RoCE based 100Gbps RDMA network stack on FPGA hardware

by

Tianli Song

to obtain the degree of

Master of Science in Electrical Engineering

at the Delft University of Technology,
Faculty of Electrical Engineering, Mathematics and Computer Science,
to be defended publicly on Monday November 29, 2021 at 3:00 PM.

Student number:	5012155	
Project duration:	Nov 2020 – Oct 2021	
Thesis committee:	Dr.ir. Z. Al-Ars,	TU Delft, supervisor
	Dr.ir. M. Zuniga,	TU Delft
	Ir. T. Ahmad,	TU Delft

An electronic version of this thesis is available at `https://repository.tudelft.nl/`.

TU Delft

Acknowledgements

Although this thesis work is mostly done solely, it could not have been finished without the help of others. First I would like to thank my parents for being my physical and mental support. I would have not developed the quality of focus and persistence and have no chance to pursue knowledge for so many years without them. I would like to thank my girlfriend and schoolmates for sharing with me joy and love.

I would also like to thank Zhenhao He, Mario Ruiz, Tanveer Ahmad, Joost Hoozemans, Matthijs Brobbel, Monica Chiosa and other people who shared their knowledge and offered their help to me during this project.

My FPGA development work could not been accomplished without the existing work and support of powerful hardware platforms. I would thank David Sidler for being the giant so that I can see further. I would also thank the Xilinx Adaptive Compute Cluster (XACC) at ETH Zurich for providing me the access to development resources and networking infrastructures.

Finally, I would like to express my special thank to my supervisor Zaid Al-Ars for guiding me throughout this long journey, being encouraging when I was at the bottom and cheering for me when I reached hilltop.

Tianli Song
Delft, November 2021

Abstract

Big data analytics is an important enabler for booming technologies such as machine learning, genomics, and computer vision. These big data applications require large amounts of data transfers for distributed and parallel processing. Networking is thus a crucial facilitator and could make big impact on big data processing.

In a computing system with a common network stack such as the TCP/IP protocol suite, many expensive memory operations are necessary to process networking traffic. This means a large percentage of CPU resources are occupied by networking rather than data processing. The memory copying overhead introduced by networking not only reduces the throughput but also increases the latency. In this case, networking is becoming a major bottleneck for big data applications. This problem can be solved by applying Remote Direct Memory Access (RDMA) technology to the network stack. RDMA enables a zero-copy mechanism and has CPU bypass ability. With RDMA implemented, both the throughput and latency can be improved.

In this work, we developed an open source 100 Gbps RDMA network stack on Field Programmable Gate Array (FPGA) hardware. The developed stack follows the RDMA over Converged Ethernet (RoCE) architecture and targets the Alveo FPGA platform. The stack includes a User kernel that can be customized for user applications. This means that computing applications can also be offloaded to this RoCE stack. Finally, we evaluate the stack and compare it with existing TCP/IP and RDMA stacks like the EasyNet and StRoM. The results show that the developed RDMA stack achieves a throughput of 100 Gbps and an RDMA READ operation latency around 4 us and an RDMA WRITE latency around 3.5 us for 64B data. It shows a large throughput advantage over the TCP/IP stack for message sizes smaller than 1 MB. The latency is also slightly lower than the TCP/IP stack.

Contents

List of Figures ix

List of Tables xi

List of Acronyms xiii

1 Introduction 1
- 1.1 Context . . . 1
- 1.2 Problem Statement and Research Questions . . . 2
- 1.3 Thesis Contributions . . . 2
- 1.4 Thesis Outline . . . 2

2 Background 3
- 2.1 Network Protocol Stack . . . 3
  - 2.1.1 Network Stack Models . . . 3
  - 2.1.2 Common Protocols and Technologies . . . 4
- 2.2 Remote Direct Memory Access (RDMA) . . . 6
  - 2.2.1 Working Principles . . . 6
  - 2.2.2 RDMA Architectures . . . 8
  - 2.2.3 RDMA Transport Layer . . . 9
- 2.3 FPGA and HLS . . . 10
  - 2.3.1 Alveo FPGA . . . 10
  - 2.3.2 High Level Synthesis (HLS) . . . 12

3 Alternative Solutions 13
- 3.1 RDMA Transport Models . . . 13
  - 3.1.1 Reliability . . . 14
  - 3.1.2 Operation Support . . . 15
  - 3.1.3 Other Comparisons . . . 16
- 3.2 RDMA Architectures . . . 17
  - 3.2.1 Deployment Efforts . . . 17
  - 3.2.2 Header Comparison . . . 18
- 3.3 Summary and Development Paths . . . 22

4 Architecture and Implementation 25
- 4.1 Architectural Overview . . . 25
  - 4.1.1 Top Level Structure . . . 25
- 4.2 RoCE Kernel . . . 28
  - 4.2.1 Kernel Hierarchy . . . 29
  - 4.2.2 Kernel Interfaces . . . 29
  - 4.2.3 Operation Modes . . . 31
  - 4.2.4 The `rocev2` IP . . . 31
  - 4.2.5 TX/RX Path . . . 33
- 4.3 User and CMAC Kernel . . . 34
  - 4.3.1 User Kernel . . . 34
  - 4.3.2 CMAC Kernel . . . 35

5 Evaluation 37
- 5.1 Evaluation Platform . . . 37
- 5.2 RDMA Benchmark . . . 38
  - 5.2.1 Throughput . . . 39
  - 5.2.2 Latency . . . 40
  - 5.2.3 Resources . . . 42

5.3 Comparison to Related Work 43
6 Conclusion and Future Work **47**
6.1 Conclusion 47
6.2 Future Work. 48
A Appendix: Alveo Device Utilisation **51**
Bibliography **53**

List of Figures

2.1 The relationship between two of the most referred to network stack models. On the left side is the 7-layer OSI model. In the middle is the Internet Protocol suite model. On the right side, some example protocols are also listed on each layer. 3
2.2 Three-way handshake to establish TCP connections. 5
2.3 RDMA stack compared to TCP/IP stack. The RDMA stack is much simpler with much less overhead. 6
2.4 Data transfer path between two RDMA end nodes, where operating system kernels are bypassed. Taken from [19]. 7
2.5 The consumer queuing model. Taken from [3]. 8
2.6 The four RDMA architectures: InfiniBand, RoCE v1, RoCE v2 and iWARP. 9
2.7 The illustration of Connected service and Datagram service. Taken from [3]. 10
2.8 Floorplan of XCU280 FPGA Device. Taken from [43] 11

3.1 The top level network stack structures for different RDMA architectures with RC transport model selected. 17
3.2 A general comparison of the complete Ethernet packet for different RDMA architectures. 18
3.3 The packet structure of an RDMA WRITE packet for different RDMA architectures. . . . 20
3.4 The packet structure of an RDMA READ packet for different RDMA architectures. . . . 21
3.5 The packet structure of an RDMA READ response packet for different RDMA architectures. 21

4.1 The top level structure of the designed RoCE v2 network stack on Alveo FPGAs. 26
4.2 The top level structure of the RoCE v2 network stack on Alveo FPGAs with a loop back setup. 27
4.3 The block diagram and interfaces of designed RoCE kernel. 28
4.4 The interfaces for the RoCE kernel. 29
4.5 The internal structure for `rocev2` IP. Protocol header are processed in a pipelined fashion. Taken and modified from [29]. 32
4.6 The interfaces for the User kernel. 34

5.1 The networking configuration for XACC cluster at ETH Zurich. Taken from XACC documentation page. 37
5.2 The schematic diagrams of the board to board test setup. 38
5.3 The throughput and message rate of the RDMA READ benchmark. 39
5.4 The latency of the RDMA READ and WRITE benchmarks. 41
5.5 The comparison for the throughput and message rate to related work. 41
5.6 The delay distribution of each part in the entire data path for the RDMA READ operation. 42
5.7 The delay timeline for the RDMA READ and WRITE operation on 64B and 1KB data. . 42
5.8 The comparison for the throughput and message rate to related work. 43
5.9 The comparison for the latency to related work. 44

A.1 Alveo U280 device utilisation for the RoCE stack with an READ/WRITE User kernel . . 51
A.2 Alveo U250 device utilisation for the RoCE stack with a dummy User kernel. 52

List of Tables

3.1 Comparison between Reliable Connection, Unreliable Connection and Unreliable Datagram transport models in terms of reliability, supporting operations, scalability and supporting message size. 14
3.2 RDMA transport layer service alternatives selection table. Higher numbers mean better. 22
3.3 RDMA architecture alternatives selection table. Higher numbers mean better. 22

4.1 Host argument parameters for the RoCE kernel. 30
4.2 Description for the `debug` field of the designed RoCE kernel. 30
4.3 Description for the fields of the `tx_meta` AXI Stream for `rocev2` IP 32
4.4 Description for the fields of the `qp_interface` AXI Stream for `rocev2` IP 32
4.5 Description for the fields of the `conn_interface` AXI Stream for `rocev2` IP 33
4.6 Description for the `debug` field of the designed RoCE read or write User kernel. 35

5.1 The necessary parameters that used to establish RDMA RC model connection for our benchmark tests. 38
5.2 The resource utilization for our RoCE v2 network stack on Alveo U280. 43

List of Acronyms

ACK Acknowledgement
AETH Acknowledgement Extended Transport Header
API Application Programming Interface
ARP Address Resolution Protocol
ASIC Application Specific Integrated Circuit
AXI Advanced eXtensible Interface
BRAM Block Random Access Memory
BTH Base Transport Header
CDC Clock Domain Crossing
CEE Converged Enhanced Ethernet
CPU Central Processing Unit
CQ Completion Queue
CQE Completion Queue Element
CRC Cyclic Redundancy Check
DDP Direct Data Placement Protocol
DMA Direct Memory Access
EEC End-to-End Contexts
FCS Frame Check Sequence
FPGA Field Programmable Gate Arrays
GID Group Identifiers
GPU Graphics Processing Unit
GRH Global Routing Header
HBM High Bandwidth Memory
HCA Host Channel Adapter
HDL Hardware Description Language
HLS High Level Synthesis
HPC High-Performance Computing
IB InfiniBand
IBA InfiniBand Architecture
ICRC Invariant Cyclic Redundancy Check
ILA Integrated Logic Analyzer
IP Internet Protocol
IP Intellectual Property
LAN Local Area Networks

LRH Local Routing Header
LUT Look-Up Table
MAC Media Access Control
MPA Marker PDU Aligned Framing
MSN Message Sequence Number
MTU Maximum Transmission Unit
NAK Negative Acknowledgement
NIC Network Interface Card
OS Operating System
OSI Open Systems Interconnection
PFC Priority-based Flow Control
PHY Physical layer transceiver
PL Programmable Logic
PSN Packet Sequence Number
QP Queue Pair
QPN Queue Pair Number
QSFP Quad Small Form-factor Pluggable
RC Reliable Connection
RDMA Remote Direct Memory Access
RDMAP RDMA Protocol
RETH RDMA Extended Transport Header
RoCE RDMA over Converged Ethernet
RQ Receive Queue
RTL Register Transfer Level
RTT Round Trip Time
RX Receive
SFD Start Frame Delimiter
SLR Super Logic Region
SQ Send Queue
STag Steering Tag
TCP Transmission Control Protocol
TOE TCP Offload Engine
TX Transmit
UC Unreliable Connection
UD Unreliable Datagram
UDP User Datagram Protocol
VCRC Variant Cyclic Redundancy Check
WQ Working Queue

WQE Working Queue Element

WR Work Request

XACC Xilinx Adaptive Compute Clusters

XRC Extended Reliable Connection

1 Introduction

1.1. Context

With the rapid development of big data and cloud computing, there is a growing demand for high bandwidth and low latency networking. In a computing system with a typical network stack such as the TCP/IP protocol suite, a large amount of CPU time is spent on handling network traffic rather than actual data processing [18, 33]. Networking with such protocol stack is becoming one of the greatest bottlenecks for big data processing.

The evolving technology RDMA (Remote Direct Memory Access) has become popular among data centers [14–16] and research institutes [17] due to its low overhead and CPU bypass capability. In a common operating system, several memory copies are needed whenever application data needs to be transferred to the network. Application data is stored in the so-called user space while the network interface works with the kernel space. Memory copy occurs between the two memory spaces before data can be sent out to the network. For a networking-intensive application, the huge amount of network traffic requires a significant amount of buffer and CPU cycles. RDMA on the other hand, allows the network interface card directly access to user space memory. The operating system is not involved in networking. As the networking processing is usually offloaded to dedicated hardware like network adapter cards, the CPU is also bypassed entirely. This *zero-copy* characteristic of RDMA allows it to achieve higher throughput and lower latency.

Hardware accelerators such as FPGAs have gained attention in recent years for big data processing. They show good potential in accelerating computation throughput and improving energy efficiency. FPGAs are also very suitable for networking purposes because of their natural strength in data streaming. The inherent reconfigurability of FPGAs further consolidates its advantage. Many works have been done to implement network stack on FPGAs [7, 10, 11, 29, 31].

