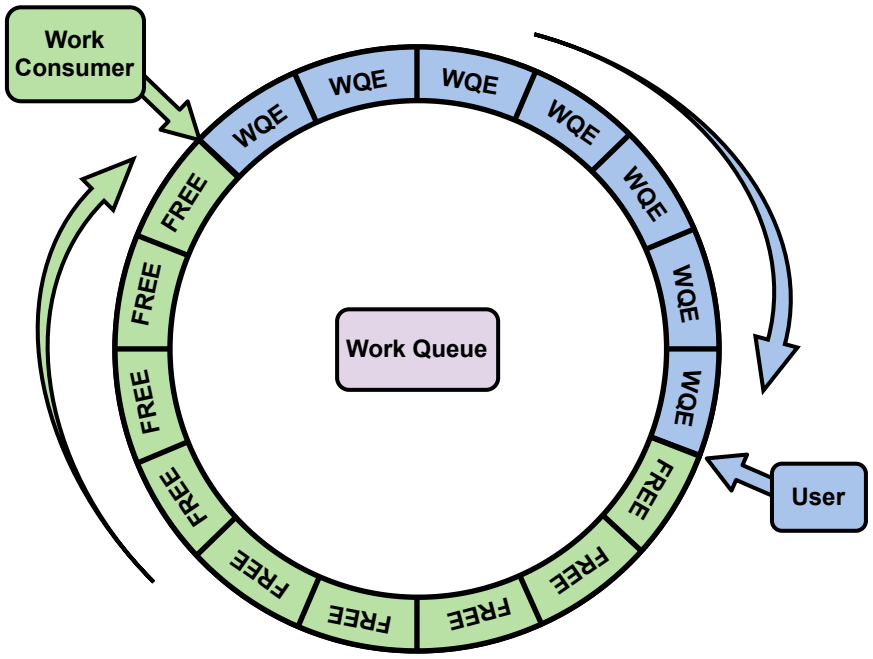

Figure 4.4: A circular buffer where WQEs are produced by a user and consumed by a work consumer.

Post work requests

The code for posting work requests to the SQ and RQ is functionally identical. However, it is split into separate functions because differing identifiers made a shared implementation messy and slightly inefficient. It can be assumed that the term "work request" applies to both send requests and receive requests in this context. The same applies to the WQs, to which the work requests are posted. Areas where the implementation concerning send requests and receive requests differ will be mentioned explicitly.

Post work request list: The RoPCIe verbs provider procedure for posting work requests is split into two functions. The first function, **dis_post_send**, is shown in listing 4.8 and iterates through a list of work requests supplied by the user. This function calls the second function, **dis_qp_post_one_sqe**, for each work request and notifies the work consumer that new work is available in the WQ. We have chosen to notify the work consumer for each work request successfully added to the WQ, as opposed to notifying the work consumer after all work requests have been posted. Notifying the work consumer immediately should lower latency in scenarios where a user supplies a large number of work requests in one call, by starting the work consumer sooner. Posting a work request to the WQ should be orders of magnitude faster than processing them, so this should not result in the work consumer being awoken multiple times. Even if the work consumer was awoken multiple times, it would quickly yield if no new work request were available.

```
// Location: dis-kverbs/src/driver/dis_verbs.c
...
    /* Post all send requests and notify SQ worker thread */
    send_wr_iter = send_wr;
    while (send_wr_iter) {
        ret = dis_qp_post_one_sqe(&qp->sq, send_wr_iter);
        dis_qp_notify(&qp->sq);
        send_wr_iter = send_wr_iter->next;
    }
...
```

Listing 4.8: RoPCIe kernel verbs provider: Post work request procedure.

Initialize work request: When the second function is called, one work request is posted to the WQ. This function is complicated, so it will be separated and explained in parts. The first part, shown in listing 4.9, shows the initialization procedure of the function. The circular buffer API in Linux is used to calculate the remaining capacity of the WQ, and the procedure fails if the WQ is full. Because each WQ only has one producer, there is no need to reserve a WQE with a mutex.

```
// Location: dis-kverbs/src/driver/dis_qp.c
...
    /* Check that circular buffer is not full */
    head = wq->wqe_circ.head;
    tail = READ_ONCE(wq->wqe_circ.tail);
    if(CIRC_SPACE(head, tail, wq->wqe_max) < 1) {
        return -42;
    }
    wqe = (struct dis_wqe*)&wq->wqe_circ.buf[head * sizeof(struct dis_wqe)];

    /* Set WQE attributes */
    wqe->opcode         = IB_WC_SEND;
    wqe->sci_msq        = &wq->sci_msq;
    wqe->byte_len       = 0;
    wqe->ibqp           = wq->ibqp;
    wqe->wr_id          = wr->wr_id;
    wqe->sci_msg.iovlen = 0;
...
```

Listing 4.9: RoPCIe kernel verbs provider: Post one work request procedure - initialization.

Calculate number of pages per segment: A work request can describe a list of memory segments(named "sge" in the code) to be used in send/receive operations. Each segment has a virtual start address, a length, and an l_key to identify which MR has the physical memory mapping for that segment. Before we can map each segment to physical memory, we need to calculate the combined amount of pages spanned by all the segments in this work request. The code for this step is shown in listing 4.10, and will be explained in detail. Calculating the number of pages before mapping them is necessary because the SCI Library requires a contiguous list of **struct iovec** data structures to represent each buffer in

a message. This list will be long for large segments, and attempting to statically allocate a large amount of **struct iovec** data structures for each WQE in the WQ, will cause the memory footprint of the WQ to grow significantly. We have elected to allocate a small amount of **struct iovec** data structures statically, and if the amount of pages exceeds this limit, we instead dynamically allocate memory for the list. When the work request is completed, the memory allocated for the list is freed.

The number of pages spanned by one segment is calculated and stored in a helper data structure of type **dis_sge_map**. First, the virtual start address of the segment is translated to a "segment base offset", which is an offset into the segments MR. Recollect that the MR can contain gaps between the pages it maps, so the segment base offset is not a usable physical address. Next, we need to calculate how many pages into the MR the segment base offset places us, which we call the "page base offset." This offset is easily calculated by dividing the page base offset integer with the size of a page because integer division in C truncates the result. We then lower the segment base offset by the number of pages we are "skipping," causing the segment base offset to become an offset into the first page populated by the segment. Finally, we can calculate the number of pages spanned by this segment. This number is calculated with the same integer division trick we used before; this time applied to the sum of the segment base offset and the segment length. We have to add one to the result of this division, because the result will be truncated to the number of page boundaries crossed by the segment, but we want to include any partially filled pages as well.

```
// Location: dis-kverbs/src/driver/dis_verbs.h
struct dis_sge_map {
    struct iovec *mr_pages;
    u64 base_offset;
    u64 page_offset;
    u64 page_count;
    u64 sge_len;
    u8  is_dma;
};

// Location: dis-kverbs/src/driver/dis_qp.c
...
    /* Calculate the number of pages spanned by each segment */
    for (i = 0; i < min(wr->num_sge, DIS_SGE_PER_WQE); i++) {
        ibsge = &wr->sg_list[i];
        sge = &sge_map[i];
        mr = pd->mr_list[ibsge->lkey];

        sge->sge_len        = ibsge->length;
        sge->base_offset    = (ibsge->addr - mr->mr_va) + mr->mr_va_offset;
        sge->page_offset    = sge->base_offset / PAGE_SIZE;
        sge->base_offset    -= sge->page_offset * PAGE_SIZE;
        sge->page_count     =
            ((u64)(sge->sge_len + sge->base_offset) / PAGE_SIZE) + 1;

        sge->mr_pages       = mr->pages + sge->page_offset;
        wqe->sci_msg.iovlen += sge->page_count;
    }
...
```

Listing 4.10: RoPCIe kernel verbs provider:Post one work request procedure - calculate number of pages per segment.

Choose store for page map: As described in a previous step, we can use either statically or dynamically allocated memory for the page map. The code for choosing a page map store is shown in listing 4.11. Using the static page map is faster because we avoid one memory allocation, but we have to set a limit on how many page addresses the static page map can store. This limit has been set to be the same as the maximum number of segments supported for a work request. The page count obtained in the last step is accumulative across all segments, meaning that a work request with one segment of multiple pages can still use the static map if it is within limits. If not, the dynamic map will be chosen in this step, and memory will be allocated for the page map. We cannot use both the static and dynamic maps because the SCI Library requires the buffer to be contiguous.

```
// Location: dis-kverbs/src/driver/dis_qp.c
...
    /* Choose store for the WQE page iovec map */
    if(wqe->sci_msg.iovlen <= DIS_SGE_PER_WQE) {
        wqe->sci_msg.iov = wqe->page_map_static;
    } else {
        wqe->page_map_dynamic = kmalloc(sizeof(struct iovec)*wqe->sci_msg.iovlen,
                                        GFP_KERNEL);
        wqe->sci_msg.iov = wqe->page_map_dynamic;
    }
...
```

Listing 4.11: RoPCIe kernel verbs provider: Post one work request procedure - choose store for page map.

Map segments to physical pages: In the final step, we map each segment to physical pages, and this is shown in listing 4.12. Here we introduce the term "chunk" to describe the number of bytes a segment occupies in a specific page. A large segment will usually be mapped with a starting chunk smaller than the page size, then a series of chunks at page size, then a final chunk smaller than the page size. All pages occupied by a segment are guaranteed to be filled, except for the first and the last page. In order to point to the start of the segment, an offset is applied to the first page. After the WQE page map store has been filled with all segments in page chunks, the procedure is finished, and the WQ circular buffer can be advanced. This concludes the post work request procedure.

```
// Location: dis-kverbs/src/driver/dis_qp.c
...
    /* Map each segment into physical pages from the MR */
    sge_page = wqe->sci_msg.iov;
    for (i = 0; i < min(wr->num_sge, DIS_SGE_PER_WQE); i++) {
        sge = &sge_map[i];
        mr_page = sge->mr_pages;

        while (sge->sge_len > 0) {
            sge_chunk = min(sge->sge_len, PAGE_SIZE - sge->base_offset);
            sge_page->iov_base  = mr_page->iov_base + sge->base_offset;
            sge_page->iov_len   = (size_t)sge_chunk;
            wqe->byte_len       += sge_chunk;
            sge->sge_len        -= sge_chunk;
            sge->base_offset    = 0;
            sge_page++;
            mr_page++;
        }
    }

    /* Advance the head of the circular buffer */
    smp_store_release(&wq->wqe_circ.head, (head + 1) & (wq->wqe_max - 1));
...
```

Listing 4.12: RoPCIe kernel verbs provider: Post one work request procedure. Variable declaration and error handling has been omitted

State transitions

The QP state transitions described in section 2.2.3 are mostly handled by IB Core, which has a state transition table to ensure all mandatory attributes are supplied for each transition. After the transition is validated in IB Core, the **dis_modify_qp** RoPCIe verbs provider function is called. This function currently does nothing for the transition to RESET and INIT states. For the transition to RTR state, the RoPCIe verbs provider starts a work consumer for the RQ with a function **dis_qp_init** shown in listing 4.13. This QP modification is also where the user will provide the destination QP number, which will be used later to connect to the remote machine. The RoPCIe verbs provider also starts a work consumer for the SQ during the RTR state transition, to avoid having a constantly polling message queue in programs where the responder is only transitioned to the RTR state.

```
// Location: dis-kverbs/src/driver/dis_qp.c
...
    /* Initialize wait queue */
    init_waitqueue_head(&wq->wait_queue);
    wq->wq_state = DIS_WQ_INITIALIZED;
    wq->wq_flag  = DIS_WQ_EMPTY;

    /* Create work consumer */
    wq->thread = kthread_create(dis_wq_thread, (void*)wq, "DIS WQ Thread");
    if (!wq->thread) {
        wq->wq_state = DIS_WQ_UNINITIALIZED;
        return -42;
    }

    /* Start work consumer */
    wake_up_process(wq->thread);
...
```

Listing 4.13: RoPCIe kernel verbs provider: Work consumer init procedure.