However, there is no open source RDMA stack implementation available off-the-shelf for FPGA hardware. There is also no open source platform that integrates data accelerators with RDMA networking for FPGAs.

In this work, we focus on the implementation of an open source RDMA network stack on FPGA hardware. The network should be implemented as a general purpose solution and allow customized accelerating kernels to enable in-network processing. With the growing bandwidth requirements in data centers, we are targeting 100 Gbps throughput. There are multiple network stack architectures for RDMA capability, which differs greatly in terms of hardware requirement and protocol layering. RDMA also defines several transport layer models for different application need. A thorough investigation of the difference between InfiniBand transport services and existing RDMA architectures should be carried out to have a comprehensive understanding of all the alternative implementations. The RDMA network stack design on FPGA should allow data to be transported in higher throughput and lower latency compared to common stacks like TCP/IP. This will be the focus of this thesis.

1.2. Problem Statement and Research Questions

Problem statement
The problem to be addressed in this thesis is stated as follows.

In a computing system with typical network communication protocols, data needs to go through the entire network stack of the host operation system before reaching the network. This network stack introduces a lot of protocol overhead such as that introduced by TCP/IP and memory copy overhead such as that between user space and kernel space. With such overhead, the data path cannot make full use of the bandwidth and has higher latency than needed.

Research questions
We formulate the research questions of our work as follows.

1. How can we enable RDMA network stack on FPGAs to allow for ultra-low latency communication?
2. How much throughput and latency can we achieve by implementing RDMA on FPGAs?
3. Can the RDMA stack achieve higher throughput and lower latency than the existing protocols (such as TCP/IP)?

1.3. Thesis Contributions

The contributions of the thesis can be summarized as follows:

1. Analysis of alternative RDMA implementations to identify the most suitable one for FPGA deployment.
2. Development of an open source 100 Gbps RoCE network stack on Alveo FPGA platform.
3. Measurements of throughput and latency characteristics of the implemented network stack on FPGAs.

Our 100 Gbps RoCE network stack for the Alveo platform is open source and publicly available. It can be found in the following Github repository `https://github.com/hcxxstl/Vitis_RoCE`.

Note that there are still problems with the memory interface of this design. It is only good for transmitting a small amount of data (in the order of megabytes).

1.4. Thesis Outline

The rest of the thesis is organized as follows.

Chapter 2: Background This chapter introduces the relevant background topics. We first explain network stack structure and some important protocols used in our work. Then, several crucial parts of Remote Direct Memory Access (RDMA) are introduced. And finally, the targeted hardware platform (FPGA) and its develop flow are also described within this chapter.

Chapter 3: Alternative Solutions In this chapter, we show the alternative solutions for several aspects: the transport model, RDMA architecture and development path. Detailed comparisons are also presented for each aspect in this chapter.

Chapter 4: Architecture and Implementation In this chapter we first introduce the overview of our implementation by showing the top level structure of our RDMA network stack. Then the details of each kernel such as the block diagrams and interface designs are presented.

Chapter 5: Evaluation In this chapter, evaluation results of the designed network stack are presented. The design is tested on Xilinx Adaptive Compute Clusters (XACC) ETH Zurich center. First, a simple read-write test is shown to validate the functionality. Then a benchmark is carried out. Both throughput and latency details are discussed. Finally, the results of our RDMA stack are compared to the existing TCP stack to show how much benefit we can achieve through our work.

Chapter 6: Conclusion and future work The conclusion together with some possible future work are discussed in the final chapter.

2

Background

2.1. Network Protocol Stack

Networking in a computing system can be extremely complex. To enable communication between systems, not only the physical medium but also high level application protocols should be specified. The layered structure of networking protocols forms a stack that brings significant complexity to the computing system and overhead to the data transmission process.

2.1.1. Network Stack Models

The network stack is usually modeled as a multi-layer structure. The layers are standardized for specific communication functions thus enabling interoperability for different protocols on the same layer. In Figure 2.1 we show two of the most referred to network stack models and the positions of some common protocols.

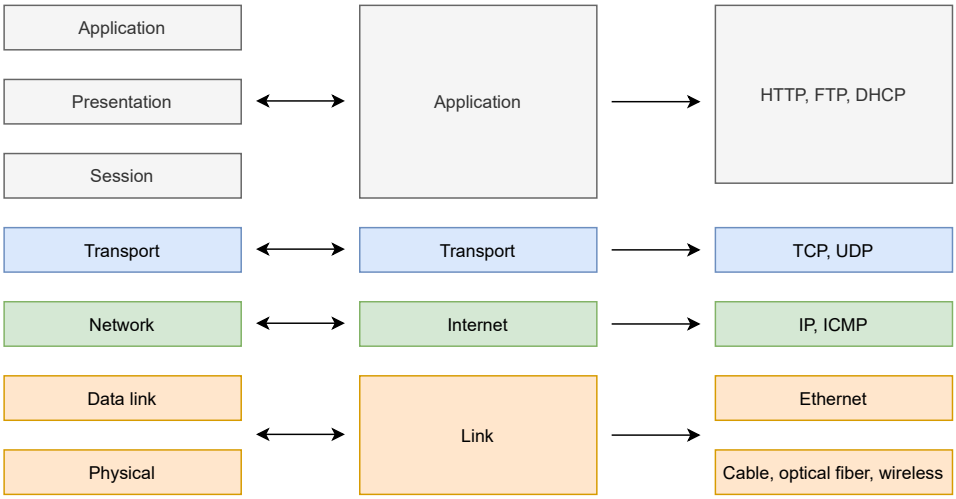


Figure 2.1: The relationship between two of the most referred to network stack models. On the left side is the 7-layer OSI model. In the middle is the Internet Protocol suite model. On the right side, some example protocols are also listed on each layer.

OSI model One well-known network stack model is the 7-layer Open Systems Interconnection (OSI) model. This model is developed during the 1980s aiming to comprehensively describe networking. It divides the network stack into 7 abstraction layers. But not all layers are necessary.

Internet Protocol suite model Another simpler model is the Internet Protocol suite which consists of 4 layers: link layer, internet layer, transport layer, and application layer. This is a more widely used model and is also the model that we will use in the following chapters.

A brief explanation about the scopes of the four layers is as follows:

- **Application layer** handles process-to-process communication. It has a direct interface to user applications.
- **Transport layer** handles the communication between hosts, providing functions like reliability and flow control.
- **Internet layer** handles the communication between independent networks. It is where the routing happens.
- **Link layer** handles the communication within a single network link. Sometimes it also includes the physical layer.

In Figure 2.1 we can see the relationship between these two models. In the Internet Protocol suite model, the link layer covers both data link and physical layers for the OSI model. All top three layers in the OSI model namely application, presentation, and session layers are abstracted as a single application layer in the Internet Protocol suite model. The transport and network layers remain the same, possibly because they are most clearly defined and easy to be separated from other layers.

On the right side of this figure, some example protocols are listed to give the reader a better view of the layer on which the well-known protocols stand.

Besides, the layers are highlighted in four colors. The same color scheme is applied for other figures in the rest of this thesis as well (e.g., the RDMA stack structures in Figure 2.6 and header structures in Figure 3.2). This way, layer information can be observed more clearly.

It is worth mentioning that such a networking model does not restrict the way of implementing each layer. Take TCP as an example, it can not only be implemented in software but also in hardware [28, 34] and even by home pigeon [36, 37] as long as it complies with the specifications.

However, there are some general patterns whether one layer is implemented on hardware or software. The link layer is usually deployed on the hardware. This is reasonable because the physical layer should always work on hardware. The application layer is often implemented in software because it needs a close connection to the users and needs flexibility. The network layer and transport layer are commonly hardware independent. Protocols like TCP is quite complex for hardware implementation but can be easily designed in software. But sometimes they can be offloaded to specific hardware such as the TCP Offload Engine (TOE) for TCP stack [13, 25, 28]. With TOE, the CPU stress can be released dramatically for networking-intensive tasks. In this work, we implemented the RDMA transport layer and network layer on hardware to offload the networking processing.

2.1.2. Common Protocols and Technologies

With the Internet Protocol suite model, we can easily understand the network stack structure. As mentioned in the previous section, protocols at different layers are for different functionalities. Here we list and briefly explain the protocols and technologies that are used in this thesis, in bottom-up order.

Ethernet

Ethernet is the most widely used networking technology for Local Area Networks (LAN) [30]. Ethernet is more of a standard than a protocol and works at the physical and link layer. This technology is developed to achieve both high speed and long link distance for data communication. Ethernet is evolving rapidly throughout its life. The most recent commercial products already support 400 Gigabit Ethernet using multi-mode fiber technology[1].

MAC

Media access control is a protocol that handles the packet transmission in a data link. It works at the link layer. The MAC protocol does the necessary job to encode or recognize packets for physical media. MAC uses an addressing mechanism to enable data exchange between two endpoints. It also does flow control and multiplexing so that packets will not interfere with each other.

[1] https://connectorsupplier.com/high-speed-transmission-update-200g400g

IP
IP is one of the most used network protocols on the planet. It is commonly deployed over Ethernet but is also able to function over other link layers. The main function of IP is to enable routability between networks. It is essentially the foundation of the Internet. There are two major versions of IP, namely IPv4 and IPv6. The IPv6 utilizes a longer IP address with 128 bits to solve the problem of exhausted IPv4 addresses. It is recommended and has been increasingly deployed globally.

TCP
Transmission Control Protocol, known as TCP, is one of the dominant transport layer protocols around the world. It often works with IP protocol. The main characteristic of TCP is reliability. When data is transferred through TCP, its completeness and correctness will be guaranteed because of the re-transmission and error checking functions. TCP can recover data that are damaged, lost, duplicated, or delivered out of order [23].

As a connection-oriented protocol, TCP has a connection establishment phase during which a three-way handshake procedure is carried out between client and server, as shown in Figure 2.2. The latency of this procedure is often used as an indicator for the performance of a TCP stack [10]. Both the server and client need the acknowledgements from the other side and both of them need to store the connection information (two pairs of IP addresses and TCP ports) for data transmission. After completing the data transmission, a similar four-way handshake is needed to end the connection. And this is called the connection termination phase.

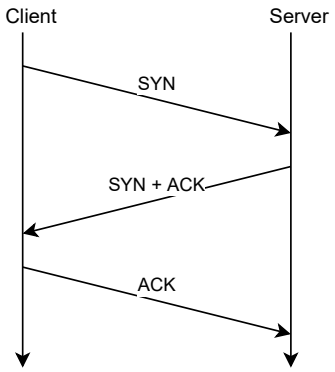


Figure 2.2: Three-way handshake to establish TCP connections.

To ensure reliability, several mechanisms are applied. Acknowledgements are necessary for every segment. If no acknowledgement is received by the sender within a preset period, the corresponding segment is considered to be lost. So the sender will re-transmit that segment. A sequence number is used to identify each packet counting each byte of data. The receiver can reconstruct the data in the correct order using this sequence order if the packets are delivered out of order. Besides, a sliding window mechanism is used to indicate how many bytes the receiver can acquire. The window size sets an upper limit for the data amount that the sender can send. This helps to avoid buffer overflow or packet dropping problems on the receiver side [35].

UDP
User Datagram Protocol is also a widely used transport layer protocol and is often considered as an alternative to TCP. UDP also works above Internet Protocol. Compared with TCP, UDP does not ensure reliability but offers simplicity and speed. No handshake and connection needed, UDP simply enables a procedure to send data with minimum overhead [21]. UDP considers the data correction and data integrity are not important. So UDP is usually applied in multimedia where delay matters while data loss is acceptable.

ARP
The Address Resolution Protocol is used between the internet layer and the link layer. It is a request-

response protocol that requests the corresponding MAC addresses for IP addresses. The ARP is an important support for IP protocol.

Both IP addresses and MAC addresses are needed for packets going through networks. However, on the application level, users usually do not want to know the lower level MAC addresses. ARP does the job of finding out the target MAC address based on its IP address. Whenever a network card needs to form a complete output packet, an ARP request containing an IP address will be broadcast to the network. If the machine with this target IP gets this ARP request, it will generate a response to show its MAC address. After the response is received by the requester, the ARP handler will save a mapping between the MAC address and IP address. After that, the requester knows the MAC address so that it can insert the MAC address into the output packets.

As mentioned previously, protocols on different layers can be switched and still work properly. If we exchange layers between these two network stacks: one is TCP/IP over Ethernet and the other is pure InfiniBand. There is RDMA over Converged Ethernet (RoCE) which puts InfiniBand transport layer over IP and Ethernet. There is also IP over InfiniBand (IPoIB) which uses TCP/IP layers over InfiniBand fabric.

2.2. Remote Direct Memory Access (RDMA)

With the rapid growth of big data, networking traffic has been increasing exponentially. Networking is becoming a bottleneck in data centers. Typical networking technology requires the operating system to process network traffic which occupies CPU time. The huge amount of traffic will eat up significant CPU resources. RDMA technology is designed to avoid this problem. Direct Memory Access (DMA) allows certain hardware to access (read/write) host memory directly without any CPU processing. As a further development of DMA, RDMA is a concept where the memory of a remote host can be accessed by a local system without interrupting the remote CPU. By bypassing the operating system, the CPU can be freed from expensive memory operations for important jobs.

However, RDMA does not natively work on the widely deployed network fabric - Ethernet. Native RDMA is based on InfiniBand, which means the entire hardware infrastructure should be replaced to enable RDMA on Ethernet-based networking. Luckily there is a technology called RoCE (pronounced ”rokie”) that wraps the InfiniBand packets within Ethernet headers. Until recently, RoCE had achieved good performance over lossy Ethernet infrastructures [26] and gained popularity. RDMA is now more and more popular among data center networks.

2.2.1. Working Principles

One of the most important characteristics of RDMA is zero-copy. This means applications can transform data directly, without the need for data copying between network layers or memory spaces.

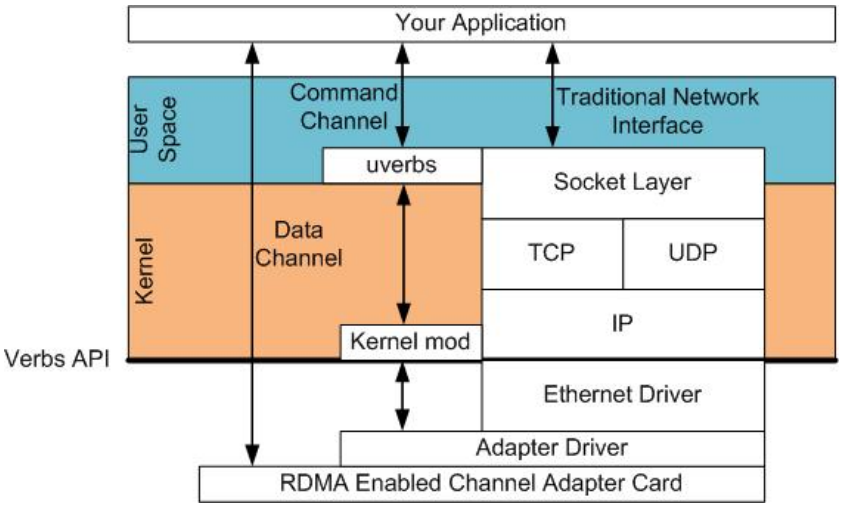


Figure 2.3: RDMA stack compared to TCP/IP stack. The RDMA stack is much simpler with much less overhead. Taken from[2].

In an Operating System (OS) memory is divided into two categories, namely the **kernel space** and the **user space**, as shown in Figure 2.3. Kernel space is reserved for lower level functions and is where the operating system kernel programs store and execute. While user applications can only access user space memory. Whenever an application needs to access the memory in kernel space, data should be accessed through an interface called the system call. When data is accessed by an application, memory copy occurs between the memory spaces.

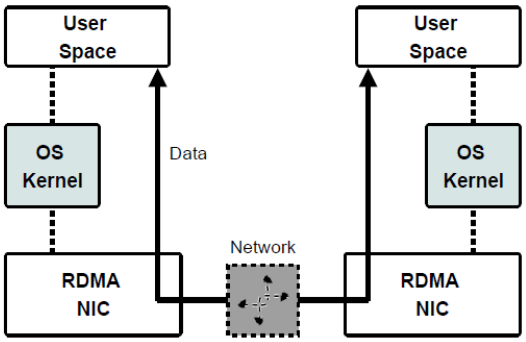


Figure 2.4: Data transfer path between two RDMA end nodes, where operating system kernels are bypassed. Taken from [19].

Networking subsystem such as TCP/IP stack is typically compiled within the kernel in an OS [1]. Layers including the transport layer and below are implemented inside the kernel. Only the application layer is open for users. The interface that allows user applications to access networking is called **socket**. When a user wants to transfer application data to a remote machine through the network, data first need to be copied from user space to kernel space then go to the network interface card. At the remote end, there will also be at least one copy from kernel space to user space so that the data can be acquired by the user application. More copies can occur in the network stack such as buffering and backups for possible re-transmission.

However, with RDMA there is zero copy needed. Figure 2.3 shows the difference between the RDMA stack and the TCP/IP stack structure. RDMA allows applications to read and write data from and to user memory directly without going through the kernel space. No memory copy will happen between kernel space and user space. This zero-copy mechanism avoids the use of CPUs and caches, thus saving CPU cycles for non-networking tasks and reducing the latency of the network stack. The data transfer path between two RDMA end nodes is illustrated in Figure 2.4, where operating system kernels are bypassed entirely.

From the perspective of a user, the required steps to perform RDMA communication are summarized as follows.

1. Each consumer (user) should first register a memory region for remote access.
2. Queue pairs should be established by each consumer as the communication port. It is also the interface exposed for the user from the network adapter.
3. The queue pair numbers, memory addresses, protection keys, and other necessary information should be exchanged between the local and remote consumers for RDMA communication. They are used for RDMA requests and connection establishment.
4. Consumer can then queue up RDMA requests through its queue pairs.
5. The RDMA requests will be processed and executed by the network adapter, usually the host channel adapter (HCA). After the communication is complete, the user will be notified.

This brief instruction is enough for the reader to understand how RDMA communication works on the user's side. But we are definitely more interested in the side of the network adapter. The new terms mentioned above will also be explained shortly.

[2] https://zcopy.wordpress.com/2010/10/08/quick-concepts-part-1-%e2%80%93-introduction-to-rdma/

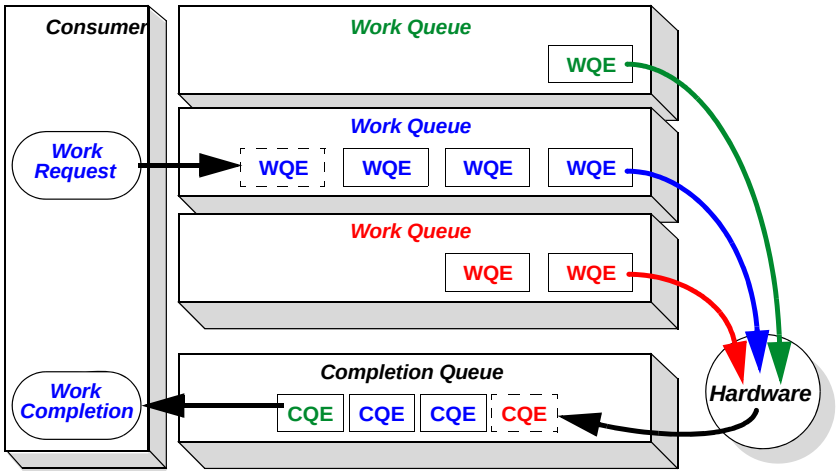


Figure 2.5: The consumer queuing model. Taken from [3].

The IBA (InfiniBand Architecture) network adapter (usually but not necessarily implemented in hardware) that handles transport level functions is called a Channel Adapter. In most cases, it is a host channel adapter (HCA). It is basically a DMA engine that allows reconfiguration. The interface for an application to access RDMA networking is a set of functions called **IBA verbs**. The queues are key components that bridge the user and HCA. There are three types of queues, Send Queue (SQ), Receive Queue (RQ), and Completion Queue (CQ). The send queue and receive queue are called **Queue Pair (QP)** as they are often created at the same time. They are also called Working Queues (WQ) since they are responsible for handling requests, as shown in Figure 2.5. Whenever the user issues an RDMA request, it is in the form of a Work Request (WR). The WR will be interpreted as the Working Queue Element (WQE, pronounced ”wookie”), which is placed in the WQ either SQ or RQ. HCA will then execute these WQEs in the order they are placed in WQ. The RDMA is fundamentally designed in an asynchronous mechanism, which means the user can issue many requests at a time. These requests will become the WQEs queuing up in a WQ. After each WQE is executed, a Completion Queue Element (CQE, pronounced ”cookie”) will be placed in the CQ by HCA. The user is then informed about the request completion by the HCA.

Besides the queuing model of IBA, RDMA communication is inseparable from its operations. Two types of operations are supported, which are called the **channel semantic** and the **memory semantic**. The channel semantic includes the `send` operation. It is a normal type of communication where the sender provides the data to be sent and the receiver determines where to place the data by itself. This means that the channel semantic requires both ends involved to operate. So we consider this is not an RDMA operation. On the other hand, the memory semantic is single-ended. It includes the `RDMA READ`, `RDMA WRITE` and `RDMA Atom` operations. With memory semantic, the local side is able to specify the remote memory address for direct memory reading or writing. These operations do not need any remote side involvement in data transfer. Although the memory semantic looks superior to channel semantic, they should be used cooperatively in real applications.

Due to the different semantics, WQEs used for these operations also differ. For a send operation, the WQE is simply a pointer to the local memory where the data should be taken from. While for RDMA operations, WQE includes not only local memory address but also remote memory address so as to enable single-ended communication. The `RDMA Atom` operations also work differently, but will not be covered in the thesis.

2.2.2. RDMA Architectures

As discussed in the previous section, a network stack can have different protocol combinations for the multi-layer structure. There are also multiple network stack architectures supporting RDMA. The protocol layering for them are shown in Figure 2.6. On the right side of the figure, we can also see whether each layer is usually implemented in software or hardware. is The application layer of each architecture remains the same while the other layers are stacked differently.

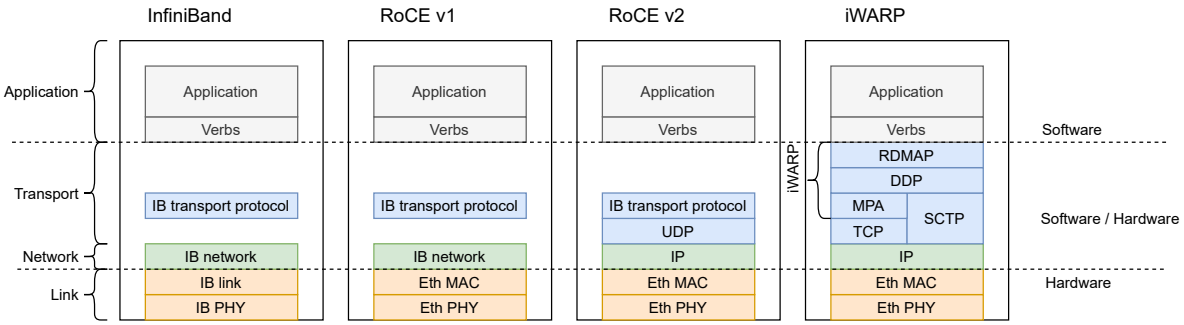


Figure 2.6: The four RDMA architectures: InfiniBand, RoCE v1, RoCE v2 and iWARP.

InfiniBand
Infiniband is considered as a competitor to Ethernet. Infiniband is not a protocol that can be fit into one of the layers. It's a complete specification that defines an entire network stack from physical layer all the way to transport and even application layer. RDMA is inherently supported over InfiniBand Architecture.

RoCE
RDMA over Converged Ethernet (RoCE) is a technology that combines the RDMA and widely used Ethernet fabric. It has two versions that differ in architecture. RoCE v1 is proposed to enable InfiniBand transport and network layers to work over Ethernet. The Ethernet link layer and physical infrastructures are kept. RoCE v2 only uses the InfiniBand transport layer while utilizing UDP/IP as the network layer. RoCE v1 is only able to work within the local area network while RoCE v2 includes the IP layer so that it is possible to route around the network.

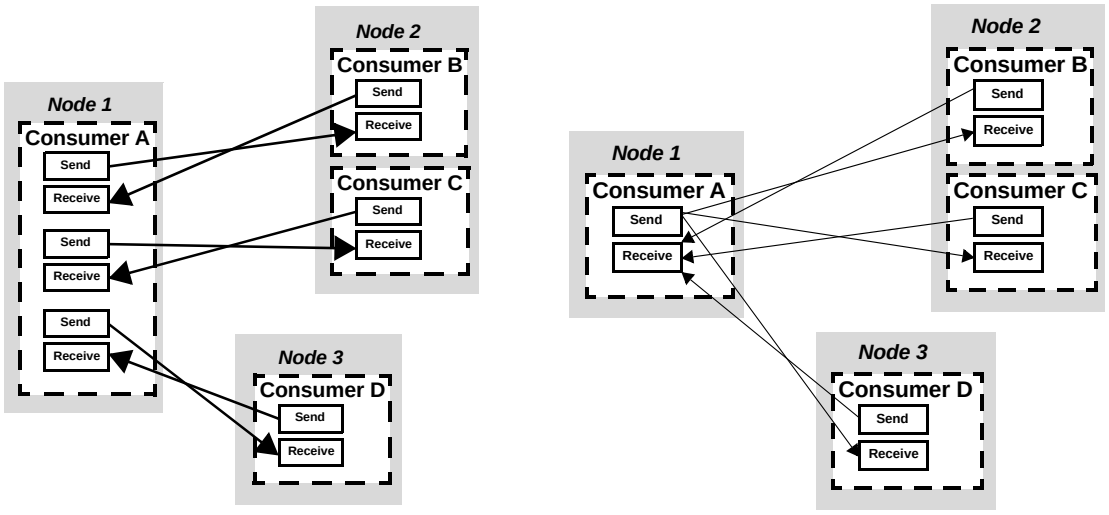
iWARP
To enable RDMA and keep even more parts of the typical TCP/IP protocol suite, there is iWARP. It implements InfiniBand RDMA Protocol (RDMAP) over Direct Data Placement Protocol (DDP) over TCP/IP stack [24]. With the entire TCP/IP stack kept, it is the architecture that could have no extra hardware needed but only implementation in software. However, the complexity is the highest among these three architectures as it has the most layers.