Destroy queue pair

The QP teardown procedure in the RoPCIe verbs provider must stop the work consumers we started in . As shown in section 4.14, the teardown is performed by commanding the work consumers to stop. The main thread will then actively wait for the thread to set its state to **DIS_WQ_EXITED**. Waiting for thread exit is necessary because the work consumer accesses memory in the **dis_qp** data structure, and will only start its exit procedure the next time it calls a function **kthread_should_stop()**. If the QP memory was freed without waiting for the thread to stop, the thread could be accessing memory that is no longer valid.

```
// Location: dis-kverbs/src/driver/dis_qp.c
...
    /* Command the worker thread to stop and wait for it to exit */
    kthread_stop(wq->thread);
    wake_up(&wq->wait_queue);
    while (wq->wq_state != DIS_WQ_EXITED) {
        msleep(1);
    }
...
```

Listing 4.14: RoPCIe kernel verbs provider: Work consumer exit procedure.

4.3.5 Completion queues

The CQ implementation in the RoPCIe verbs provider behaves as described in section 2.2.3, from the perspective of the user. When work requests are consumed, a work completion is added to the CQ associated with that WQ. The performance of the CQ implementation is important because users may wait for one or more completions before continuing to do work. Getting the work completions to the user as fast and efficiently as possible is crucial. Currently, the verb "poll completion queue" is implemented in the RoPCIe verbs provider as a means for users to retrieve new work completions.

Create completion queue

The RoPCIe verbs provider procedure for creating a CQ is shown in listing 4.15. First, a spinlock is initialized, which will be used to ensure serialized access to the CQ between multiple work completion producers(work consumers). The CQ uses the same circular buffer implementation as the WQs in the QP, buffering CQEs instead of WQEs. While the WQ implementation is designed for only one producer and one consumer, the CQ must be capable of handling multiple producers, as shown in figure 4.5. The work completion capacity is rounded up to a power of two, and memory is allocated for this buffer capacity. After this operation is completed successfully, the CQ is ready to be used.

```
// Location: dis-kverbs/src/driver/dis_verbs.c
...
    /* Initialize CQ attributes */
    spin_lock_init(&cq->cqe_lock);
    cq->cqe_max = roundup_pow_of_two(init_attr->cqe + 1);
    ibcq->cqe = cq->cqe_max;

    /* Allocate memory for CQE buffer */
    cq->cqe_circ.buf = kzalloc(sizeof(struct dis_cqe) * cq->cqe_max,
                               GFP_KERNEL);
...
```

Listing 4.15: RoPCIe kernel verbs provider: Completion queue creation procedure.

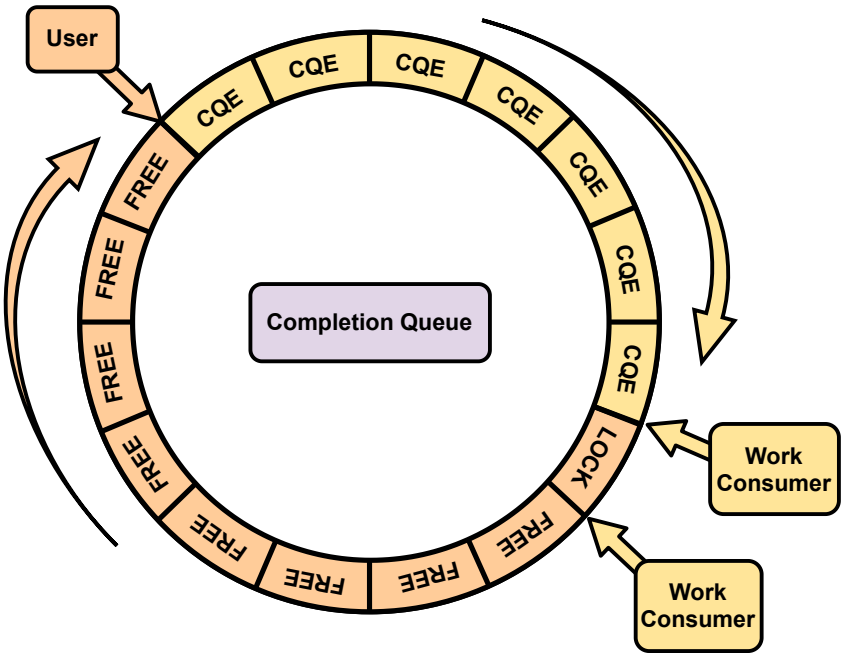

Figure 4.5: A circular buffer where CQEs are produced by a work consumer and consumed by a user.

Poll work completions

The "poll completion queue " verbs operation is the most commonly used method for users to retrieve work completions from the CQ. Users must supply the number of work completions they want to consume from the CQ, and the function will result in the consumption of zero or more work completions. It is normal for a CQ not to have enough work completion in the buffer to satisfy the user, so the user may repeatedly poll the CQ until some number of work completions have been consumed. The CQ polling operation should not block until enough work completions have been produced, and ideally, be as fast as possible. It is normal for users to poll the CQ many times within a short period.

The RoPCIe verbs provider procedure for polling a CQ is shown in listing 4.16. Here, CQEs will be consumed from the circular CQ buffer until the buffer is empty, or the user is satisfied. For each work completion, attributes are copied over from our custom CQE data structure to a user-allocated buffer for work completions. We do not use the **ib_wc** data structure defined by IB Core because it contains several fields we do not currently need in the RoPCIe verbs provider. To save memory, we chose to define a separate structure for the circular CQE buffer. However, using the **ib_wc** data structure could be a better overall solution, as the entire structure could be copied over to the user buffer with a very fast **memcpy()**. The sanity of this decision depends on how a memcpy compares to the current field-by-field copying, and how important it is to save memory.

```
// Location: dis-kverbs/src/driver/dis_verbs.c
...
    /* Consume num_wc CQEs from the CQ */
    ibwc_iter = ibwc;
    for (i = 0; i < num_wc; i++) {
        /* Check if circular buffer is empty */
        head = smp_load_acquire(&cq->cqe_circ.head);
        tail = cq->cqe_circ.tail;
        if(CIRC_CNT(head, tail, cq->cqe_max) < 1) {
            break;
        }
        cqe = (struct dis_cqe*)&cq->cqe_circ.buf[tail * sizeof(struct dis_cqe)];

        /* Set work completion attributes */
        ibwc_iter->wr_id    = cqe->wr_id;
        ibwc_iter->status   = cqe->status;
        ibwc_iter->opcode   = cqe->opcode;
        ibwc_iter->byte_len = cqe->byte_len;
        ibwc_iter->qp       = cqe->ibqp;

        smp_store_release(&cq->cqe_circ.tail, (tail + 1) & (cq->cqe_max - 1));
        ibwc_iter++;
        wc_count++;
    }
...
```

Listing 4.16: RoPCIe kernel verbs provider: Poll completion queue procedure.

4.3.6 Work consumers

A work consumer is an active process that consumes work requests from a WQ, fulfills the work request, produces a work completion. In the existing hardware verbs providers, this process is run entirely by a processor on the HCA hardware. In contrast, the existing software verbs providers run this process entirely on the CPU. The RoPCIe verbs provider share the work between the CPU and a processor on the dolphin NTB. Sharing this work makes RoPCIe unique among the verbs providers and places it somewhere in the middle of a hardware verbs provider and a software verbs provider. It would be reasonable to say that RoPCIe is closer to a hardware verbs provider, as the Dolphin hardware does the majority of work.

The first part of the RoPCIe work consumer, which is run by a CPU on the host machine, has the goal of completing its task as quickly and efficiently as possible. The task is as follows: Consume a work request from a WQ, pass this on to the Dolphin NTB, wait for the work request to be completed, and post a work completion to a CQ. The usage of a CPU based work consumer may introduce an additional latency overhead to the work request processing path. Ideally, the work consumer would be run by the Dolphin device driver to make the RoPCIe a full hardware verbs provider, but this is out of scope for this thesis.

The second part of the RoPCIe work consumer is run by either the CPU on the host machine or a processor on the Dolphin NTB. What processor is

used depends on whether the messages passed to the Dolphin SCI Library has been configured to be completed with DMA or Programmed Input-Output (PIO). These modes are discussed in more detail in section 4.3.7.

Worker implementation

When implementing the work consumer in the RoPCIe verbs provider, we had to select a worker implementation. The Linux kernel has multiple different facilities for doing work asynchronously, and it was not immediately obvious which to use for this task. After some initial research, we considered using either tasklets, workqueues, kthreads, or some combination of the three. In the end, we decided to use kthreads exclusively. We think this is the best worker option for our use case, but we have not implemented all three options to test which one is better. This choice may warrant more study, as the worker implementation can have a significant impact on latency and performance of the transport.

Tasklets

The first worker option we considered was tasklets. Tasklets are the "smallest" worker option, as they are not threads, but dynamically registrable software interrupts. A tasklet cannot sleep and is intended to perform a task quickly and completely. In other words, tasklets are not meant to stay alive for a long time. While tasklets may not sound like a viable worker solution for a verbs transport, it is the implementation used by the RXE software verbs provider to operate Soft-RoCE transport. However, the Soft-RoCE transport is not implemented with efficiently as a primary goal, so the fact that RXE is using tasklets did not weigh heavily in our choice of worker implementation. Also, the RoPCIe transport could require the worker to be polling for a connection to a remote computer for a relatively long time, which does not sound like an ideal use case for tasklets. In the end, we decided against tasklet workers, as we suspected this implementation could have some hidden complications, especially in interactions with the Dolphin SCI Library.

Workqueues

The second worker option we considered was based on workqueues. The kernel has a workqueue API that offers a construct for "posting" work to a queue, which will eventually be performed by a worker thread (kthread). At first, it seemed like workqueues solved more than our work consumer problem, and that workqueues could even replace the SQ and RQ implementations in the RoPCIe verbs provider. In the current implementation of RoPCIe, workqueues could probably serve the same functional purpose as WQs with circular buffers and kthreads. However, what RoPCIe would gain in simplicity, it would lose in flexibility and performance. The circular buffer does not only serve as a queue but also as a buffer of already allocated memory. Workqueues do not offer a buffer

of memory for intermediary storage for work requests, which leaves us with two options. The first option is to allocate memory for each work request to be posted, but this would be unacceptably slow. The second option is to implement a circular memory buffer for work requests, but this somewhat defeats the purpose of the workqueues. We concluded that workqueues were not well suited for our task, and decided to implement a work consumer with kthreads.

KThreads

The third and final worker option we considered was kthreads. This option may require a bit more work than the two other options, but in return, offers much greater flexibility and control. Implementing a work consumer with kthreads meant that one WQ could have one dedicated kthread, that is started at the moment the QP state is transitioned into the RTR state, and lives until the QP is destroyed. However, with great control comes great responsibility, and our experience working with kthreads has included multiple difficult to debug race conditions, infinite loops and kernel panics. Despite this, we think kthreads was the right approach for the RoPCIe work consumers.

Wait queues

Having the RoPCIe work consumers poll the WQs actively to see if any new work requests have been posted would be either wasteful or slow, depending on the polling frequency. Instead, we wanted to use wait/notify semantics, to have the kthread in a dormant state when no work is available, and wake up the thread as fast as possible when new work is posted. After some research, we decided that wait queues are the best solution to achieve this goal. Similarly to workqueues, wait queues are a part of the Linux kernel API. It offers a wait queue construct and many different functions for waiting for an event on that wait queue.

We use the term "wait type" to describe the semantics of a wait function in the wait queue API. When a kthread waits on a wait queue, it yields and is taken out of the scheduling queue. Depending on the wait type, different events can cause the kthread to be rescheduled. Some wait types will reschedule the thread on spurious signals or timeouts, and some will only wake up when directly awoken. We have chosen the "killable" wait type, which will only reschedule the kthread when the wait queue is directly woken up, or when the kthread receives a fatal signal. This wait type saves us from having to handle spurious signals, while still correctly shutting down in the event of a fatal signal.

The main work consumer loop is shown in listing 4.17, and this is where the kthread work consumer waits on the wait queue with a function **wait_event_killable**. If this function returns anything other than 0, it means that the kthread received a fatal signal, and should terminate immediately. The kthread is also awoken if it receives a direct wake-up, and the condition in the wait event evaluates to true. In the RoPCIe

work consumer implementation, the wait queue is awoken directly with a function **wake_up**, when a new work request is posted (shown in listing 4.8), and when the QP is destroyed(shown in listing 4.14). If the WQ flag is set to **DIS_WQ_POST**, one or more work requests have been posted, and if the function **kthread_should_stop()** returns true, the work consumer has been told to stop.

```
// Location: dis-kverbs/src/driver/dis_qp.c
...
    /* Initialize connection to remote QP */
    wq->wq_state = DIS_WQ_RUNNING;
    ret = dis_wq_init(wq);

    while (!kthread_should_stop()) {
        /* Process all new work requests */
        wq->wq_flag = DIS_WQ_EMPTY;
        ret = dis_wq_consume_all(wq);
        if (ret) {
            break;
        }

        /* Wait for new work requests to be posted */
        ret = wait_event_killable(wq->wait_queue, wq->wq_flag != DIS_WQ_EMPTY ||
                                  kthread_should_stop());
        if (ret) {
            break;
        }
    }

    /* Tear down connection to remote QP */
    dis_wq_exit(wq);
    wq->wq_state = DIS_WQ_EXITED;
...
```

Listing 4.17: RoPCIe kernel verbs provider: Main work consumer wait/event loop.

Consume work request

Each time the work consumer is awoken due to a work request being posted, it will start consuming all work requests in the WQ. This procedure is shown in listing 4.18, where WQEs are consumed repeatedly from the circular buffer until it is empty, or the kthread is told to stop. As a general guideline, a kthread should periodically call the **kthread_should_stop()** function to check if it should terminate. If this is not called often, the kthread could take a long time to exit, which is not good, as it will reduce the overall responsiveness of the system. After a work request is processed, a work completion is posted to the CQ associated with the WQ. If the WQ is empty, the function will return to the main thread loop, and the work consumer will wait for the next wake up.

```
// Location: dis-kverbs/src/driver/dis_qp.c
...
    while (!kthread_should_stop()) {
        /* Check if circular buffer is empty */
        head = smp_load_acquire(&wq->wqe_circ.head);
        tail = wq->wqe_circ.tail;
        if(CIRC_CNT(head, tail, wq->wqe_max) < 1) {
            return 0;
        }
        wqe = (struct dis_wqe*)&wq->wqe_circ.buf[tail * sizeof(struct dis_wqe)];

        switch (wq->wq_type) {
        case DIS_RQ:
            wc_status = dis_wq_consume_one_rqe(wqe);
            break;
        case DIS_SQ:
            wc_status = dis_wq_consume_one_sqe(wqe);
            break;
        default:
            return -42;
        }

        dis_wq_post_cqe(wq, wqe, wc_status);

        /* Advance the tail of the circular buffer */
        smp_store_release(&wq->wqe_circ.tail, (tail + 1) & (wq->wqe_max - 1));
    }
...
```

Listing 4.18: RoPCIe kernel verbs provider: Work consumer - consume all work reqeusts procedure.

The procedure for consuming one work request is simple. It will repeatedly call the appropriate function in the Dolphin SCI Library until the work has been completed, or the work consumer is told to stop. It will return a work completion status, which will be included in the work completion posted to the CQ. The procedure for posting a send request to the SQ is shown in listing 4.19. This procedure is very similar to the procedure for posting a receive request to the RQ, with the only significant difference being that the receive function in the Dolphin SCI Library may have to be called multiple times in order to receive a large message. Because of this behavior, we create some additional variables to ensure we receive the whole message.

```
// Location: dis-kverbs/src/driver/dis_qp.c
...
    while (!kthread_should_stop()) {
        ret = dis_sci_if_send_v_msg(wqe);
        if (!ret) {
            return IB_WC_SUCCESS;
        }
    }
    return IB_WC_RESP_TIMEOUT_ERR;
...
```

Listing 4.19: RoPCIe kernel verbs provider: Work consumer - consume one send reqeust procedure.

Post work completion

A work completion is posted to a CQ by a work consumer for every work request consumed. The procedure for posting a CQE to a CQ is shown in listing 4.21. This procedure is similar to posting a work request to a WQ because they share a more or less identical circular buffer implementation. One difference in the work completion procedure is the use of spinlocks for mutual exclusion. This is necessary because multiple WQs can be associated with the same CQ, and therefore multiple work consumers may try to post a work completion to the CQ at the same time. Without mutual exclusion for serialized access, multiple CQEs may be written to the same memory location. This would result in corrupted and lost work completions. We considered implementing a lock for every CQE, which would allow many work consumers to post to the CQ at the same time. However, we see this as an edge case that could be optimized if a use case arises that shows locking on work completion posting to be a bottleneck.

```
// Location: dis-kverbs/src/driver/dis_qp.c
...
    spin_lock_irqsave(&cq->cqe_lock, flags);

    /* Check that circular buffer is not full */
    head = cq->cqe_circ.head;
    tail = READ_ONCE(cq->cqe_circ.tail);
    if(CIRC_SPACE(head, tail, cq->cqe_max) < 1) {
        spin_unlock_irqrestore(&cq->cqe_lock, flags);
        return -42;
    }
    cqe = (struct dis_cqe*)&cq->cqe_circ.buf[head * sizeof(struct dis_cqe)];

    cqe->wr_id      = wqe->wr_id;
    cqe->opcode     = wqe->opcode;
    cqe->byte_len   = wqe->byte_len;
    cqe->ibqp       = wqe->ibqp;
    cqe->status     = wq_status;

    /* Advance the head of the circular buffer */
    smp_store_release(&cq->cqe_circ.head, (head + 1) & (cq->cqe_max - 1));
    spin_unlock_irqrestore(&cq->cqe_lock, flags);
...
```

Listing 4.20: RoPCIe kernel verbs provider: Work consumer - post work completion procedure.

Combined consecutive pages

We attempted to increase the performance of the RoPCIe transport by detecting consecutive pages in the MR and combining them into one buffer. We assumed that a given work request would be mapped to a smaller amount of longer buffers and that longer buffers would enable the SCI Library to utilize optimizations like cache lines more effectively.

Having rewritten both the MR and work request implementations to use combined consecutive pages, we got some unexpected results. The first thing we noticed was that the pages in the MR were heavily fragmented, with a few larger chunks of consecutive memory. This issue persisted when running a test program immediately a reboot of the machine, leading us to believe that the system boot procedure is causing memory fragmentation.

The second thing we noticed was that the performance of the RoPCIe transport was significantly reduced. Latency in for both small and large message sizes was increased, meaning that the overhead scaled with message size. The reason for this behavior probably lies in the Dolphin SCI Library implementation, but we have not invested more time in researching this. We reverted our implementation to use single pages, and the performance of the RoPCIe was restored.

4.3.7 SCI Library interface

In order to complete work requests, work consumer needs to call functions in the Dolphin SCI Library API. To do so, we have implemented an SCI

Library interface kernel module shortened to the "SCI IF module," separate from the driver module. The SCI IF module wraps all the functions from the SCI Library API needed by the work consumer and makes the wrapper functions available to the work consumer through exported symbols. While this gives the code structure some separation of function, it also introduces an extra function call into the call chain of processing a work request, which can lead to a small amount of overhead. We are willing to accept this overhead because the kernel module currently relieves a lot of context management and argument passing. Given more development time, this module should probably be integrated into the driver code.

A minor reason for creating a separate SCI IF module is the utility brought by the module init and exit procedures. Before the Dolphin SCI Library API can be used to create or connect to any message queues, a function **SCILInit()** must be called once to initialize the SCI context. The SCI IF module calls this function when loaded into the kernel, ensuring the interface is ready to use once the module is loaded. A function **SCILDestroy()** must be called to tear down the SCI context, which is done when the SCI IF module is removed from the kernel. While this would be possible to do at some point in the driver module, it is much cleaner to do so in the SCI IF module, and it allows us to remove and reload the driver module without tearing down the SCI context.

The primary reason for creating a separate SCI IF module is the utility brought by the module parameters. The SCI IF module can take a series of arguments, which are used when establishing a message queue connection with a remote machine. These parameters have been beneficial to the development and testing of the RoPCIe verbs provider. However, we recognize that the functionality the module parameters provide should come from either the driver or the SCI Library. The first two parameters, "local adapter number" and "remote node ID," are used to determine which adapter attached to the local machine should target which remote machine. These two parameters should be replaced by information obtained from the Dolphin software stack, but this information is not currently available from the SCI Library. There are other facilities in the Dolphin software stack that can provide such information. The second two parameters, "is initiator" and "use local QP number," are used to resolve message queue IDs. These two parameters should be replaced by information obtained from QP attributes, and are mainly used for running programs without external QP coordination.

Message queues

The Dolphin SCI Library is centered around message queues, much in the same way as RDMA is centered around QPs. An SCI message queue is a unidirectional channel used for passing messages from a host machine to a remote machine. From this point on, we will call the host machine the requester, and the remote machine the responder. We intend to translate the role of the host and remote machines in this explanation to the role of the host, and remote machines in send/receive transfer operation. The

requester can send messages, and the responder can receive messages. If the user wants bidirectional communication between two machines, each machine must serve as both requester and responder. In this case, we need to establish two message queues, one for the SQ and one for the RQ.

Both machines must actively seek communication to establish a message queue between a requester and a responder. Figure 4.6 shows an illustration of the main steps involved in establishing a message queue between a requester and a responder. The work consumer will create a message queue if it is associated with an RQ, or connect to a message queue if it is associated with an SQ. The requester and responder each need to know the node ID of the other machine, as well as the message queue ID of the remote message queue. Even though the responder creates the message queue, it needs to know the ID of the message queue being connected. For this reason, the RoPCIe verbs provider has to generate message queue IDs for each message queue and coordinate the message queue IDs between the two machines before the message queues can be created/connected.

We have decided to generate message queue IDs from local and remote QP numbers. A local QP number is unique to one QP in a PD, and users are required to supply a valid remote QP number when transitioning a QP to the RTS state. The RoPCIe verbs provider will establish two message queues per QP, so we need to generate two different message queue IDs from one QP number. We have solved this by doubling the QP number, which gives us twice the "address space" by incrementing the ID of one of the message queues. However, we can not always increment the ID of message queues associated with either SQs or RQs, as SQs need to target RQs, and RQs need to target SQs. The heterogeneous message queue connection requires that the two machines can have a heterogenous state. This is where the "is initiator" module parameter comes into play. Listing 4.21 shows how we create crossed message queue IDs for SQs and RQs based on whether the argument is true or false.

```
// Location: dis-kverbs/src/driver/dis_qp.c
...
/* SQ: Double QPN and increment if this node is not the initiator */
l_msq_id = is_initiator ? l_msq_id * 2 : (l_msq_id * 2) + 1;
r_msq_id = is_initiator ? r_msq_id * 2 : (r_msq_id * 2) + 1;
...
/* RQ: Double QPN and increment if this node is the initiator */
l_msq_id = is_initiator ? (l_msq_id * 2) + 1 : l_msq_id * 2;
r_msq_id = is_initiator ? (r_msq_id * 2) + 1 : r_msq_id * 2;
...
```

Listing 4.21: RoPCIe kernel verbs provider: Message queue ID calculation.