In this section, only the general concept and protocol structure is presented. More details of each of the three RDMA architectures will be discussed in Section 3.2.

2.2.3. RDMA Transport Layer
As shown in the Figure 2.6, every architecture needs the InfiniBand transport protocol for RDMA. On the transport layer, there are multiple possible working models. They are specialized for different purposes such as reliable data transmission or broadcasting. In this section, the division of such transport models will be discussed. The necessary knowledge for the later alternative evaluation is provided.

Connected Service
As indicated by the name, the connection service on the transport layer is connection-oriented. If the implementation follows a connection service, any communication requires a connection established, which means a pair of QPs should be created. If there are many processes on multiple devices that want to communicate with each other, QPs should be created between every two endpoints as illustrated in Figure 2.7a. The connected service includes the Reliable Connection model, Unreliable Connection model, and Extended Reliable Connection model.

Datagram Service
Unlike the connection service, the datagram service is connectionless. There is no need to establish that many QPs as in connection services if the datagram service is applied. Any QP is allowed to communicate with any other QP on any node, as shown in Figure 2.7b. This means only one QP is needed for one endpoint, which greatly reduces the complexity of heavily interconnected systems.



(a) The illustration of connected service.

(b) The illustration of Unreliable Datagram service.

Figure 2.7: The illustration of Connected service and Datagram service. Taken from [3].

The datagram service includes Unreliable Datagram and Reliable Datagram. The Reliable Datagram service requires extra End-to-end contexts (EEC) to multiplex the communication links. The EEC is also responsible for reliability mechanisms.

2.3. FPGA and HLS

In this section, necessary background knowledge about Field Programmable Gate Arrays (FPGAs) will be discussed. First, we show the Xilinx Alveo FPGA platform and its working flow. Then we will briefly introduce High Level Synthesis (HLS).

Networking processing is highly focused on data streaming. The entire stack of networking protocols is like a pipeline where data is going through. A huge amount of data should be processed as the data traffic needs to be encoded to or decoded from formatted packets that have certain headers. The data processing of a network stack cannot be entirely parallelized for a single data stream.

There are mainly four types of hardware for computing, CPU, GPU, FPGA, and Application Specific Integrated Circuit (ASIC). While CPUs and GPUs are good at general-purpose computation, FPGAs and ASICs are more task-specific. Since there are no complex computations and parallelism for networking CPUs and GPUs are not favoured candidates for offloading networking. FPGAs and ASICs both run algorithms on circuits, which brings ultra-low latency and good energy efficiency. A lot of effort has been done to offload the networking from CPUs to networking specific hardware based on ASICs [8, 12] or FPGAs [13, 25]. Compared to FPGAs, ASICs have less hardware overhead thus providing better performance. They are commonly used for commercial products. However, it is less flexible since there is no way to reprogram or change the functions once the chips are fabricated. Plus the high cost throughout the ASIC design cycle, FPGAs show a strong potential for this networking offloading task because of their relatively low design cost and flexibility. When developing FPGAs, it is very easy to modify the design or add new features. FPGAs are also the choice for many research purpose projects. This provides not only valuable documents but also design platforms such as Xilinx Adaptive Compute Clusters (XACC) for us.

2.3.1. Alveo FPGA

In this project, we use the Xilinx Alveo U280 boards on XACC as the hardware. With XCU280 FPGA inside, Alveo U280 is one of the best high-end FPGAs of the latest generation. U280 has not only huge hardware resources such as LUTs and registers but also powerful peripheral components. The U280 floorplan is shown in Figure 2.8. The board offers 2 Quad Small Form-factor Pluggable (QSFP) ports for networking. They are QSFP28 ports both support 100Gbps Ethernet. In terms of memory, it has 2 blocks of High Bandwidth Memory (HBM) sum up to 8 GB capacity with a 460 GB/s total bandwidth. It also provides 2 PCIe 4.0x8 interfaces connecting to host processors [43]. These high bandwidth

memory connections are necessary for an RDMA network stack implementation.

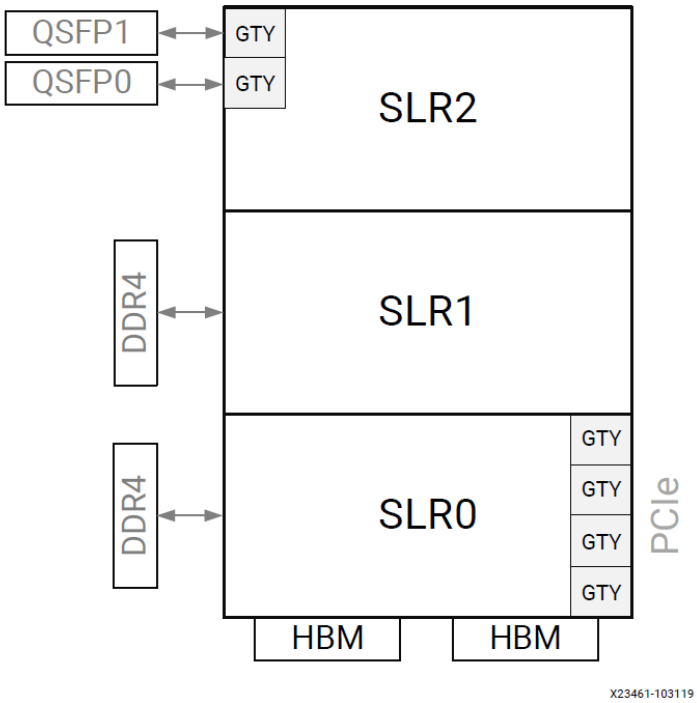


Figure 2.8: Floorplan of XCU280 FPGA Device. Taken from [43]

As we can see, the XCU280 FPGA hardware resources are divided into 3 blocks called Super Logic Regions (SLRs). Resources within the same SLR have shorter paths connecting to each other compared to those located at different SLRs. Thus, it is important to bound design kernels within the same or the nearby SLR for a shorter delay. So that the design can achieve better timing results for a higher frequency. In our RDMA case, the QSFP port is used as the interface to the networking fabric. So the kernel handling the lowest network layer should be placed at SLR2, close to the network ports. Similarly, kernel connecting to host or HBMs should be at or close to SLR0. As a result, it is inevitable for our design to spread over all three SLRs. This could lead to problems for timing closure and should be paid attention to.

Besides the internal resources of Alveo FPGAs, it is also important to understand its programming model and design flow. The Alveo platform usually uses the Xilinx Vitis as its development toolkit. Compute tasks are written to the FPGA programmable logic (PL) in the form of kernels. The kernels should be triggered and controlled by the host executable from the host processor. There can be multiple kernels and also multiple instances of a same kernel on one FPGA. The kernels can be developed either using HDL or through HLS using C/C++. The OpenCL framework can be used as the API for HLS kernels.

Interfaces are required for kernels in order to communicate with the host program or with other kernels. There are three types of kernel interfaces, namely the register interface, memory-mapped interface, and streaming interface. The register interface is used to accept scalar arguments from the host. The memory-mapped interface is for data transfer between the kernel and global memory on FPGA. It should be implemented with AXI4 Master interfaces. A pointer to the start of the memory address should be given by the host through the register interface once the memory is allocated. Finally, the streaming interface is used for data transfer between kernels. It should be implemented with AXI4 Stream interfaces. These interfaces can be automatically generated by the RTL kernel wizard for RTL kernels or generated by HLS tools for OpenCL kernels.

There are three execution models for a kernel, namely the sequential mode, pipelined mode, and free-running mode. In the sequential mode, kernels are triggered by the host to start execution. When the task is finished, the kernel will go to a ready state and inform the host. Then the host application can restart the task. The pipelined mode works in the same way except that the next task can be started

before the current task completes. On the other hand, the free-running mode does not need a trigger to start its work. Kernels in this mode run as they are programmed. A memory-mapped interface is not allowed in the free-running mode.

As we might have multiple kernels needed for a whole implementation, they should be linked together. Vitis will first compile the kernels into Xilinx objects (.xo files) and then link them according to users' configurations. The design will then be synthesised, placed, and routed into the binary files (.xclbin files). At the same time, host C++ codes should be also compiled into an executable file (.exe files). These important steps are part of the Vitis design flow that we followed in our project.

With the bitstream and host executable, the kernel can finally be run on FPGAs. Starting from the host program, it should first program the binary to FPGAs and then assign values to kernels arguments and allocate memory blocks. After that, the host program will trigger the kernels on FPGA to start their jobs. The kernels then acquire data from global memory and perform the computation workloads. Once the task is completed, the kernel will write the results back and inform the host. The host program can finally read the results back from the global memory. Users can repeat all of these steps as needed.

2.3.2. High Level Synthesis (HLS)

The design and test for hardware description language (HDL) is complex and sometimes cumbersome. High level synthesis [6] is a technology that aims to reduce the effort of developing lower level HDLs using higher level languages such as C, C++, and SystemC. With a higher level abstraction focusing on algorithmic level rather than register transfer level (RTL), developers can more easily design the logic and interfaces for hardware modules. The tools will analyze and compile the C code to generate HDL with the same functionality automatically. This automated process also helps reduce error codes.

Despite the convenience of HLS, the HDLs it produces are usually not as good as those written by experienced hardware developers [38]. In some certain cases, the hardware utilization and performance can be huge. The HLS generated HDLs have great potential to be improved by manual modification.

Luckily, some HLS tools already provide options for hardware optimizations. Since we are using Xilinx FPGAs, we take the Xilinx Vivado HLS tool as an example. It provides many `pragma` commands for the compiler to understand how do you want the code to be generated. A common example is loop unrolling as shown below.

```
loop: for(int i = 0; i < N; i++) {
  #pragma HLS unroll
  a[i] = b[i] + c[i];
}
```

Verification is important to verify the correctness of HLS codes. The Vivado HLS verification procedure includes multiple steps. The higher level code should be first verified to ensure its functionality. C-synthesis can do this with a C testbench file. This step is very fast as simple software code which takes a few seconds. Then the C code can be compiled into HDLs. HDLs should also be verified to ensure their correctness. After that, the HDL will be processed by synthesis tools to generate gate level implementations. Timing problems might occur at this point. After solving all problems through these steps, the HLS code can finally be deployed on FPGAs for a hardware test.

3

Alternative Solutions

In this chapter, we will discuss the alternative solutions for the RDMA network stack on FPGAs. As mentioned in Chapter 2, there are mainly three different top level architectures: the InfiniBand, RoCE v1 or v2, and iWARP. On the transport layer of these architectures, there is the same InfiniBand transport layer. This transport layer supports multiple services focusing on different functionality. It is free to select the transport model that suits our goal. The packet format will also vary if a different transport model is applied. In combination, the overall architecture and the inside transport model lead to an alternative solution tree. The purpose of this chapter is to discuss the alternatives and identify their strengths and drawbacks.

We will first discuss the possible transport layer models because it remains the same for each architecture. With a detailed comparison in terms of functionality and complexity, the best transport model will be selected. Then different architectures of the RDMA stack are presented in detail. The architecture structures will be shown to discuss their complexity and hardware requirements. As we already selected the transport models, packet headers can be determined for each RDMA architecture. The header overhead will be discussed and compared to show the appropriate architecture to implement on FPGAs. And finally, we show some of the related work in this field and discuss alternative developing paths for our work. At the end of this chapter, we will present the preferred solution.

3.1. RDMA Transport Models

There are mainly two types of RDMA transport models: reliable service and unreliable service as discussed in Section 2.2. The difference is obvious through their names. Reliable services ensure the messages are delivered in order, complete, and with no corruption, while unreliable services do little work in reliability. Here we introduce the three most basic and most used transport models, namely reliable connection, unreliable connection, and unreliable datagram. The comparison between them in terms of supporting operations, scalability, reliability, and supporting message size can be seen in Table 3.1.

Reliable Connection (RC)

As indicated by the name, reliable connection is a connection-based model that ensures reliability.

Connection in RDMA is established by QPs. With the RC model implemented as the transport layer service, there should be one and only one QP for each connection. This means that each machine needs multiple QPs for different connections to multiple remote machines. And each QP is matched with only one of the remote QPs.