Vectored messages

The Dolphin SCI Library provides multiple different ways of sending messages. The RoPCIe provider is required to gather data from multiple buffers on the requester side, and scatter data into multiple buffers on the responder side, so we have chosen to use vectored messages. As shown

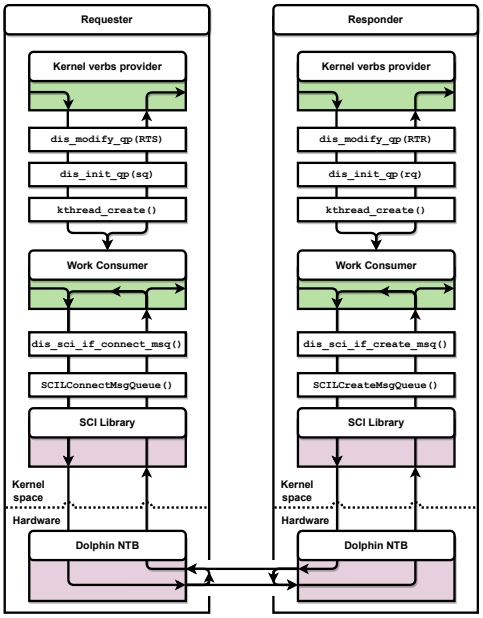

Figure 4.6: A requester connecting to a SCI message queue created by a responder.

in listing 4.8, a work consumer will build an IO Vector from the segments specified in the work request. This IO Vector will be passed directly to the Dolphin SCI Library, which will handle the send/receive. Figure 4.7 shows an illustration of the main steps involved in sending a message from a requester to a responder. Multiple flags can be passed along with the IO Vector, which can affect performance drastically. The optimal way to send this data through the message queues is a subject for future optimization. However, the Dolphin SCI Library API is sparsely documented, so further work on message sending/receiving should be done in closer collaboration with Dolphin ICS.

A message can be configured to use either DMA or PIO. The DMA mode will utilize the processor on the Dolphin NTB to transfer the message directly from the memory buffers specified in the message. This is similar to the standard mode of operation in InfiniBand transport and should have low CPU overhead and high throughput for large messages. The PIO mode will utilize the CPU on the host machine to power the message transfer, and DMA from the Dolphin NTB will not be used. This mode is similar to the inline mode of operation in InfiniBand transport and should have low latency for small messages.

4.4 RoPCIe user verbs provider

To use the RoPCIe transport from user-space, we have implemented a user verbs provider for the Libibverbs library, as described in section 2.4.1. Because this library is a part of rdma-core, we have chosen to integrate the user verbs provider with Libibverbs by using the rdma-core

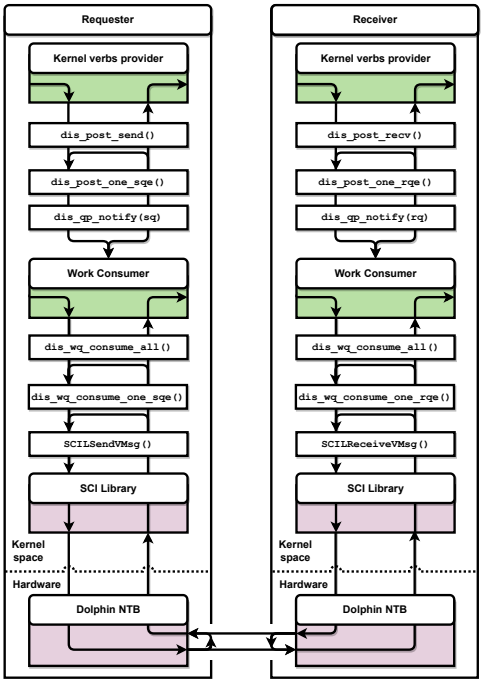

Figure 4.7: A requester sending a message to a responder.

build system. Building rdma-core can be done by forking the rdma-core GitHub repository, installing the required dependencies, and following the build instructions. When our fork of rdma-core is built, one of the binaries produced in the build is our version of Libibverbs that can utilize the RoPCIe kernel-space provider. By exporting this newly built library to the system library load path, it will be supplied to any application requesting the Libibverbs library at runtime. If one or more versions of the Libibverbs library is already installed on a system, the library with the highest version number will be chosen first, which is most likely our newly built library. Apart from being able to use any RoPCIe transport devices to do RDMA, the Libibverbs library we build behaves precisely like the upstream version.

While the build system in rdma-core is surprisingly fast and easy to deal with, we wanted to explore building the RoPCIe user-space provider by itself. The providers in rdma-core are built as shared object files, meaning that they are dynamically linked to Libibverbs. This linkage entails that it is technically possible to build a provider by itself, and integrating it with an already installed version of Libibverbs. We did not succeed in this endeavor because the development packages for rdma-core that were available for the version of CentOS we used did not provide most of the headers needed to compile a user-space provider by itself. After a failed attempt to manually find and provide all the correct headers, we determined that this task was a poor use of development time, mainly because it would not grant RoPCIe any tangible benefits.

4.4.1 Register provider

The process of registering a RoPCIe user verbs provider with Libibverbs is similar but less complicated than registering a RoPCIe kernel space driver with IB Core. A provider context data structure is dynamically linked to Libibverbs with a macro **PROVIDER_DRIVER**, as shown in listing 4.23. A data structure context of type **verbs_device_ops** contains information about the provider, and Libibverbs can get a list of provider supported verbs operations by calling the **alloc_context** function pointer.

```
// Location: dis-uverbs/providers/dis/dis.c
static const struct verbs_device_ops dis_device_ops = {
    .name                   = DIS_ROPCIE_NAME,
    .match_min_abi_version  = DIS_ABI_VERSION,
    .match_max_abi_version  = DIS_ABI_VERSION,
    .match_device           = dis_match_device,
    .alloc_device           = dis_alloc_device,
    .uninit_device          = dis_uninit_device,
    .alloc_context          = dis_alloc_context,
};
PROVIDER_DRIVER(dis, dis_device_ops);
```

Listing 4.22: RoPCIe user verbs provider: Provider structure registration.

To match the user verbs provider with a kernel verbs provider, Libibverbs will first look at the **match_table** in the provider context. The user verbs provider can place multiple entries in the matching table, and Libibverbs will pass the matching table to IB Core for hardware device matching. This way, a user verbs provider can target multiple hardware devices from the same vendor. The RoPCIe kernel verbs provider differs in that it is a virtual device, and cannot be matched against this matching table. Instead, the RoPCIe user verbs provider supplies a simple matching function. This matching function is shown in listing 4.23, and is very similar to the bus matching function in listing 4.1.

```
// Location: dis-uverbs/providers/dis/dis.c
bool dis_match_device(struct verbs_sysfs_dev *sysfs_dev)
{
    return !strncmp(sysfs_dev->ibdev_name, DIS_ROPCIE_NAME,
     strlen(DIS_ROPCIE_NAME));
}
```

Listing 4.23: RoPCIe user verbs provider: Device match function.

4.4.2 Verbs operations

Each verbs operation in the RoPCIe user verbs provider currently follows the same design pattern. Variables are declared, one or more commands are sent to the kernel verbs provider, and the result is checked and returned. All the verbs operations currently go through the slow path to the kernel, and a fast path to the Dolphin NTB is not yet implemented. While fast path posting of work requests and polling of completion queues would likely lower latency significantly for small messages, it is out of scope for this thesis.

Chapter 5

Evaluation and Discussion

5.1 Test environment

All of our tests were performed on the same two machines, with identical hardware and software configurations. The configuration of each test machine is listed in table 5.1. The software installed on each test machine is listed in table 5.2.

Component	Name
CPU	AMD Ryzen 7 3700X @ 3600MHz
Motherboard	ASUS TUF GAMING X570-PLUS
Integrated NIC	Realtek L8200A Gigabit Ethernet
RAM	16 GB @ 2133MHz
PCIe NTB	Dolphin PXH830
InfiniBand HCA	MT27800 Family ConnectX-5

Table 5.1: Test bench hardware configuration.

Component	Version
Linux distribution	CentOS 8
Linux kernel	5.5.8
Rdma-core	22.3
Perftest	4.2.2
Trace-cmd	2.7
Dolphin ExpressWare	5.5.8

Table 5.2: Test bench software configuration.

5.2 Test tools

5.2.1 dis-xtest

To aid in development, and to test the functional aspects of the RoPCIe transport, we have developed dis-ktest and dis-utest. As described in

section 4.1, these two programs exist in kernel-space and user-space respectively and provide more or less the same test of an RDMA transport. The test will exchange some predefined amount of data between two machines, and each side will check the integrity of the received data, byte for byte. We use this test to investigate the functional correctness of the RoPCIe transport in user-space and kernel-space, but it does not tell us anything about the performance of the transport.

5.2.2 Perftest

To do end to end benchmarks of the RoPCIe transport service, we have chosen a performance testing package called Perftest. This package is included in the Mellanox Open Fabrics Enterprise Distribution (OFED) and serves as the official performance testing facility for RDMA transports. It includes multiple test programs designed to measure either the latency or the bandwidth of an RDMA transport. Each test program can be configured to use any valid combination of RDMA transport operations and service types. The performance tests are built on top of Libibverbs, and can only test user-space RDMA. Perftest can be installed as an RPM package, or it can be built from source.

5.2.3 FTrace

To do micro-benchmarks of the RoPCIe transport, we have chosen a tracing utility called FTrace. This utility is built into the Linux kernel and can be used to debug and time processes in the kernel. Configured correctly, this utility can record a function trace with both timestamps and functions completion times, filtered on functions of our choice. The function trace recording induces virtually zero overhead, as tracing is built into every function in the kernel [21]. The overhead was the deciding factor when choosing among benchmark utilities and methods, as most of the functions we want to benchmark have completion times in the range of tens of microseconds to hundreds of nanoseconds.

The FTrace API can be reached through special files in the file system. While this approach lends itself well to custom scripting, it is also time-consuming to learn and use. We have instead opted to use a command line tool called trace-cmd, which acts as a front end to the FTrace file system API. Trace-cmd can be called with several options to start a recording in FTrace or print the result of a trace.

5.3 Benchmarks

5.3.1 Latency

We have used Perftest to measure and compare the latency of RoPCIe transport with PIO data transfer, RoPCIe transport with DMA data transfer, and InfiniBand transport with InfiniBand data transfer. Perftest measures latency with the "ping pong" method, where messages of the same size are

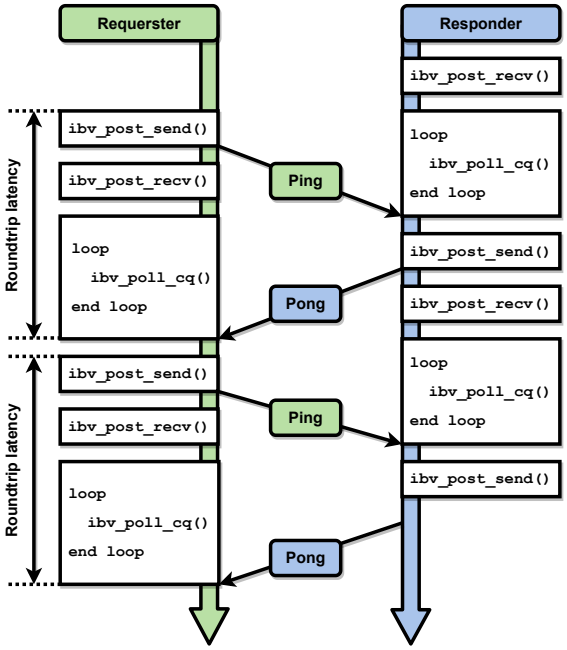

Figure 5.1: Two iterations of the ib_send_lat program measuring roundtrip latency between a requester and a responder.

sent back and forth between two nodes. The roundtrip latency is measured from the start of the ping message transmission procedure to the end of the pong message reception procedure. Perftest reports these results as one way latency by halving the roundtrip time. The main Libibverbs API calls made during two iterations of the latency test program are shown in figure 5.3. The user can specify the number of iterations, and we have chosen to use the default of 1000 iterations for our tests, as recommended by the Perftest documentation. Increasing the number of iterations to 10 000 did not change the result significantly.

The latency of each transport was measured using the ib_send_lat program in Perftest with the **–all** option. This option runs several tests with increasing message sizes, each test repeating for the defined number of iterations. The message size starts at 2^1 bytes, completes a test with 1000 iterations, and repeats with the next power of two until a message size of 2^{23} is reached. The latency of the transport for each message size is output to the user as a set of values computed from the halved roundtrip times collected during the test. The output values include the minimum, maximum, median, average, 99% percentile, and the 99.9% percentile one-way latencies, as well as the standard deviation of the set. We follow the Perftest documentation advice, which recommends using the median value as the latency of the transport, as it is the most resilient to variations in roundtrip times. At 1000 iterations, the median value should not be affected by any large initial values caused by warmup effects. We use the median value to plot graphs for latency unless otherwise is stated.

Figure 5.2 shows the results of benchmarking each transport, plotted

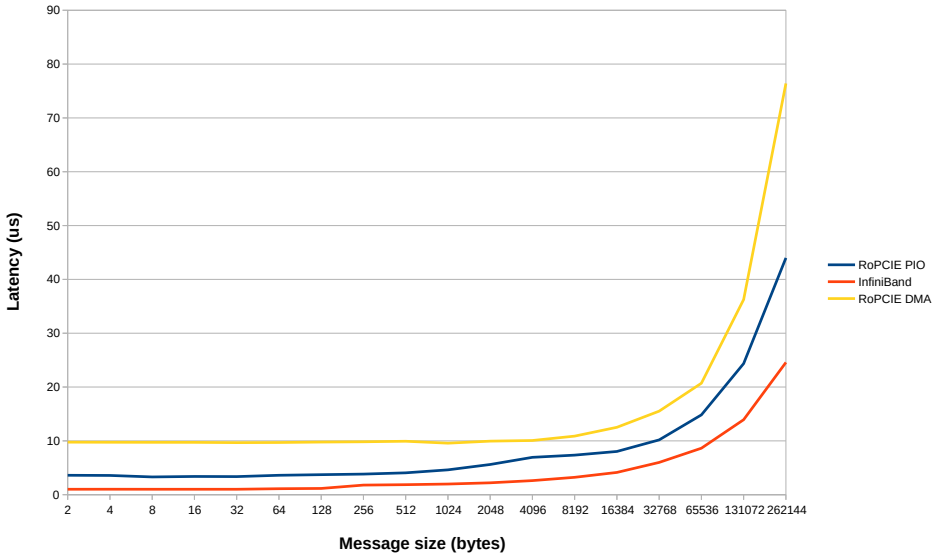

Figure 5.2: Comparison of median one way latency between RoPCIe with PIO, RoPCIe with DMA, and InfiniBand at different message sizes.

on a graph. RoPCIe with PIO appears to scale with message size similarly to InfiniBand while having around double the latency of InfiniBand at any given message size. RoPCIe with DMA has a higher latency for smaller messages but also appears to follow the same curve as the two other RDMA transports. The plot of RoPCIe stops at message size 262144 because the next message size surpasses some limit inside the Dolphin SCI Library. For the data points we were able to obtain, the three RDMA transports show the curve we expected, with a doubling in latency for a doubling in message size.

InfiniBand

InfiniBand has an inline transfer mode that can use the CPU to transfer small messages, similar to Dolphin's implementation of PIO. The results for InfiniBand in figure 5.2 were obtained with the inline mode threshold set to 500 bytes, which we expect most InfiniBand networks to do. With this threshold, InfiniBand was able to achieve a one-way latency of 1 μs for messages under the threshold. Running a test with the inline threshold set to 0 bytes, InfiniBand achieved a latency of 1.67 μs for the smallest messages.

RoPCIe with PIO

RoPCIe with PIO performs the data transfer with the CPU and does not bypass the kernel at all. This behavior means that the PIO mode will have significantly higher computational overhead compared to InfiniBand without inline and RoPCIe with DMA. The PIO mode provides the lowest latency of the two modes provided by the Dolphin SCI Library, for small

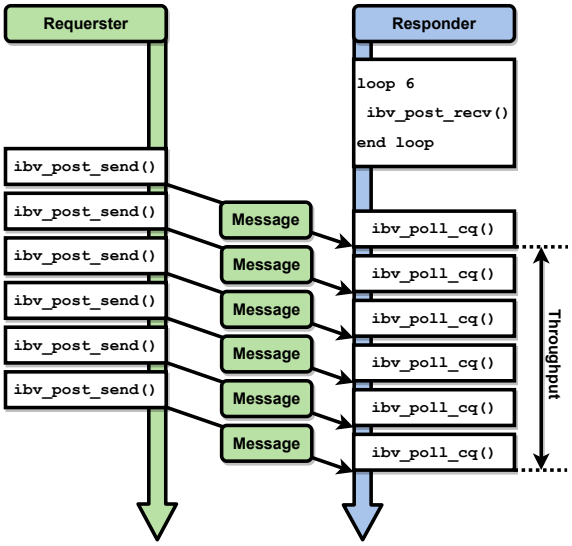

Figure 5.3: Six iterations of the ib_send_bw program measuring message throughput from a requester to a responder.

messages. In theory, the PIO mode should have higher latency for large messages than the DMA mode, because the throughput of PIO should be smaller than DMA.

RoPCIe with DMA

RoPCIe with DMA performs the data transfer with a processor on the Dolphin NTB, bypassing the kernel. We have learned that the DMA mode has a constant overhead when "spinning up" its DMA engine before a data transfer, leading to higher latency than PIO for small messages. For large messages, the DMA mode should outperform PIO, but we have not been able to test this due to an unfinished implementation in the Dolphin SCI Library.

5.3.2 Throughput

We have used Perftest to measure and compare the throughput of RoPCIe transport with PIO data transfer, RoPCIe transport with DMA data transfer, and InfiniBand transport with InfiniBand data transfer. Perftest measures throughput by sending a unidirectional stream of messages from one node to the other. The throughput is measured at the message receiver side, as the number of bits received per second over the duration of the message stream. The main Libibverbs API calls made during six iterations of the throughput test program are shown in figure 5.3. The user can specify the number of iterations, and here we have chosen to use 5000 iterations because this gave us a slightly more stable result.

The throughput of each transport was measured using the ib_send_bw program in Perftest with the **–all** option. This option runs several tests with increasing message sizes, each test repeating for the defined number

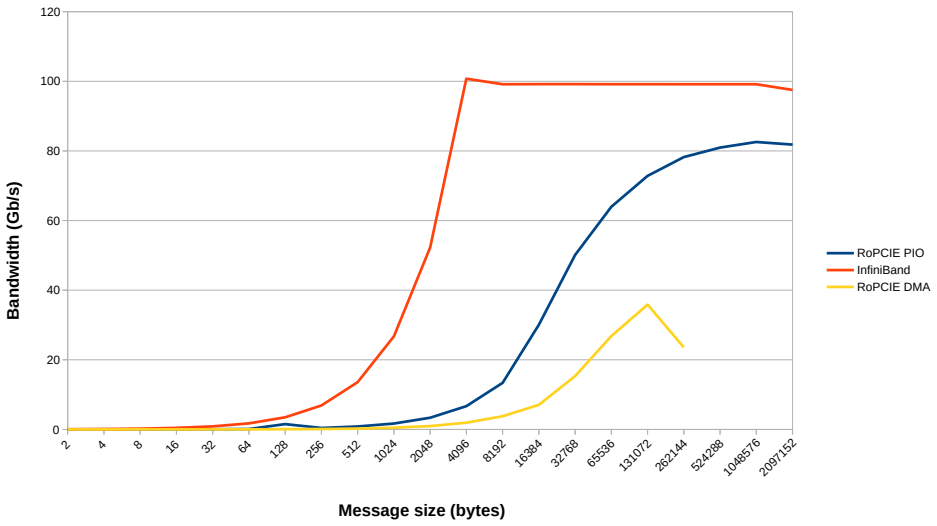

Figure 5.4: Comparison of average throughput between RoPCIe with PIO, RoPCIe with DMA, and InfiniBand at different message sizes.

of iterations. The message size starts at 2^1 bytes, completes a test with 5000 iterations, and repeats with the next power of two until a message size of 2^{23} is reached. The throughput of the transport for each message size is output to the user as a set of values computed from the time it took to transmit all the messages successfully. The output values include the throughput average and the message rate.

Figure 5.4 shows the results of benchmarking each transport, plotted on a graph. InfiniBand appears to plateau at 100 Gb/s for a message size of 4096 and beyond. RoPCIe with PIO experiences a more gradual increase in throughput and appears to flatten out at 82 Gb/s. RoPCIe with DMA has a very gradual increase in throughput, and after reaching 35 Gb/s, it plummets to 23 Gb/s for the last message size. The plummet in throughput is consistent across multiple tests, as are the results of the other transports.

InfiniBand

The InfiniBand transport was tested with a Maximum Transmission Unit (MTU) of 4096. Any messages larger than the MTU would be split into two or more packets. From the bandwidth benchmark, we can see that InfiniBand reaches its advertised throughput of 100 Gb/s at the 4096 byte message size, and plateaus at this throughput. Because each subsequent message size after 4096 is a multiple of 4096, InfiniBand experiences the best-case scenario where all packets are filled.

RoPCIe with PIO

The RoPCIe with PIO transport has a more gradual curve towards its peak. The profile of the curve is likely due to PIO not being packet-based, thus following an S curve. The latency benchmarks for RoPCIe with PIO explain

why the curve lags behind InfiniBand, it simply takes more time to transfer the same number of bytes.

RoPCIe with DMA

The RoPCIe with DMA transport follows approximately the same gradual curve as RoPCIe with PIO. This curve lags even further behind, again due to higher latency numbers. The dip at the end of the graph is sudden, and we have not been able to track down the root cause. We speculate that this is due to the unfinished implementation of vectored DMA messages in the Dolphin SCI Library.

5.4 Overhead

The overhead of a task is the cost of performing it. The overhead of an RDMA transport can be calculated as the combined cost of its transport characteristics. Transport characteristics include performance, computation, stability, reliability, security, monetary, and more. Every RDMA transport use case will weigh the cost of each transport characteristic differently, so it is almost pointless to claim one transports superiority over another. Instead, we will attempt to measure and analyze specific transport metrics with relevance to the work presented in this thesis.

In order to identify sources of overhead in the RoPCIe transport, we divide the transfer operation into transport segments from end to end. The transport segments are shown in figure 5.5. In this context, we consider everything after a user has made a call to the Libibverbs API as part of the transport. We do not consider the overhead incurred by the user program. The following describes each segment of the RoPCIe transport in detail, following the path of a message from a requester to a responder.

5.4.1 Requester side

In an RDMA transfer operation, a user that creates a send request and posts it to a QP is called a requester. A send request posted by a requester is processed to completion at the requester side of a transfer operation. We divide the requester side into two transport segments; "user to kernel" and "sender."

User to kernel segment

The first transport segment at the requester side is called the "user to kernel segment." This segment starts with a requester in user-space calling the **ibv_post_send()** function in the Libibverbs API, and ends when a call is made to the **dis_post_send()** function in the RoPCIe kernel verbs provider. The sole responsibility of the RoPCIe user verbs provider in this segment is to pass on the work request to Libibverbs command/response machinery. This work division means that the RDMA stack performs almost all work

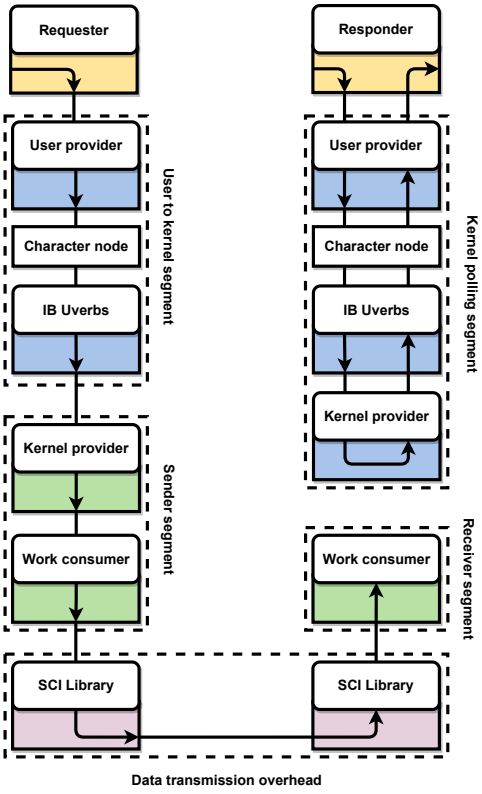

Figure 5.5: Transport overhead segments.

done in this overhead segment, and the overhead of the user verbs provider is close to zero.