RC makes sure that messages will be delivered only once, in order and correct [3]. A lot like in the TCP, the RC service uses many similar mechanisms to ensure reliability. A packet sequence number (PSN) is used to label every packet. This helps the HCA detect missing or out-of-order packets. An acknowledgement mechanism is applied to ensure messages are successfully delivered. And finally, CRCs are also appended at the end of the packets to detect packet bytes correctness. Error recovery is also an essential part of RC service which further improves its reliability.

Attribute		RC	UC	UD
Reliability	**Data order**	guaranteed	detected	no
	Data delivery	guaranteed	detected	no
	Corrupt data	recovered	detected	dropped
Operation support	**send**	y	y	y
	RDMA WRITE	y	y	n
	RDMA READ	y	n	n
Multicast		n	n	y
Message length (bytes)		2G	2G	4096
Scalability (M processes on N Processor nodes communicating with all processes on all nodes)		M*M*N	M*M*N	M

Table 3.1: Comparison between Reliable Connection, Unreliable Connection and Unreliable Datagram transport models in terms of reliability, supporting operations, scalability and supporting message size.

Unreliable Connection (UC)
Similarly, an unreliable connection model is also connection-oriented. There is a one-to-one correspondence between a local QP and a remote QP.

On the other hand, no reliability is guaranteed. UC does not apply acknowledgment mechanism. This means there is no acknowledgement header fields needed for it. But packet sequence number is still kept so that the responder can be notified when out-of-order problems occur. CRC is also implemented to detect data corruption.

Unreliable Datagram (UD)
Unreliable datagram model allows a source QP to send messages to one of many destination QPs on multiple destination end nodes [3]. Simply speaking, only one QP is sufficient to send data to multiple remote QPs. This leads to good scalability as the number of QP doesn't grow quadratically with the number of end nodes in a fully connected way.

UD does not check missing packets or out-of-order packets. Only CRC is implemented to drop those packets that carry corrupt data. Either the requester or the responder will not get informed when such an error is detected.

3.1.1. Reliability

In terms of reliability, there are mainly three indicators as listed below. They are also placed in Table 3.1.

1. Whether the data is transferred **in order** or not?
2. Is data guaranteed to be **delivered**?
3. Will **corrupt data** be recognized and recovered?

For the first metric, RC guarantees data order using PSN. Unordered packets and duplicated packets will be detected and handled. The two abnormal packets can be described by two conditions. The first is **duplicated packet**. When a response packet is lost in the fabric or just too late, the requester cannot get it on time. This causes the requester to send the same request packet again which has already been acknowledged by the responder. And duplication happens on the responder side. There is also another case where the first packet sent by the requester is delayed in the fabric for a very long period and it finally arrived after the re-send request packet has been already acknowledged by the responder. Duplication also happens on the responder side. The other condition is called **invalid request packet sequence**. This means the responder compares PSN for each packet and detects that one or more packets sent by the requester are lost.

The duplicated packets can be easily handled by validating the duplicated packets and giving ACK to inform the requester. The expected PSN will not be updated. For the invalid request packet sequence, it is handled as follows. The packets with invalid PSNs will be dropped as they are not what we expect.

The expected PSN of the responder will remain not updated. A NAK will be sent back to indicate the problem to the requester. Thus, the correct packets will be re-transmitted by the requester. With these mechanisms, data order is guaranteed for the RC model.

UC also needs PSN in its packets but unlike RC, there is no acknowledgement mechanism. In case of duplicated packets or invalid request packets, the UC service only drops them. Since no acknowledgement is required, the dropped data is literally abandoned. The responder side can detect these out-of-order packets but will not inform the requester. As for UD, the PSN field is kept only in case it is useful for the upper layer application to detect out-of-order packets. However, PSN is generally ignored with UD service. Thus, we would say that the data order is not guaranteed in UC and UD.

For the second metric, RC guarantees data will be delivered. The acknowledgement mechanism ensures this. Whenever a packet is lost, either the responder will notice and request a re-transmission using the NAK packet or the requester will automatically re-transmit it after not receiving ACK in a certain amount of time. Both UC and UD do not have acknowledgement mechanisms, so the data is not guaranteed to be delivered to the destination. A lost packet will be detected by UC as it checks PSN. However, the requester does not aware of any packet loss. As mentioned previously, PSN is ignored by UD. UD even cannot detect such missing data problems.

For the third metric, every transport model supports CRC, so data corruption can always be detected. However, the way different services handle data corruption differs. RC will drop the packet and generate the NAK packet after detecting corrupt data. The error data then can be recovered through the re-transmission. UC will also drop the erroneous packet but will only notify the responder side. The requester will not be informed. UD on the other hand will only drop the corrupt data silently without notifying anyone.

Depending on the use cases, reliability could play a crucial role or have very little impact. If the RDMA stack is used to transform important data for downstream processing where every byte of information matters, reliability is necessary. If consumer video streams are running over this RDMA stack, nobody cares whether there is one frame missed or blurred at some random point. This thesis project topic is initially brought up by a genetic processing use case. Every piece of genetic data is important for a reliable and accurate result. So the RC model needs to be applied. Although our RDMA stack is targeting a wider range of use cases, the RC model is preferred because of its reliability.

3.1.2. Operation Support

In terms of functionality for the transport servers, three important operations are discussed. Namely, **send**, **RDMA READ** and **RDMA WRITE**. Besides, the ability of multicasting is also presented.

As the most complete and complex transport model, RC is capable of all three RDMA operations. UC supports only the send and the RDMA WRITE but not RDMA READ. While UD only supports send operation.

As introduced in Chapter 2, send is a channel semantic operation and is not considered as an RDMA operation. The requester only pushes data from its QP to the responder's QP without specifying a remote address. The responder deals on its own where to place the data. It does not necessarily require acknowledgement and all three of the transport models support it.

RDMA WRITE operation is similar to send operation because they both send data from the requester to the responder. But there are several differences. First, the requester (or sender) specifies the virtual remote address to place the data. So the destination address to write the data is decided by the requester rather than the responder. Second, the requester explicitly indicates the message length, while during send operation the message length can only implicitly acquired by counting received packets. Acknowledgement is not necessary and can be left out for RDMA WRITE operations. These explain why UD does not support any RDMA commands while RC and UC support the RDMA WRITE operation.

RDMA READ operation is the reverse of RDMA WRITE operation but more complex. Starting from the requester, it sends RDMA READ packets telling the responder the virtual remote address where data is located as well as the message length it requests. With such RDMA READ packet, the responder sends back the required data. Data will then be written to the local virtual address of the requester. The RDMA READ response includes acknowledgement extension transport header. This means the RDMA READ requires acknowledgement mechanism. Thus only RC supports RDMA READ operation.

As a special function, multicast is only supported by UD service. But it is also an optional function for UD. For some use cases where the same blocks of data need to be broadcasted to multiple downstream machines, this can save a significant amount of hardware resource and bandwidth. The multicast defined in the InfiniBand specification involves network layer protocols, so we will not discuss it in detail.

For this project, if we still targeting the genetic use case, the send operation is enough to connect the pipelined generic algorithm. However as described in the last subsection, the UC can only perform send operation with a message size equal to or smaller than the MTU. Plus, it requires the remote side's involvement and is not considered a pure RDMA operation. So the UD service is not a strong candidate. Besides, the send and RDMA WRITE operations are both data transformations in the same direction. The RDMA READ operation is vital for a bidirectional network stack. Although bidirectional data transform can be achieved by only send or RDMA WRITE operation, there is no way to entirely bypass the data source host. The operation should always be initiated by the host where the data locates. With RDMA READ also supported, the network stack allows hosts to communicate with each other more flexibly. This also means more optimization possibilities in terms of CPU bypass.

3.1.3. Other Comparisons

Worth mentioning that the UD only allows message size within a packet size. So the message size is pretty small ranging from 0-4kB depending on the Ethernet packet size. For example, it would be limited by the Maximum Transmission Unit (MTU) if it works over IP protocol [22]. Both RC and UC support a message size up to 2GB (2^{31} bytes). The upper layer application can benefit a lot from the support for large message size. The user application does not need to segment the message into small pieces that can be fit into one packet. It also does not have to take care of the header difference. For example, if an extra header field is needed for the first packet carrying data, the size of the actual payload should be shrunk to comply with the MTU. A message level abstraction rather than packet level abstraction significantly reduces the complexity for developing the upper layer applications.

Scalability is another important metric when we talk about big data. The scalability for RDMA transport services is evaluated by how the number of required QPs is related to the number of processes. As indicated in Table 3.1, we consider the case where M processes on N processor nodes need to communicate with all other processes on all nodes. Both UC and UD need $M \times M \times N$ QPs on one processor (which means in one HCA) because they are connection-based services. Each local process endpoint needs one QP for a single remote endpoint. Thus, $M \times N$ QPs are required for this local process. While there are M local processes, the total number becomes $M \times M \times N$ for the local HCA. On the contrary, UD service requires only M QPs for all these communication combinations due to its connectionless characteristic. Only the number of local endpoints matters because a receive queue can accept messages from any remote send queue and the send queue is able to send data to any remote receive queue. If scalability is prioritized, UD has much better performance. But the number of QPs does not actually require that many hardware resources. Plus there is the Extended Reliable Connection service (XRC), which has improved scalability with $M \times N$ QPs on each HCA.

Finally, the overall complexity of development should also be taken into account. With the least function supported and no reliability mechanisms, UD is the simplest transport service. RC is the most complicated one with much more reliability capability. Apart from the RDMA READ operation implementation, reliability mechanisms should be integrated. Some extra packet header processing such as PSN validation and the acknowledgement processes are important parts and are specific to the RC model. The development effort for it is significantly higher than UC and UD. UC sits in the middle. It can be seen as a subset of the RC service.

In conclusion, RC is the most suitable InfiniBand RDMA transport model for our goal considering all these metrics. It is also the model that requires the most effort to build. A preferable path is to develop UC as a starting point. After verifying its functionality, we can insert the reliability mechanisms as well as the extra RDMA READ operation support. Furthermore, upgrading RC to XRC would also be beneficial in terms of scalability.

3.2. RDMA Architectures

In Chapter 2 we already showed the structures of different architectures. Here in this section, we compare InfiniBand, RoCE v1, RoCE v2, and iWARP architectures with each other. To compare them, we consider the deployment efforts and packet header lengths.

3.2.1. Deployment Efforts

What we mean by deployment efforts refers to the effort to update or test networking hardware and the effort for FPGA development.

InfiniBand is working on its own InfiniBand fabric. This means that in most cases where the existing fabric is an Ethernet network, a great amount of hardware needs to be replaced. For example the network switches, routers, and even cables. There are Ethernet switches that also support InfiniBand traffic such as the latest NVIDIA NDR 400Gb/s InfiniBand Switch. Such hardware supporting both Ethernet and InfiniBand are more and more widely deployed in enterprise and academic data centers or high-performance computing (HPC) infrastructures in recent years. But they are not accessible in many cases. If the fabric of our developing environment does not support InfiniBand, then a huge amount of effort needs to be paid to upgrade the entire fabric. This networking hardware requires not only a significant amount of money but also a huge human effort to deploy and test.

On the other hand, RoCE and iWARP architectures both work over Ethernet fabric. No modification is needed for networking hardware to enable either one of them.

Note that RoCE stands for RDMA over Converged Ethernet. A Converged Enhanced Ethernet (CEE) is a single Ethernet with support for many protocols, which is also known as data center bridging. This concept is part of the 802.1 standard [33]. The main purpose of CEE is to combine the adapters and cables of different protocols into one single more effective set. Thus, CEE helps to simplify the interfaces and network stacking for data centers. What makes the converged Ethernet stand out is that it applies Priority-based Flow Control (PFC) to handle traffic congestion problems. The objective of PFC is to handle congestion at end nodes rather than in the middle of a complex network [4]. By propagating the congestion of the buffer at one middle node to its upstream devices recursively, there will be no packet being dropped inside the network. And the sender can stop sending packets on time. PFC defines 8 levels of priority from 0 to 7. Ideally, each priority level link is independent of the other. Whenever congestion occurs for one priority level, the traffic running on that level will be pulsed. However, traffic on other priority levels is not affected. With PFC applied in converged Ethernet, its fabric can be lossless.

Hence, RoCE implementation and test require the network fabric to support PFC. As PFC is already a widely used flow control technology in data centers and computing clusters, we assume the converged Ethernet fabric is not hard to access.

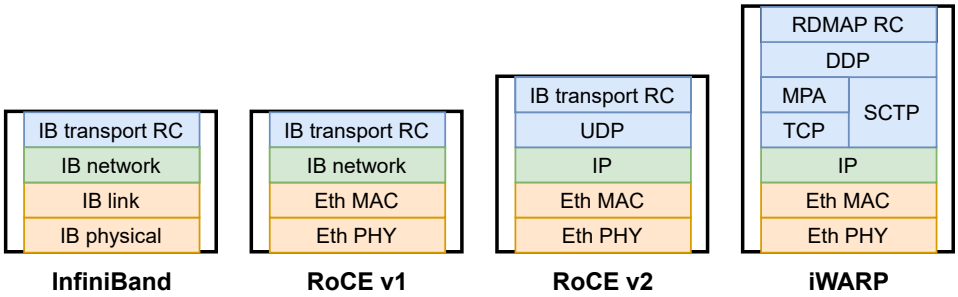


Figure 3.1: The top level network stack structures for different RDMA architectures with RC transport model selected.

In terms of development effort, we start the comparison from the network stack block structures in Figure 3.1. The application layer is omitted because they are out of the scope of our network stack. The application layer for all these architectures should work in the same way. The physical layer is separated from the link layer because they are often working on specific hardware like PHY chips rather than on FPGAs. So the development effort for the physical layer is compared separately.

Starting from the InfiniBand, all of the InfiniBand link layer, network layer, and transport layer need to be developed on FPGA. The InfiniBand physical layer supports both normal copper cables and optical fibers [9]. The most commonly used is QSFP connectors, which support a bandwidth of one

hundred gigabits per second. For all the three architectures on the right hand, Ethernet physical layer, as well as Ethernet MAC layers, are used. As one of the most widely used protocols, there are already many mature MAC layer implementations on FPGA off the shelf either from academia [31] or industry [39, 41, 44]. The Ethernet PHY chips are also included on FPGA boards that provide networking interfaces such as the Alveo U280 [43]. We can easily access the MAC layer intellectual properties and integrate them into our network stack. Similarly, the UDP/IP and TCP/IP are also widely used protocols that certainly have already existing works done for FPGAs out there. Thus, the development efforts for these protocols can be considered relatively small.

We can see that the iWARP has the deepest stack with 2 more layers that others do not have. And iWARP is indeed the most complicated stack among these four. Above the TCP layer, there is the Marker PDU Aligned Framing (MPA) protocol. Working as the adaption layer between TCP and DDP, there is some interfacing logic for MPA to interact with both the TCP layer and DDP layer. The DDP protocol is a rather complex protocol that has a varying header length for different conditions. For example, the DDP header for an RDMA READ operation can be 3 times the size of the header for an RDMA WRITE operation [24]. The DDP applies a tagged buffer model to achieve the same remote direct memory access capability. The local node can advertise a tagged buffer to the remote node with the Steering Tag (STag) so that the remote node can directly specify the address to place data. DDP supports another service called the untagged buffer model which is similar to the RDMA send operation. On top of DDP, there is RDMAP. RDMAP is closely bound with DDP and is usually combined with it. So the RDMAP header field is often referring to the DDP header. These middle layers in iWARP require a considerable amount of effort to develop.

Finally, the InfiniBand transport layer is the same for all architectures except for iWARP. Since the RC model is selected for our design, it requires the most effort to build. This is also one of the focus areas of our development.

3.2.2. Header Comparison

Every packet needs headers that contain certain fields of information so that it can be transmitted correctly in the network. The headers take up spaces inside the packets so the actual data throughput is smaller than the maximum bandwidth of a fabric. Packet headers introduce data overhead in the networking packets.

The header overhead introduced by networking protocols may differ for each of the RMDA architectures since they use different network protocol stacks. However, the stack with the most layers does not necessarily have the highest header overhead. There is a debate between the company Chelsio and Mellanox about what architecture has better performance. Mellanox claims that RoCE can achieve higher throughput and lower latency [32] while Chesio denies that iWARP has more packet overhead [5]. In this subsection, we will discuss and compare the header length for several types of packets for every RDMA architecture. Although the header overhead might be insignificant, it is still interesting to have a close look at these header fields of RDMA architectures.

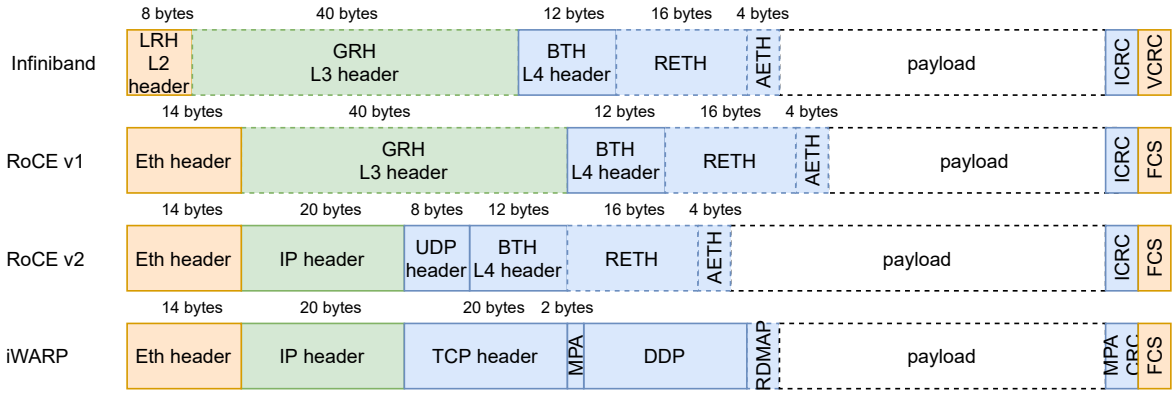


Figure 3.2: A general comparison of the complete Ethernet packet for different RDMA architectures.

The Figure 3.2 shows the complete packet headers for all four RDMA architectures in general cases. Each block indicates one type of packet header. The width of each header is proportional to its length,

except for the payload blocks. So that the amount of header overhead can be observed. The actual lengths are also marked on top of each block. The total width of every packet is fixed to indicate the maximum Ethernet packet size restriction. The payload blocks are the leftover parts showing how many vacancies are left for the application data. A narrow payload block means a heavy header overhead. What we mean by general is that we show all of the header fields possibly used for each architecture in this figure. Blocks with dashed broader are the fields that can be omitted or used on certain occasions.

The link layer not only introduces headers but also Frame Check Sequences (FCS). For the Ethernet linker layer, in total 14 + 4 = 18 bytes are occupied [1]. For InfiniBand, the Local Routing Header (LRH) plus its Variant Cyclic Redundancy Check (VCRC) contributes a sum of only 8 + 4 = 12 bytes overhead.

The networking layer handling routing among wide area networks is also very different for InfiniBand network and IP. Global Routing Header (GRH) consisting of 40 bytes takes up a great amount of space. While the IP header only needs 20 bytes. If you take a look at what fields the GRH and IP header have, they are actually very similar. The GRH uses 128 bits Group Identifiers (GIDs) to indicate both source and destination addresses but IP uses 32 bits IP addresses. This brings us to the IPv6 protocol. GRH is exactly the same as the IPv6 header. Minus the 24 extra bytes for local and remote addresses, GRH is more lightweight than the IP header. Thus, the GRH has a higher overhead if IPv4 is used but is the same with the IPv6 header. Besides, GRH can be omitted if the network stack works only for local area networks.

The transport layer is more complex. Even with a single InfiniBand transport layer, there are multiple headers used. Namely, Base Transport Header (BTH), RDMA Extended Transport Header (RETH), and Acknowledgement Extended Transport Header (AETH). BTH is indispensable for RDMA transport services. Several important fields are presented in BTH. The OpCode is the first field of BTH that shows what type of operation this packet does or what type of data it carries. The partition key is also part of BTH. The partition key specifies which partition the remote QP is a member of. An invalid partition value can lead to a refusal of communication. Destination QP number (QPN) is declared in BTH as well. It shows the remote network adapter which QP is the target. Wrong QPN will also lead to packets being dropped. Finally, the Packet Sequence Number (PSN) is attached within BTH. It is the key component to ensure reliability as discussed in Section 3.1. RETH contains supplement fields for RDMA operations. A remote virtual address is specified so that the remote network adapter knows where the data is to be written to or read from. The address takes 8 bytes of space. Protection key R_key is also part of the RETH. R_key needs to be priory exchanged through other network protocols so that both endpoints have access to the other end point's memory. Finally, the DMA length is specified in RETH. Both RDMA WRITE and RDMA READ operations use this field to indicate the length of data to be transferred. The length is the message size up to 2GB, which occupies another 4 bytes (32 bits) in RETH. AETH header is only required for several cases such as ACK packets and the first or last RDMA READ response packets. It takes only 4 bytes, with 3 bytes carrying the Message Sequence Number (MSN) and 8 bits showing the ACK status. MSN is different from PSN, its purpose is to simplify the completion process of a working queue element at the requester side.

RoCE v2 is based on UDP, so there are extra 8 bytes UDP header overhead. However, even combine the UDP and IP header they are still less than the GRH (in the case of IPv4 applied).

iWARP has a different transport layer protocol stack. TCP is applied which brings 20 bytes of overhead. It offers reliability in return. The MPA is very lightweight with only 2 bytes that works effectively as a packet length field. DDP and RDMAP are always handled in a combined fashion. The RDMAP header can be left out for many RMDA operation cases. The DDP header has a floating length from 14 bytes to 46 bytes [24].

With all the packet headers explained, we can now perform a more detailed and specific header comparison. The RC transport model supports all three operations we mentioned: send, RDMA WRITE and RDMA READ. As discussed in Section 3.1, RDMA WRITE operation is preferred than send operation for a flexible network stack. We only compare the RDMA WRITE and RDMA READ packets. Note that RDMA READ response packets are also compared as it differs greatly from the other two packets. It is also an example where AETH is used.

The link level and network level headers remain the same for different transport layer operations. These lower layer headers are already discussed previously, so we only discuss the InfiniBand transport headers. Besides, The first three architectures have a common InfiniBand transport layer so they are

[1] In this thesis the Ethernet packet refers to the Ethernet frame which does not include the preamble and SFD. Preamble and SFD always exist for Ethernet packets. They are not involved in our development.

explained together. Only the iWARP has a different set of RDMA transport headers.

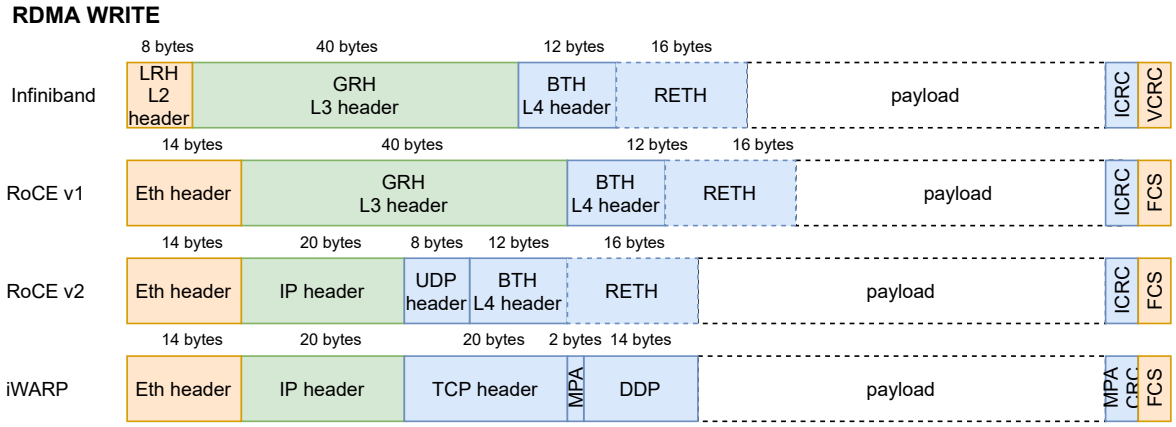


Figure 3.3: The packet structure of an RDMA WRITE packet for different RDMA architectures.

RDMA WRITE headers

First, we consider the RDMA WRITE packets. The header comparison is shown in Figure 3.3. An RDMA WRITE operation carries the data as well as the target destination memory address.

For the three architectures using InfiniBand transport layer specification, RDMA WRITE operation has mainly 4 types of OpCode: *RDMA WRITE first*, *RDMA WRITE middle*, *RDMA WRITE last* and *RDMA WRITE only*. If the amount of data to be transferred is so small that it can be fit into one Ethernet packet, one packet is sufficient for this RDMA WRITE operation. In this case, OpCode *RDMA WRITE only* is used. Otherwise, message data are cut into segments and carried by a series of packets. The first and last packets use the *RDMA WRITE first* and *RDMA WRITE last* OpCode respectively. While the packets in the middle use *RDMA WRITE middle* as the OpCode. As we know, BTH carries the necessary information for a packet to successfully reach its destination QP. It is always present in RDMA packets. What may vary is the RETH, which carries the operation parameters (remote address and data length). For the first or the only one packet of an RDMA WRITE operation, RETH is indispensable. But it is omitted for the middle and last RDMA WRITE packets because the first packet already carries the parameters for RDMA operation. Data are directly attached to BTH for these packets.

iWARP on the other hand does not share the same headers for the same RC service on the RDMA transport layer. For an RDMA WRITE operation, the DDP segment is a normal tagged buffer model header with a length of 14 bytes. The remote STag and the tag offset are the main parameters it carries. They correspond to the remote address field in RETH.