We speculate that the primary overhead of this segment comes from the choice of path to the kernel by the RoPCIe user verbs provider. The Libibverbs command/response mechanism uses a character node managed by the IB Uverbs character driver. This mechanism is known as the slow path for RDMA transports and contributes a constant increase to the latency of sending a message. Most other hardware-based RDMA transports use a fast path that more directly communicates with the RDMA adapter(section 2.1.1). A fast path was not developed for the RoPCIe transport because it would require significant additions to the device driver of the Dolphin NTB, which we consider out of scope for this thesis.

Measuring the overhead of the slow path "post send request" operation is difficult. Here we attempt to measure the time difference from when the function **ibv_cmd_post_send()** is called in the RoPCIe user provider, to when the function returns. This benchmark includes the time taken to reach the RoPCie kernel provider from user-space, the time taken by the kernel provider to complete its function, and the time taken to return to user-space. We measure this roundtrip time to about 1 μs, which means we are going to need nanosecond precision to isolate the slow path overhead. Using a function **timespec_get()** to get time in nanoseconds, we observe the average time of 1000 roundtrips to be within a range of 900 to 1400

nanoseconds across multiple tests. We exclude any readings above 2000 nanoseconds, which occur at the start due to warmup effects. Then we use FTrace to measure the time spent in the RoPCIe kernel provider, which gives us a range of 600 to 800 nanoseconds. Subtracting the time taken by the kernel provider, we make a conservative estimate of the overhead incurred by the slow path to be 400 nanoseconds.

The severity of the overhead incurred by using the slow path to the kernel is dependent on the transport workload. Slow path overhead is constant in relation to both the message size of the work request and the number of work requests chained together. For workloads with large messages or a large number of small messages chained together, the latency of the slow path will be dwarfed by the latency of transferring all the messages. However, for workloads where many small messages are sent one by one, the latency and computational overhead of the slow path can become significant.

Sender segment

The second transport segment at the requester side is called the "sender segment." This segment starts with the **dis_post_send()** function being called in the RoPCIe kernel provider, and ends with the **SCISendVMsg()** function being called in the Dolphin SCI Library by a work consumer thread. We divide this segment into two: the kernel provider thread, and the work consumer thread.

Kernel provider: The task of the kernel provider in this segment is to create a send request, post it to a circular buffer, and wake up the work consumer. We have measured this task to take between 600 to 800 nanoseconds to complete, and want to break down what causes this overhead. FTrace can be used to trace one or more functions to trace and generate a function graph of the traced functions. We have traced the **dis_post_send()**, **dis_qp_notify()**, and **dis_qp_post_one_sqe()** functions, and the function graph from one such trace is shown in listing 5.1. The trace is made during a run of the ib_send_lat program in Perftest with a message size of 2. The function graph shows the total duration of **dis_post_send()**, as well as the duration of the two other functions called from it. From the results, we can observe that the duration of the **dis_qp_notify()** function is about 321 nanoseconds, which means that waking up the work consumer thread is the largest source of overhead in this operation, at small message sizes. The second function **dis_qp_post_one_sqe()** has a comparatively small duration, at about 110 nanoseconds.

```
ib_send_lat funcgraph_entry:          | dis_post_send() {
ib_send_lat funcgraph_entry: 0.110 us |   dis_qp_post_one_sqe();
ib_send_lat funcgraph_entry: 0.321 us |   dis_qp_notify();
ib_send_lat funcgraph_exit:  0.682 us | }
```

Listing 5.1: FTrace function graph of a call to dis_post_send() with message size 2.

To measure how the overhead scales with message size, we ran the same test with a message size of 2097152. This message size will force the **dis_qp_post_one_sqe()** function to allocate memory for the dynamic page map store, and iterate over a large number of pages in the MR. A function graph of this trace is shown in listing 5.2. Here we can observe that the **dis_qp_notify()** function duration stays more or less constant with message size, while the **dis_qp_post_one_sqe()** function duration has increased tenfold to 1.1 μs. In comparison, at this message size, the total one-way latency of the RoPCie transport is about 320 μs.

```
ib_send_lat funcgraph_entry:          | dis_post_send() {
ib_send_lat funcgraph_entry: 1.102 us |   dis_qp_post_one_sqe();
ib_send_lat funcgraph_entry: 0.370 us |   dis_qp_notify();
ib_send_lat funcgraph_exit:  1.784 us | }
```

Listing 5.2: FTrace function graph of a call to dis_post_send() with message size 2097152.

From the results of these function traces, we observe that the overhead incurred in the kernel provider thread has similar characteristics to the overhead incurred in the user to kernel transport segment. It is apparent that waking up the work consumer thread is the dominant source of overhead at smaller message sizes, and that the overhead incurred by the kernel provider thread increases at a very low rate with message size, relative to other parts of the transport. Therefore we conclude that the same workloads will be impacted by this overhead as described in the kernel to user transport segment.

Work consumer: The task of the work consumer in this segment is to wake up when new send requests are posted, process the send requests, post a work completion, and go back to sleep. In other words, this thread picks up where the kernel provider thread left off. To measure the wake-up overhead from the kernel provider thread to the work consumer thread, we can use FTrace timestamps, which have nanosecond resolution. By creating a new trace and calculating the time difference between the call to **dis_qp_notify()**, and time of the first call to **dis_wq_consume_one_sqe()**, we can get an estimate of the wake-up latency overhead. Listings 5.3 and 5.4 show the function graph of the new trace with message size 2 and 2097152 respectively. Because the timestamp is long, we only include the difference to the previous timestamp(in the format of (+nanoseconds)). From these timestamps differences we can observe that about 750 nanoseconds elapses between **dis_qp_notify()** and **dis_wq_consume_one_sqe()**, and the message size does not significantly impact this overhead. This result also tells us that there is a lag of about 400 nanoseconds from **dis_qp_notify()** returns to the thread starts doing work.

```
ib_send_lat [000] (+4819) funcgraph_entry: 0.331 us | dis_qp_notify();
       DIS [011] (+722)  funcgraph_entry:          | dis_wq_consume_one_sqe() {
       DIS [011] (+551)  funcgraph_entry: 0.381 us |   dis_wq_post_sqe_cqe();
       DIS [011] (+200)  funcgraph_exit:  0.751 us | }
```

Listing 5.3: FTrace function graph of the wake up of the work consumer, and the processing and completion of a work request with message size 2.

```
ib_send_lat (+424165) funcgraph_entry:     0.391 us | dis_qp_notify();
       DIS (+762)    funcgraph_entry:              | dis_wq_consume_one_sqe() {
       DIS (+201929) funcgraph_entry:     0.401 us |   dis_wq_post_sqe_cqe();
       DIS (+260)    funcgraph_exit:  ! 202.189 us | }
```

Listing 5.4: FFTrace function graph of the wake up of the work consumer, and the processing and completion of a work request with message size 2097152.

The new trace also gives us a measurement of the time elapsed between the call to the function **dis_wq_consume_one_sqe()**, and the call to the function **dis_wq_post_sqe_cqe**. This measurement gives us the latency from when the work consumer starts to process a work request, to when the work consumer starts to post a work completion to the CQ. For the small message size, the latency is about 550 nanoseconds, and for the large message size, the latency is about 202 μs. For both message sizes, the latency measured here is smaller than the end to end latency we measure with Perftest. The discrepancy is very apparent in the trace of the large message, where almost 120 μs of the end to end latency is unaccounted for. Because we now have traced and measured the entire requester side of the transport, we conclude that the remaining end to end overhead must occur in the data transmission segment or at the responder side of the transport.

5.4.2 Data transmission

Between the requester side and the responder side, there is a segment called the "data transmission segment." This segment starts with the **SCISendVMsg()** function being called by a work consumer thread at the requester side and ends with the **SCIReceiveVMsg()** function returning to a work consumer thread at the responder side. The responsibility of this segment is to transmit data from one or more buffers on the requester side and receive the transmission into one or more buffers on the responder side.

The RoPCIe kernel provider attempts to configure the SCI Library message queues and messages optimally, but the overhead in this transport segment is ultimately dictated by the Dolphin software stack and accompanying hardware. Because we cannot synchronize the time of the requester and responder machine to nanosecond precision, the only way of measuring the overhead of this transport segment is to do end to end tests. By subtracting the requester and responder side overhead from an end to end test, we can get an estimate of the transport overhead.

Ultimately, the optimization goal of RoPCIe should be to minimize requester and responder side overhead, while using the data transmission

segment as effectively and efficiently as possible.

5.4.3 Responder side

Like the requester side, we divide the responder side into two transport segments; "receiver" and "kernel polling."

Receiver segment

The first transport segment at the responder side is called the "receiver segment." This segment starts with the **SCIReceiveVMsg()** function returning to a work consumer thread, and ends with the **dis_wq_post_rqe_cqe** function being called by the same work consumer thread.

This transport segment starts with the assumption that a work consumer is already awake and processing a receive request. We can make this assumption because RDMA requires the responder side to post receive requests before the requester can post send requests. If this assumption is correct, the work consumer will spend some amount of time alive waiting for a message transfer, which would make the duration of the **dis_wq_consume_one_rqe** function longer than that of the message transfer. Listings 5.5 and 5.6 show the function graph of a work consumer processing one receive request with message size 2 and 2097152 respectively, and posting a work completion to a CQ. Both function graphs conform to our assumption, because the time spent in **dis_wq_consume_one_rqe** is about double the end to end latency. A work completion is posted inside this function, so we have to subtract the time this takes, but the assumption still holds.

```
DIS (+90)    funcgraph_entry:              | dis_wq_consume_one_rqe() {
DIS (+5811)  funcgraph_entry:      0.441 us |   dis_wq_post_rqe_cqe();
DIS (+251)   funcgraph_exit:       6.062 us | }
```

Listing 5.5: FFTrace function graph of the work consumer processing a receive request with message size 2, and posting a work completion.

```
DIS (+170)   funcgraph_entry:              | dis_wq_consume_one_rqe() {
DIS (+667272) funcgraph_entry:     0.381 us |   dis_wq_post_rqe_cqe();
DIS (+230)   funcgraph_exit:   ! 667.492 us | }
```

Listing 5.6: FFTrace function graph of the work consumer processing a receive request with message size 2097152, and posting a work completion.

We can also observe that posting a work completion to a CQ takes between 200 to 250 nanoseconds. The **dis_wq_consume_one_rqe** function has to take a spinlock, and then advance the CQ circular buffer. This overhead is another part of the total end to end overhead. We expect this overhead to stay constant in relation to message size, and this is supported by function graphs in figures 5.5 and 5.6.

Kernel polling segment

The second transport segment at the responder side is called the "kernel polling segment." This segment starts with a responder calling the **ibv_poll_cq()** function in the Libibverbs API and ends with the same function returning to the responder.

This transport segment is similar to the "kernel to user" segment. When a responder polls the CQ, the RoPCIe user provider creates and sends a command down the slow path to the kernel provider. When the RoPCIe kernel provider has polled the CQ, the function call chain returns to the requester. We can also measure the overhead of this segment in the same way we measured the user to kernel segment. The average time of 1000 calls to **ibv_poll_cq()** is calculated, only including calls where a work completion was returned. Running this test multiple times, we get latency averages in the range of 550 to 650 nanoseconds. We then use FTrace to measure the duration of the **dis_poll_cq** function, which comes out to around 90 nanoseconds. Subtracting the **dis_poll_cq** duration from the total poll duration, we again get a slow path latency of about 500 nanoseconds.

We have so far not analyzed the latency of going down the slow path versus returning up the slow path. Consider a scenario where the requester goes down the slow path and sends a message to a requester. The sender returning up the slow path will not affect the end to end latency of the message transmission. Now consider this message is received at the responder side, and a work completion is posted to the CQ, the moment before a responder polls the CQ. The responder can then return up the slow path with the work completion, and the trip down the slow path at the responder side will not affect the end to end latency. This is the best-case scenario, where, in total, one roundtrip of the slow path has been made between the requester and the responder.