In comparison, the iWARP is the architecture with the least header overhead, a tie with RoCE v2, and better than RoCE v1 or InfiniBand. However, if we consider the middle packets where RETH can be omitted, the RoCE v2 takes the lead.

RDMA READ headers

Then we compare the RDMA READ operation. The header comparison is shown in Figure 3.4. The RDMA READ operation carries only the operation meta but not data.

The header of the RDMA READ operation is much simpler for the top three architectures compared to an RDMA WRITE operation. The read operation requires only one packet since there is no data being transferred. The OpCode for it is *RDMA READ Request*. The same as a write operation, RETH is necessary as it is always the only packet.

iWARP's story is different. Although the same tagged buffer model is used for an RDMA READ operation, the DDP header segment occupies 46 bytes. Apart from the normal DDP fields, extra RDMA fields are added to it such as remote queue pair number, message sequence number, and message offset. These extra header fields introduce a lot of overhead to the RDMA READ packet.

The RoCE v2 stands out again with the lowest header overhead for RDMA READ operation. The difference between RoCE v2 and iWARP architectures is a significant 32 bytes. Note that the entire packet for an RDMA READ request in RoCE v2 contains 78 bytes including Ethernet headers. The

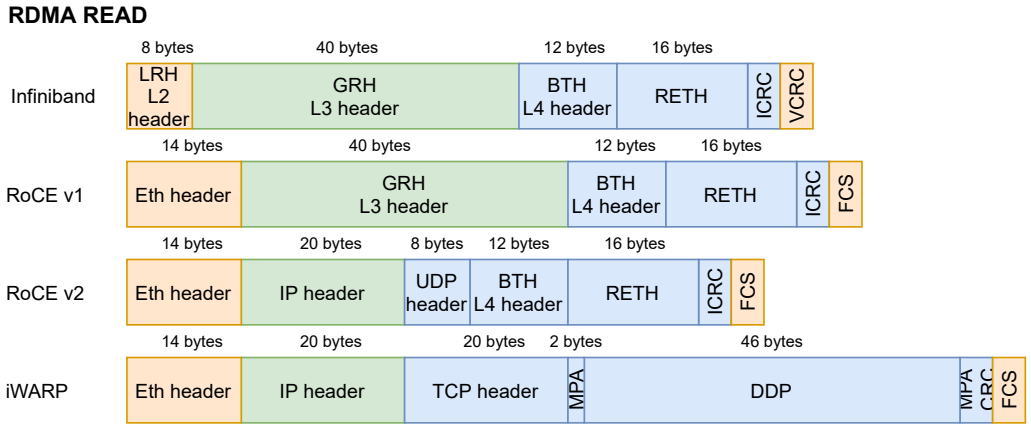


Figure 3.4: The packet structure of an RDMA READ packet for different RDMA architectures.

iWARP RDMA READ request header is 41% larger than that. However, this read request packet is a single packet that does not add overhead to data transform. When the message size is large enough this overhead of a single packet has a very small impact on the overall data throughput.

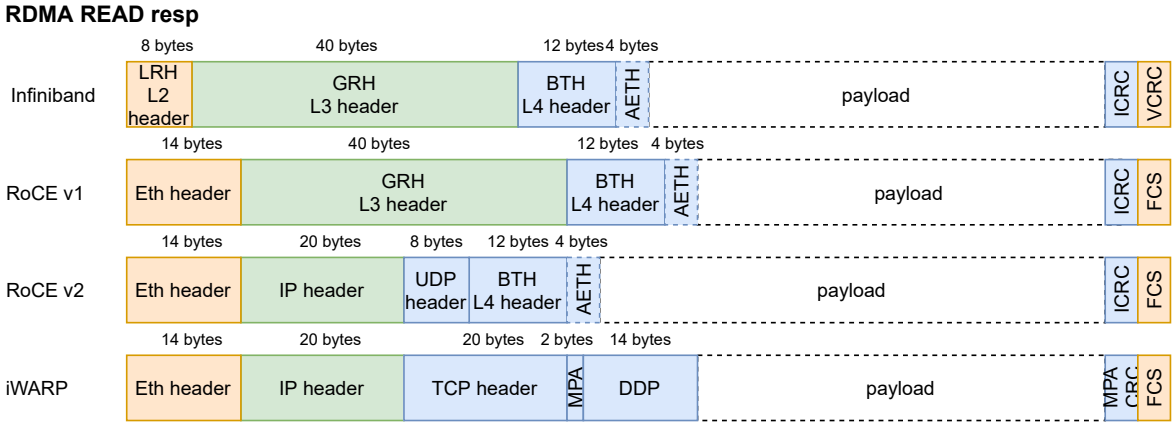


Figure 3.5: The packet structure of an RDMA READ response packet for different RDMA architectures.

RDMA READ response headers

Finally, we compare the RDMA READ response packet headers. The header comparison is shown in Figure 3.5. The RDMA READ response carries not only the response data but also acknowledgement.

For the top architectures, BTH is still a must. The OpCode for RDMA READ response is similar to RDMA WRITE with four types. Namely, *RDMA READ response First*, *RDMA READ response Middle*, *RDMA READ response Last* and *RDMA READ response Only*. The AETH is surely required for the first or the only response packet to acknowledge this request. The *RDMA READ response Last* packet also needs AETH to acknowledge that the entire read operation is completed. The middle packets do not necessarily require ACKs so the payload is attached directly to BTH. We can see from the figure that the RDMA READ response has the least packet header overhead among the three types of operations we discussed.

The DDP header filed for the RDMA READ response packet is the same as the RDMA WRITE packet. It occupies 14 bytes of space.

So the RoCE v2 wins again not surprisingly in terms of RDMA READ response packet header overhead.

In conclusion, the RoCE v2 introduces the least header overhead in all three types of packets. The greatest difference appears in RDMA READ request packets. But the read request packets are just a minority of the traffic because one RDMA READ request is usually followed by thousands of

RDMA READ response packets or even more that carry actual data. The difference is not significant for RDMA WRITE or RDMA READ response packets. With only 10 bytes of difference in one packet which usually works close to MTU (1500 bytes), the difference is below 0.7%. This corresponds to a 0.7Gb/s difference for a 100Gbps Ethernet network.

Note that we did not dig deeper for the DDP protocol in iWARP architecture. If the DDP header can also be omitted for those ”middle” packets carrying data, it is a very competitive solution with almost the same small header overhead as RoCE v2.

3.3. Summary and Development Paths

In this section, we will first conclude and make our selection for the RDMA transport service and architecture. Then we will show the existing work relative to RDMA implementations on FPGAs. Finally, we will decide the development path of our work.

		Alternatives		
Criteria	**Weight**	**RC**	**UC**	**UD**
Operation support	0.25	5	3	1
Reliability	0.25	5	2	0
Message size support	0.20	5	5	0
Scalability	0.15	1	1	5
Implementation effort	0.15	0	2	5
Score		**3.65**	**2.70**	**1.75**

Table 3.2: RDMA transport layer service alternatives selection table. Higher numbers mean better.

			Alternatives			
Criteria		**Weight**	**InfiniBand**	**RoCE v1**	**RoCE v2**	**iWARP**
Deployment effort		0.50	0	5	4	4
Header overhead	RDMA WRITE	0.20	3	1	5	1
	RDMA READ	0.10	4	3	5	0
	RDMA READ resp	0.20	4	3	5	3
Score			**1.80**	**3.60**	**4.50**	**2.80**

Table 3.3: RDMA architecture alternatives selection table. Higher numbers mean better.

As we already did a detailed comparison for RDMA transport services and architectures, two evaluation tables are made to show the final selection. As shown in Table 3.2 and Table 3.3, every alternative is evaluated by the metrics we discussed. The criteria are assigned with weights to indicate their importance. The weights are normalized so they sum up to one. Scores are given to each of the alternative per criteria on a 5 point scale. A higher score means better.

The tables show that an RDMA configuration using **RoCE v2 architecture** with **RC transport service** is the best solution. To achieve our goal of RoCE v2 network stack on FPGAs, three different approaches are ahead of us.

1. Purchase commercial FPGA Intellectual Properties (IP) on RoCE v2 stack support. Or connect FPGA to a commercial RoCE v2 capable network adapter and develop only the interface on FPGA.

2. Looking for existing work about RoCE stack as a starting point. And construct the complete network stack with it.

3. Build the entire RoCE transport layer from scratch.

Note that only the existing work about the InfiniBand transport layer is our focus. There are already many works done for the lower Ethernet MAC and IP layers as discussed in Section 3.2. Here we show some of the related commercial products and existing work.

Commercial Products
There is a Xilinx IP called ETRNIC (Embedded Target NIC) that supports RDMA with RoCE v2 implemented [40]. It works on a 100 Gbps data path and supports both RDMA READ and RDMA WRITE operations. However, it supports only the outgoing direction but not the incoming direction. Hence it can only work as the requester. Reliability is also guaranteed by the re-transmission mechanism. It is capable to have 127 QPs connecting to different remote nodes concurrently. In a word, it is a great but not optimum product for our goal. To be able to work as the responder is certainly important. Another problem with this product is the lack of flexibility. We are not able to add components or functions inside this IP.

Apart from it, NVIDIA networking provides many Mellanox networking adapter cards that support RoCE v2 such as the ConnectX®-3 Pro Adapter Card[2]. It fully supports RoCE v2 as well as InfiniBand. It can do all the work of networking below the transport layer. The adapter has an FPGA attached to it to gain extra programmability. It also offers high bandwidth PCIe ports so they can be comfortably connected to external FPGA boards. The only thing left for our development is to perform the connection and build a wrapper with the desired interface on FPGA. There is a similar work that develops RoCE v2 UD service on FPGA and did a comparison against the Mellanox ConnectX-4 adapter for an acquisition system [17].

We like to mention one more interesting product here, the NVIDIA networking department also provides BlueField series DPU[3]. DPU is a new infrastructure-on-a-chip that includes networking, ARM CPU, NVIDIA GPU, storage and security on the same chip. RoCE is also supported in such DPUs.

Existing TCP/IP and UDP/IP
The predecessors have already developed many great TCP/IP and UDP/IP stacks for FPGAs. Dollas et al. showed an open source TCP/IP stack IP for FPGAs in VHDL in 2005 [7]. It is a complete network stack with all necessary protocols like ARP and ICMP. UDP is also supported in their work. A total of 700 Mbps full-duplex throughput at 37.5 MHz was achieved on Xilinx Virtex 2 FPGAs. [11] showed a UDP/IP stack working at 1 Gbps. Nikolaos et al. also designed a UDP/IP core with gigabit Ethernet for communication between FPGAs and PCs [2]. A centralized structure is used in [13] to implement a TCP/IP stack. With the capability of receiving stream data at 4Gbps while sending data at 40Gbps, it shows good potential for asymmetric video streaming tasks. In 2015, Sidler et al. presented a scalable 10Gbps TCP/IP stack for FPGAs [28]. This TCP/IP stack applies novel data structures for connection tables and uses external memory as buffers so that it supports over 10,000 concurrent TCP sessions. This is the first open source TCP/IP stack that has huge scalability. UDP is also included in their work. Based on this work, they also introduced Limago [25] that works at 100Gbps Ethernet in 2019. Note that most of the implementation for their TCP/IP is written using HLS. The work is also published on Github by ETH Zurich systems group[4].

Existing RoCE v2
David Sidler also presented an implementation for RoCE v2 on FPGAs as part of his Ph.D. thesis [27, 29]. Although the work is not entirely open-sourced, the InfiniBand transport layer IP is available at the same Github repository. The RC service is also developed with HLS in C++. The transport layer model for this stack is RC service which is exactly what we need. What's more, there is a RoCE v2 IP that already combined the UDP and IP layers. It works at 100Gbps bandwidth and also supports good scalability. There is another RoCE v2 implementation on FPGA for a data acquisition system at 100Gbps presented in [17]. However, there's no available code for us to use.

With an academic and open source spirit, we would like to build the RoCE v2 network stack on top of the existing open source projects. Thus, the work done by David Sidler is the perfect starting point for our thesis project. The existence of this InfiniBand RC service as well as all necessary lower level protocols such as IP, UDP, and ARP help us save a lot of effort building the entire RoCE v2 network stack.

[2] https://www.mellanox.com/related-docs/prod_adapter_cards/PB_Programmable_ConnectX-3_Pro_Card_EN.pdf
[3] https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-3-dpu.pdf
[4] https://github.com/fpgasystems/fpga-network-stack

4

Architecture and Implementation

To ensure an efficient development process, a step-by-step plan was developed and carried out. The entire development road map is explained below.

1. Perform simulations to verify the **`rocev2` HLS IP**.
2. Simulate the RoCE kernel in a **loop back setup**.
3. Verify the entire stack in a **board to board test**.
4. **Banchmark** the entire stack on FPGA boards.