Now we consider the worst-case scenario. A work completion is posted to the CQ the moment after the responder polls the CQ. The responder now has to return up the slow path without a work completion. Unsatisfied by the lack of work completions, the responder starts a new trip down the slow path, retrieves a work completion, and returns up the slow path. In the worst-case scenario, in total, two roundtrips of the slow path have been made between the requester and the responder. We conclude that between 500 and 1000 nanoseconds of the end to end latency of the transport can be attributed to the traversal of the slow path.

5.4.4 End to end

Figure 5.6 shows an overview of the latency numbers we have calculated for each RoPCIe transport segment. The numbers are generally conservative estimates of the largest sources of overhead, and adding the numbers up for each message size will yield a number slightly below the minimum end to end latencies we observe. We observe that most overhead incurred by the RoPCIe user and kernel providers is constant in relation to message

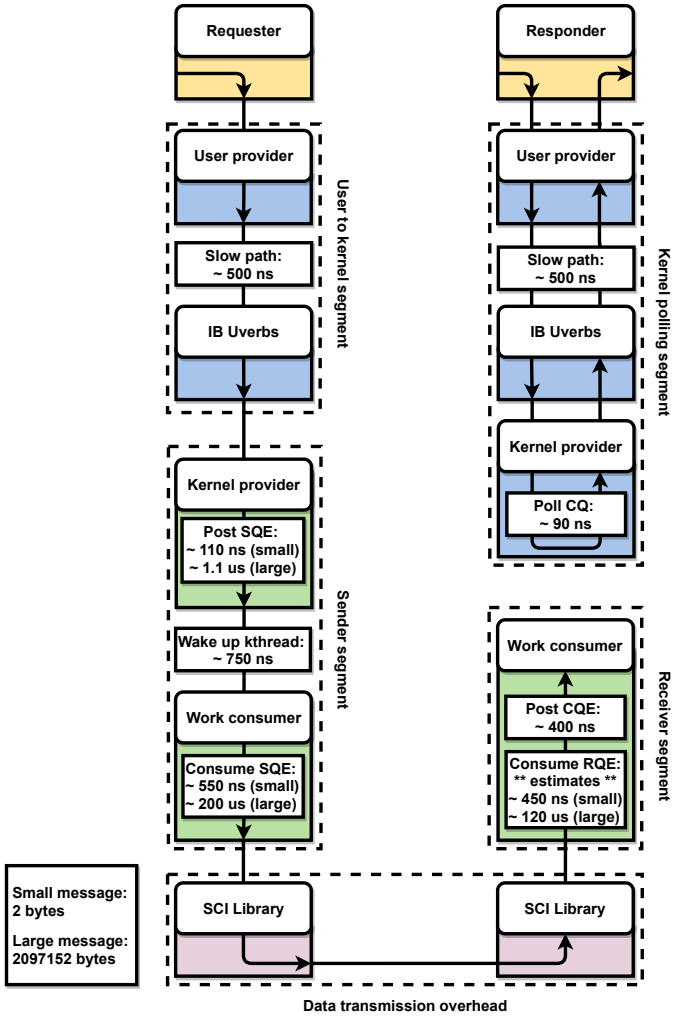

Figure 5.6: Transport overhead segments with approximate latency numbers.

size, and the overhead incurred by the Dolphin software and hardware grows with the message size.

Chapter 6

Conclusion

6.1 Summary

In this thesis, we have implemented a verbs compliant RDMA transport over PCIe. By integrating with existing RDMA systems and libraries, we have enabled RDMA applications in user-space and kernel space to communicate over PCIe interconnect. The chapters in this thesis outline the process of creating a new RDMA transport: RoPCIe.

We give a presentation of the RDMA technology in chapter 2. RDMA is described in the InfiniBand Architecture Specification, as an asynchronous data transmission technology revolving around a set of semantic behaviors called verbs. An implementation of RDMA is being developed as a native subsystem of the Linux kernel. The RDMA subsystem is extensive and has been designed to accept any device presenting a valid interface as an RDMA endpoint. While applications in kernel-space can interface with the RDMA subsystem directly, applications in user-space can reach the RDMA subsystem through a library called Libibverbs.

The design of RoPCIe is discussed in chapter 3. We present our design philosophy and outline some requirements for the RoPCIe transport. Two rejected transport architectures are described, as well as our reasons for choosing the third and final transport architecture. The chosen transport architecture emulates existing RDMA transports, with an interface at both kernel and user level.

A bottom-up description of the implementation of the RoPCIe transport is given in chapter 4. We start by implementing a bus, a device driver, and a device as three separate kernel modules. When these three kernel modules are loaded into the kernel, they create and initialize a virtual RDMA device for the RoPCIe transport. The RoPCIe virtual RDMA device mimics the functionality of a physical RDMA adapter inside the kernel and proceeds to register itself with the RDMA subsystem. We implement the mandatory verbs for the device, and a kernel worker to provide the asynchronous work consumption behavior of a physical RDMA adapter. A fourth kernel module is implemented to serve as a translation layer between the kernel worker and the API to the PCIe interconnect adapter. To make the RoPCIe transport available to user-space, we develop a library

plugin for Libibverbs.

The implementation of RoPCIe is tested and evaluated in chapter 5. We benchmark the latency and throughput of the RoPCIe transport and compare the results to benchmarks of the InfiniBand transport. Then we do a deeper examination of overhead in the RoPCIe transport, dividing the transport into sections to isolate the sources of overhead in an end to end data transfer. The transport is examined under conditions where both small and large messages are transmitted, and we note the effects of message size on each source of overhead.

6.1.1 Goals

The first goal we set for the RoPCIe transport was to create a design for its integration with RDMA infrastructure in Linux. We describe our research of the RDMA stack in chapter 2, and detail the design process in chapter 3.

The second goal we set for the RoPCIe transport was to implement it according to design. We describe our implementation of the design in chapter 4, including both a kernel and a user component.

The third goal we set for the RoPCIe transport was to test it and measure its performance. We describe our testing procedures in chapter 5. Here, the RoCPIe transport is tested alongside InfiniBand transport, and the performance of each transport is measured and compared at various message sizes. We verify that the RoPCIe transport delivers all messages intact with tests in user-space and kernel-space. Finally, we identify sources of overhead in the RoPCIe transport.

The fourth goal we set for the RoPCIe transport was to verify that RDMA software can use it, without modifications to the software or the transport. We verify that this works in chapter 5 by running programs in the Perftest package. The Perftest package is installed through the CentOS RPM package manager, and the RoPCIe transport is not modified to run Perftest programs.

6.2 Main contributions

In this thesis, we show that proprietary data transmission technology without support for verbs-based RDMA can be integrated with existing verbs-based RDMA libraries, and used by existing verbs-based RDMA software.

First, we design and implement a device driver capable of presenting a virtual RDMA device to the Linux kernel as a physical RDMA endpoint. The virtual RDMA device is first created by loading a set of kernel modules into the kernel, then registered with the RDMA subsystem by the device driver.

Next, we implement the verbs needed to operate an RDMA transport between the kernel of two machines. This implementation includes the verbs needed to create and use verbs resources like protection domains, memory regions, queue pairs, and completion queues. Asynchronous

work consumers translate work requests from the queue pairs to a message format used by the message queues in the Dolphin SCI Library. When a work request is fulfilled, the work consumer posts a work completion to the associated completion queue.

Finally, we make the RDMA transport available to applications in user-space by implementing a dynamically linked plugin for a user-space library called Libibverbs. The plugin defers all RDMA operation requests from user-space to the virtual RDMA device, through a command/response character node managed by IB Uverbs in the kernel. To enable DMA to user-space memory buffers, we create a pinned memory map between virtual and physical memory, as a part of the memory region implementation.

6.3 Future work

Several verbs remain to be implemented for the RoPCIe transport to comply with the InfiniBand Architecture Specification. Implementing both mandatory and optional verbs would enable the RoPCIe transport to provide RDMA for a greater number of RDMA applications.

The RoPCIe transport would benefit from being integrated into the Dolphin device driver. This integration could be a step on the path to let processors on the physical adapter consume work requests directly, eliminating the need for kthreads as asynchronous work consumers. In section 5.4.1, we find that the time used to wake up and schedule a work consumer kthread is about 750 nanoseconds or 21% of the end to end latency for small messages.

As we discuss in section 2.1.1, latency, and computational overhead could also be lowered by implementing a fast path for latency-sensitive RDMA operations. In section 5.4.1, we find that the slow path introduces about 500 nanoseconds per side of a transfer. Worst case, the slow path incurs 1 μs or 28.5% of the end to end latency for small messages.

The Dolphin SCI Library does not yet fully support DMA from multiple buffers without a copy to an intermediate buffer. Support for proper DMA should be implemented in the Dolphin SCI Library to achieve kernel bypass in RoPCIe, or another Dolphin API should be chosen.

The RoPCIe transport should be extended to support the remaining RDMA transfer operations. With transfer operations like RDMA Read and RDMA Write, a responder enables direct memory access from a requester. The implementation of these transfer operations should be made with strict memory security measures.

Bibliography

[1] Anuj Kalia, Michael Kaminsky and David G. Andersen. 'Design Guidelines for High Performance RDMA Systems'. In: *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*. USENIX ATC '16. Denver, CO, USA: USENIX Association, 2016, pp. 437–450. ISBN: 9781931971300.

[2] Chuanxiong Guo et al. 'RDMA over Commodity Ethernet at Scale'. In: *Proceedings of the 2016 ACM SIGCOMM Conference*. SIGCOMM '16. Florianopolis, Brazil: Association for Computing Machinery, 2016, pp. 202–215. ISBN: 9781450341936. DOI: 10.1145/2934872.2934908. URL: https://doi.org/10.1145/2934872.2934908.

[3] Wael Noureddine. 'NFS over 40Gbps iWARP RDMA'. SDC 2014. Sept. 2014. URL: https://www.snia.org/educational-library/nfs-over-40gbps-iwarp-rdma-2014 (visited on 09/05/2020).

[4] Lavanya Ramakrishnan et al. 'Evaluating Interconnect and Virtualization Performance for High Performance Computing'. In: *Proceedings of the Second International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computing Systems*. PMBS '11. Seattle, Washington, USA: Association for Computing Machinery, 2011, pp. 1–2. ISBN: 9781450311021. DOI: 10.1145/2088457.2088459. URL: https://doi.org/10.1145/2088457.2088459.

[5] Vernard Martin. 'Linux Clusters Institute:Intermediate Networking'. Georgia Institute of Technology. Aug. 2017. URL: http://linuxclustersinstitute.org/workshops/archive/aug2017/ (visited on 09/05/2020).

[6] Jerome Vienne et al. 'Performance Analysis and Evaluation of InfiniBand FDR and 40GigE RoCE on HPC and Cloud Computing Systems'. In: *Proceedings of the 2012 IEEE 20th Annual Symposium on High-Performance Interconnects*. HOTI '12. USA: IEEE Computer Society, 2012, pp. 48–55. ISBN: 9780769548319. DOI: 10.1109/HOTI.2012.19. URL: https://doi.org/10.1109/HOTI.2012.19.

[7] 'China Extends Lead in Number of TOP500 Supercomputers, US Holds on to Performance Advantage'. In: *TOP500 News* (Nov. 2019). URL: https://www.top500.org/news/china-extends-lead-in-number-of-top500-supercomputers-us-holds-on-to-performance-advantage/ (visited on 02/05/2019).

[8] Jilong Xue et al. 'Fast Distributed Deep Learning over RDMA'. In: *Proceedings of the Fourteenth EuroSys Conference 2019*. EuroSys '19. Dresden, Germany: Association for Computing Machinery, 2019. ISBN: 9781450362818. DOI: 10.1145/3302424.3303975. URL: https://doi.org/10.1145/3302424.3303975.

[9] D. E. Comer et al. 'Computing as a Discipline'. In: *Commun. ACM* 32.1 (Jan. 1989), pp. 9–23. ISSN: 0001-0782. DOI: 10.1145/63238.63239. URL: https://doi.org/10.1145/63238.63239.

[10] Chuanpeng Li, Chen Ding and Kai Shen. 'Quantifying the Cost of Context Switch'. In: *Proceedings of the 2007 Workshop on Experimental Computer Science*. ExpCS '07. San Diego, California: Association for Computing Machinery, 2007, 2–es. ISBN: 9781595937513. DOI: 10.1145/1281700.1281702. URL: https://doi.org/10.1145/1281700.1281702.

[11] Ken Darrow and Bruce Hedman. *Oportunities for Combined Heat and Power in Data Centers*. ICF International, Mar. 2009, p. 49. URL: https://www.energy.gov/sites/prod/files/2013/11/f4/chp_data_centers.pdf (visited on 02/05/2019).

[12] Jonathan Corbet. *The trouble with get_user_pages()*. Apr. 2018. URL: https://lwn.net/Articles/753027/ (visited on 30/08/2019).

[13] *InfiniBand Architecture Specification Volume 1*. 1.3. InfiniBand Trade Association. Mar. 2015. URL: https://cw.infinibandta.org/document/dl/7859.

[14] Ryan E. Grant et al. 'Scalable Connectionless RDMA over Unreliable Datagrams'. In: *Parallel Comput.* 48.C (Oct. 2015), pp. 15–39. ISSN: 0167-8191. DOI: 10.1016/j.parco.2015.03.009. URL: https://doi.org/10.1016/j.parco.2015.03.009.

[15] Bob Woodruff et al., eds. *Introduction to the InfiniBand Core Software*. Vol. 2. Proceedings of the Linux Symposium. Ottawa, Canada: Ottawa Linux Symposium, July 2005. URL: https://www.kernel.org/doc/ols/2005/ols2005v2-pages-279-290.pdf.

[16] Jason Gunthorpe. 'Challenges of the RDMA subsystem'. Linux Plumbers Conference. July 2019. URL: https://www.linuxplumbersconf.org/event/4/contributions/364/ (visited on 02/05/2020).

[17] *Linux kernel v5.5*. URL: https://github.com/torvalds/linux/releases/tag/v5.5 (visited on 02/05/2020).

[18] Jonathan Corbet and Alessandro Rubini. *Linux Device Drivers*. 2nd ed. Champaign, Illinois, United States: O'Reilly Media, June 2001. ISBN: 0596000081.

[19] Asim Kadav and Michael M. Swift. 'Understanding Modern Device Drivers'. In: *SIGARCH Comput. Archit. News* 40.1 (Mar. 2012), pp. 87–98. ISSN: 0163-5964. DOI: 10.1145/2189750.2150987. URL: https://doi.org/10.1145/2189750.2150987.

[20] David Howells and Paul E. McKenney. *Circular Buffers*. English. Version 5.5.8. Linux Kernel Organization. 1 p. URL: https://www.kernel.org/doc/Documentation/circular-buffers.txt (visited on 02/05/2019).

[21] Steven Rostedt. *Debugging the kernel using Ftrace - part 1*. Dec. 2009. URL: https://lwn.net/Articles/365835/ (visited on 24/04/2020).

Appendix A

Accessing the source code

The source code of the RoPCIe kernel verbs provider is available at https://github.com/AlveElde/dis-kverbs. This repository consists of four kernel modules and a Makefile-based build system to link, build, load, and unload the kernel modules. When the modules are built and loaded into the kernel, RoPCIe functionality will be added to the kernel by a virtual RDMA device.

The source code of the RoPCIe user verbs provider is available at https://github.com/AlveElde/dis-uverbs. This repository is a fork of the official rdma-core repository and can be built with the same procedure as the upstream repository. When built and exported to the system library load path, the library enables user-space access to the RoPCIe transport.

The source code of two RDMA test programs in kernel-space and user-space is available at https://github.com/AlveElde/dis-ktest and https://github.com/AlveElde/dis-utest. When executed, these two tests create RDMA resources and perform a simple ping-ping operation.

Appendix B

Additional tables

Bytes	Iterations	Median(us)	Average(us)	Std. dev.(us)
2	1000	9.76	19.43	81.00
4	1000	9.74	18.96	79.90
8	1000	9.73	19.70	82.91
16	1000	9.72	18.97	79.91
32	1000	9.67	18.89	79.55
64	1000	9.70	19.68	82.86
128	1000	9.78	19.01	79.86
256	1000	9.83	19.76	82.99
512	1000	9.93	19.26	80.10
1024	1000	9.57	18.86	79.92
2048	1000	9.95	19.83	83.00
4096	1000	10.07	19.31	80.00
8192	1000	10.88	20.18	80.01
16384	1000	12.52	22.52	83.01
32768	1000	15.52	22.71	63.07
65536	1000	20.71	28.06	57.15
131072	1000	36.25	46.54	82.95
262144	1000	76.40	92.66	140.17

Table B.1: RoPCIe DMA latency benchmarks.

Bytes	Iterations	Median(us)	Average(us)	Std. dev.(us)
2	1000	3.61	3.50	0.60
4	1000	3.58	3.57	0.50
8	1000	3.30	3.54	0.47
16	1000	3.39	3.63	0.46
32	1000	3.36	3.60	0.46
64	1000	3.61	3.70	0.46
128	1000	3.72	3.85	0.49
256	1000	3.83	3.94	0.52
512	1000	4.06	4.21	0.43
1024	1000	4.62	4.68	0.46
2048	1000	5.61	5.64	0.48
4096	1000	6.94	6.94	0.49
8192	1000	7.35	7.31	0.48
16384	1000	8.03	8.23	0.82
32768	1000	10.18	10.30	0.70
65536	1000	14.83	14.91	0.72
131072	1000	24.36	24.41	0.83
262144	1000	43.99	44.22	1.51
524288	1000	84.47	84.91	2.44
1048576	1000	163.87	164.88	4.88
2097152	1000	319.44	321.39	9.00

Table B.2: RoPCIe PIO latency benchmarks.

Bytes	Iterations	Median(us)	Average(us)	Std. dev.(us)
2	1000	1.01	1.01	0.03
4	1000	1.02	1.02	0.05
8	1000	1.01	1.01	0.03
16	1000	1.01	1.01	0.03
32	1000	1.01	1.01	0.04
64	1000	1.11	1.11	0.01
128	1000	1.16	1.16	0.03
256	1000	1.79	1.79	0.03
512	1000	1.86	1.86	0.02
1024	1000	1.98	1.98	0.05
2048	1000	2.21	2.21	0.04
4096	1000	2.62	2.62	0.04
8192	1000	3.23	3.20	0.12
16384	1000	4.14	4.20	0.14
32768	1000	6.00	6.07	0.12
65536	1000	8.64	8.68	0.10
131072	1000	13.93	13.96	0.08
262144	1000	24.57	24.60	0.09
524288	1000	45.76	45.78	0.09
1048576	1000	88.27	88.26	0.10
2097152	1000	173.39	173.41	0.08
4194304	1000	342.37	342.38	0.05
8388608	1000	682.93	682.93	0.07

Table B.3: InfiniBand latency benchmarks.

Bytes	Iterations	Median(us)	Average(us)	Std. dev.(us)
2	1000	18.75	18.95	1.21
4	1000	19.36	19.36	0.34
8	1000	19.33	19.35	0.31
16	1000	19.34	19.38	0.25
32	1000	19.32	149.25	1776.4
64	1000	20.28	184.36	1927.1
128	1000	34.79	628.95	4184.2
256	1000	27.11	27.55	2.07
512	1000	33.61	69.00	790.46
1024	1000	47.28	47.59	1.22
2048	1000	56.67	56.99	0.80
4096	1000	75.10	75.33	0.55
8192	1000	109.67	109.72	0.29
16384	1000	181.77	181.80	0.20
32768	1000	325.96	326.00	0.23
65536	1000	612.91	612.95	0.28
131072	1000	1190.70	1190.72	0.24
262144	1000	2349.32	2349.34	0.22
524288	1000	4650.98	4651.02	0.22
1048576	1000	9261.84	9261.90	0.31
2097152	1000	18489.97	18490.00	0.74
4194304	1000	37013.67	37013.96	1.33
8388608	1000	73874.24	74035.92	172.70

Table B.4: RXE latency benchmarks(Realtek L8200A Gigabit Ethernet).

Bytes	Iterations	Average(us)	Message rate
2	5000	0.000961	0.060077
4	5000	0.001912	0.059740
8	5000	0.003831	0.059859
16	5000	0.007596	0.059344
32	5000	0.015495	0.060527
64	5000	0.030791	0.060139
128	5000	0.061451	0.060011
256	5000	0.12	0.059190
512	5000	0.24	0.058112
1024	5000	0.49	0.059411
2048	5000	0.96	0.058680
4096	5000	1.93	0.058960
8192	5000	3.79	0.057896
16384	5000	7.06	0.053868
32768	5000	15.32	0.058432
65536	5000	26.80	0.051118
131072	5000	35.84	0.034177
262144	5000	23.58	0.011243

Table B.5: RoPCIe DMA bandwidth benchmarks.

Bytes	Iterations	Average(us)	Message rate
2	5000	0.023809	1.488064
4	5000	0.047039	1.469982
8	5000	0.013227	0.206667
16	5000	0.026439	0.206557
32	5000	0.052857	0.206471
64	5000	0.11	0.205653
128	5000	1.53	1.493630
256	5000	0.42	0.206914
512	5000	0.84	0.206279
1024	5000	1.69	0.206283
2048	5000	3.36	0.205296
4096	5000	6.68	0.203858
8192	5000	13.38	0.204128
16384	5000	30.05	0.229257
32768	5000	50.11	0.191153
65536	5000	63.96	0.121994
131072	5000	72.84	0.069470
262144	5000	78.23	0.037301
524288	5000	80.96	0.019303
1048576	5000	82.57	0.009843
2097152	5000	81.82	0.004877

Table B.6: RoPCIe PIO bandwidth benchmarks.

Bytes	Iterations	Average(us)	Message rate
2	5000	0.053817	3.363541
4	5000	0.11	3.381769
8	5000	0.22	3.381570
16	5000	0.44	3.401193
32	5000	0.87	3.397718
64	5000	1.74	3.394138
128	5000	3.48	3.395526
256	5000	6.88	3.360069
512	5000	13.58	3.314441
1024	5000	26.74	3.264509
2048	5000	52.26	3.189781
4096	5000	100.75	3.074498
8192	5000	99.16	1.513064
16384	5000	99.19	0.756777
32768	5000	99.19	0.378366
65536	5000	99.17	0.189152
131072	5000	99.17	0.094573
262144	5000	99.16	0.047283
524288	5000	99.16	0.023642
1048576	5000	99.16	0.011821
2097152	5000	97.52	0.005812
4194304	5000	99.16	0.002955
8388608	5000	99.16	0.001478

Table B.7: InfiniBand bandwidth benchmarks.

Bytes	Iterations	Average(us)	Message rate
2	5000	0.003661	0.228783
4	5000	0.007457	0.233035
8	5000	0.015005	0.234451
16	5000	0.030098	0.235144
32	5000	0.059924	0.234079
64	5000	0.12	0.231927
128	5000	0.23	0.225448
256	5000	0.45	0.219585
512	5000	0.78	0.190164
1024	5000	0.91	0.111027
2048	5000	0.91	0.055505
4096	5000	0.91	0.027745
8192	5000	0.91	0.013872
16384	5000	0.91	0.006929
32768	5000	0.91	0.003467
65536	5000	0.91	0.001733
131072	5000	0.91	0.000866
262144	5000	0.91	0.000433
524288	5000	0.91	0.000217
1048576	5000	0.91	0.000108
2097152	5000	0.91	0.000054
4194304	5000	0.91	0.000027
8388608	5000	0.91	0.000014

Table B.8: RXE bandwidth benchmarks(Realtek L8200A Gigabit Ethernet).