First of all, the `rocev2` HLS IP is tested with Vivado C-simulations before the development of our RoCE Vitis kernels. The functionalities for RDMA READ/WRITE operations have been validated. This C-simulation aims to verify the IP at the highest level. It is the simplest and least time-consuming step for debugging. After the RoCE kernel has been developed, the loopback setup is used to verify the RoCE kernel alone with behavioural simulations. This step is done in Vivado with the help of the Vivado RTL kernel wizard. Other kernels such as the read User kernel are also tested at the RTL level. Then, the kernels are linked together and built into binary files for FPGAs. The design is now tested on real FPGA hardware. The functionality for this entire RoCE stack is verified by a read-write-read test between two FPGA boards. This step is done by capturing internal signals through Vivado Hardware Manager and observing HBM data through Xilinx XRT tools. Finally, benchmarks are carried out for the RoCE network stack.

In this chapter, the architecture of our RoCE v2 network stack on FPGA will be presented. Implementation details such as the block diagrams, instantiated IPs, and interfaces for each kernel will also be discussed.

4.1. Architectural Overview

Thanks to the Xilinx Adaptive Compute Clusters, we can easily design and test FPGA network stacks with a well-designed 100Gbps network fabric. The FPGAs on XACC clusters are Alveo boards. Each Alveo board is connected to one host CPU through PCIe. Every Alveo board is also connected to a 100Gbps Ethernet switch through its QSFP port 0. FPGAs are linked directly through QSFP port 1 as pairs. There is already an open source 100Gbps TCP/IP stack on the Alveo U280 platforms [10]. Our design is initially implemented on the Alveo U280 platform, but can also work on Alveo U250. We used this open source TCP/IP stack project [1] as a starting point. Our RoCE stack is fully compatible with the TCP/IP stack and is an addition to it. Our work is also open sourced and is available here [2].

4.1.1. Top Level Structure

As we are using the Alveo platform, the designed network stack should be implemented as Vitis kernels. Following with the kernel division in [10], we also implemented our RoCE kernel with three kernels. Namely, the User kernel, the RoCE kernel, and the CMAC kernel.

[1] `https://github.com/fpgasystems/Vitis_with_100Gbps_TCP-IP`
[2] `https://github.com/hcxxstl/Vitis_RoCE`

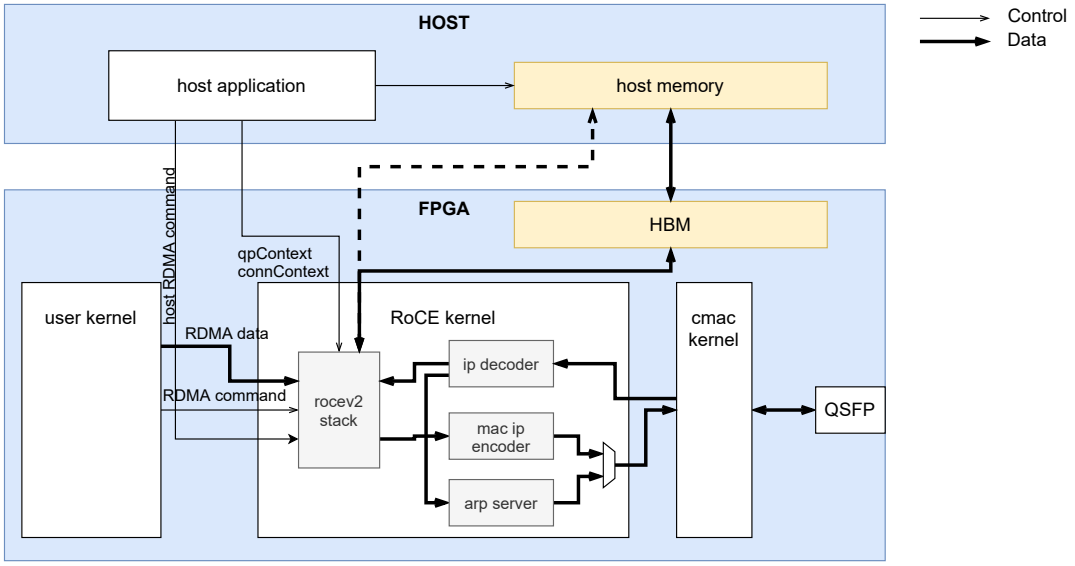


Figure 4.1: The top level structure of the designed RoCE v2 network stack on Alveo FPGAs.

The Figure 4.1 shows the top level structure of our RoCE v2 network stack. As explained in the Alveo platform execution model, there are both HOST and FPGA involved. They are indicated with blue blocks. Three kernels on the FPGA are shown as white blocks. The yellow blocks are the memory for hos and FPGA. The thin arrows represent control or meta signals, while the bold arrows indicate data flows.

On the host side, the host application is developed for several purposes. First, it is used to program the bitstreams to FPGAs. Second, parameters for each kernel should be set and passed to the kernels on FPGAs. The RoCE kernel requires many parameters to establish an RDMA RC connection. These parameters include the queue pair numbers, message sequence numbers, IP address, and UDP ports for both the local and remote sides. The local virtual address and the rKey are also part of parameters that need to be written to the RoCE kernel by the host. Third, to communicate with the memory on FPGAs, host memory should also be allocated and initialized. This is done by the host executable codes. Finally, the host application can hang the kernel and this is very helpful for debugging purposes. The Integrated Logic Analyzer (ILA) module is inserted in the interested part of the kernel on FPGAs to detect the values of specific signals. Triggers need to be set up before the operations on FPGAs happen. The host program can break after the bitstream is written and before the tasks are executed. So that hardware triggers can be properly set to catch waveform.

The host can also access the HBM on FPGAs through the Vitis shell using `xbutil` command. This is very useful to verify whether the RoCE kernel has read data from or written data to the global memory on FPGAs. The global memory can also be initialized with certain patterns by the host so that we can easily monitor the data during hardware debug.

Now we look at the FPGA. The three kernels are connected in sequence. The User kernel issues RDMA operations, which are on the application layer. The RoCE kernel handles the RoCE protocols and memory access. It works as the transport layer and network layer. The CMAC kernel does the MAC layer job, which is on the link layer. Finally, the CMAC kernel is hooked to the GT pins provided by the Vitis shell and then connected to a QSFP port. The network stack is then connected to the outside network fabric.

The User kernel is designed to be used as user-defined applications. It is the only kernel that is not part of the general-purpose RoCE network stack. Developers can customize this kernel for different applications as long as the interface is correctly designed. There are two ports from the User kernel to the RoCE kernel. One for RDMA commands, the other for RDMA data. The User kernel can issue RDMA operations through the RDMA commands. The User kernel can also provide the data to be transmitted through the RDMA data AXI stream. Of course, the user kernel can also not provide data. In that case, data should be fetched from an address of the memory on this FPGA. If there is no

application at all for the User kernel, a dummy kernel should be used to connect the two ports so that there won't be signal lines left open which might introduce metastable states.

The RoCE kernel is the largest and the key kernel in our RoCE network stack. There are four IPs in it that work together to enable the RoCE functionality. The `rocev2` IP includes the InfiniBand Reliable Connection transport service. Note that UDP and IP layers are also wrapped inside the `rocev2` IP. The `ip_decoder` IP analyses input Ethernet packets and distribute them to either the `rocev2` IP or `ARP` IP based on their header fields. The `mac_ip_encoder` IP takes the RoCE packets (more accurately with UDP header) and inserts IP as well as Ethernet headers. The MAC addresses are acquired from the `ARP` IP based on target IP addresses. The `ARP` IP contains a lookup table where the correspondences between MAC addresses and IP addresses are stored. It is also in charge of sending the ARP request packets or dealing with ARP response packets. These four IPs combine into a minimum protocol set to form the RoCE v2 network stack. More IPs can be added for a more complete network stack. For example, an ICMP IP enables echo tests which are known as the `ping` tests. This could be helpful to detect if the network stack is online or not.

The CMAC kernel of our design is the same as the one in [10]. It includes the Xilinx UltraScale+ Integrated 100G Ethernet Subsystem IP which handles the link layer functionalities. It is connected to GT pins in the Vitis shell to connect the QSFP port. As mentioned above, QSFP port 0 of the Alveo board in XACC is connected to a 100Gbps switch rather than directly to another Alveo board. For a more realistic test, the QSFP port 0 is used.

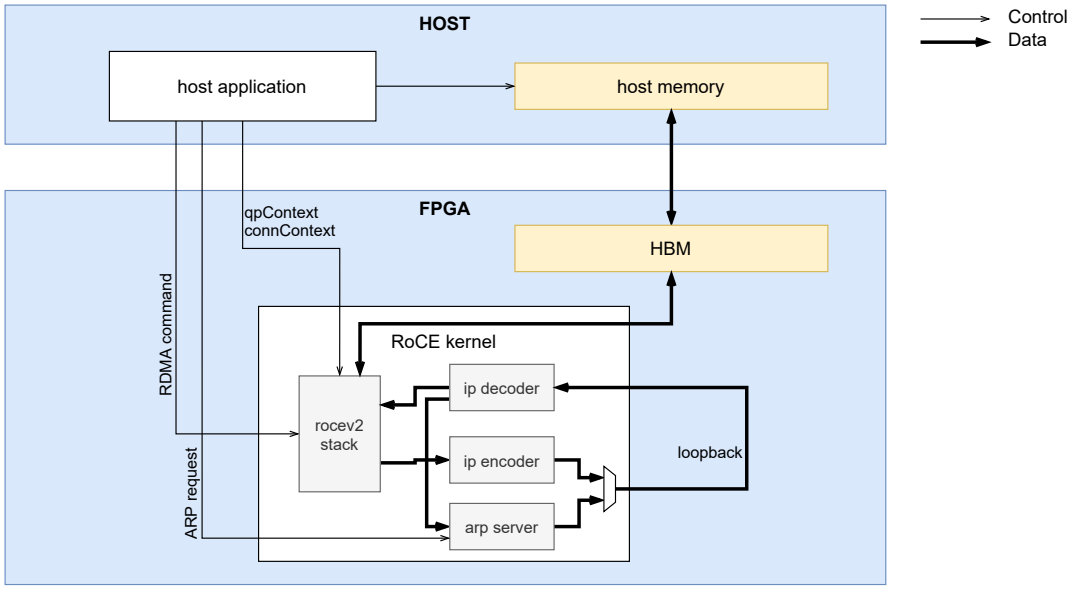


Figure 4.2: The top level structure of the RoCE v2 network stack on Alveo FPGAs with a loop back setup.

Besides the complete structure, there is a simpler loop back structure for verification purposes. The development of the entire stack is a step-by-step procedure. This loopback setup is a stepping stone for the complete RoCE v2 network stack.

As shown in Figure 4.2, the User kernel and CMAC kernel are left out and only the RoCE kernel is kept. With this setup, we can more easily determine the functionalities of the RoCE v2 transport and network layers. With only two RDMA operation tests, the RDMA WRITE and RDMA READ operations, we can already verify the correctness of our network stack design. This can be done by the host executable passing the operation parameters. RDMA commands are issued by the host application so that there is no User kernel needed. OpCodes and other necessary parameters are also passed to the RoCE kernel by the host code. CMAC kernel is removed to further simplify the structure. All traffic is above the level of the link layer. The lowest traffic format in this setup is the Ethernet packets.

Another important reason why we use this loopback structure is that it needs only one single FPGA to complete the tests. It is also much simpler to perform behavioural simulations using a loopback setup. During the simulation procedure for one Vitis kernel, the kernel is instantiated with its stream ports connected to standard interface analyzers and its host AXI interface driven by testbench functions.

For a complete kernel verification, this testbench can be very lengthy. If we need to simulate the normal case where two RoCE kernels are connected to each other, there should be two instances of the kernel. Thus, the host interfacing and the stream ports verification become more complex.

4.2. RoCE Kernel

Now that we have already seen the top level structure of the RoCE network stack on the Alveo platform, let's take a closer look at the core part: the RoCE kernel.

A more complex structure block diagram for it is shown in Figure 4.3, where the RoCE kernel is explained in more detail. The other parts remain the same as Figure 4.1. Apart from the four IP blocks, there are more rectangles in the background. They show the scope of the HDL files with their corresponding HDL file names on the top left corners of the rectangles. So that the hierarchical relationship between modules can be more easily understood. The thin arrows represent control or meta signals, while the bold arrows indicate data flows. The arrows here are all labeled with their real signal naming in the design files. So the figure shows the internal control and data flow clearly. However, the state machine and the operation modes are not indicated in the figure. They will be discussed shortly below.

This kernel works in the sequential mode as a Vitis kernel. Although in many cases the RoCE kernel can run without any host involvement, there are some necessary instructions such as an initial ARP request should be given by the host for proper functionality. The memory interface is also necessary for RDMA purposes. As Vitis do not allow free-running mode kernels to have a memory-mapped interface, it is now following the sequential mode. The clock frequency of the RoCE kernel is 250 MHz.

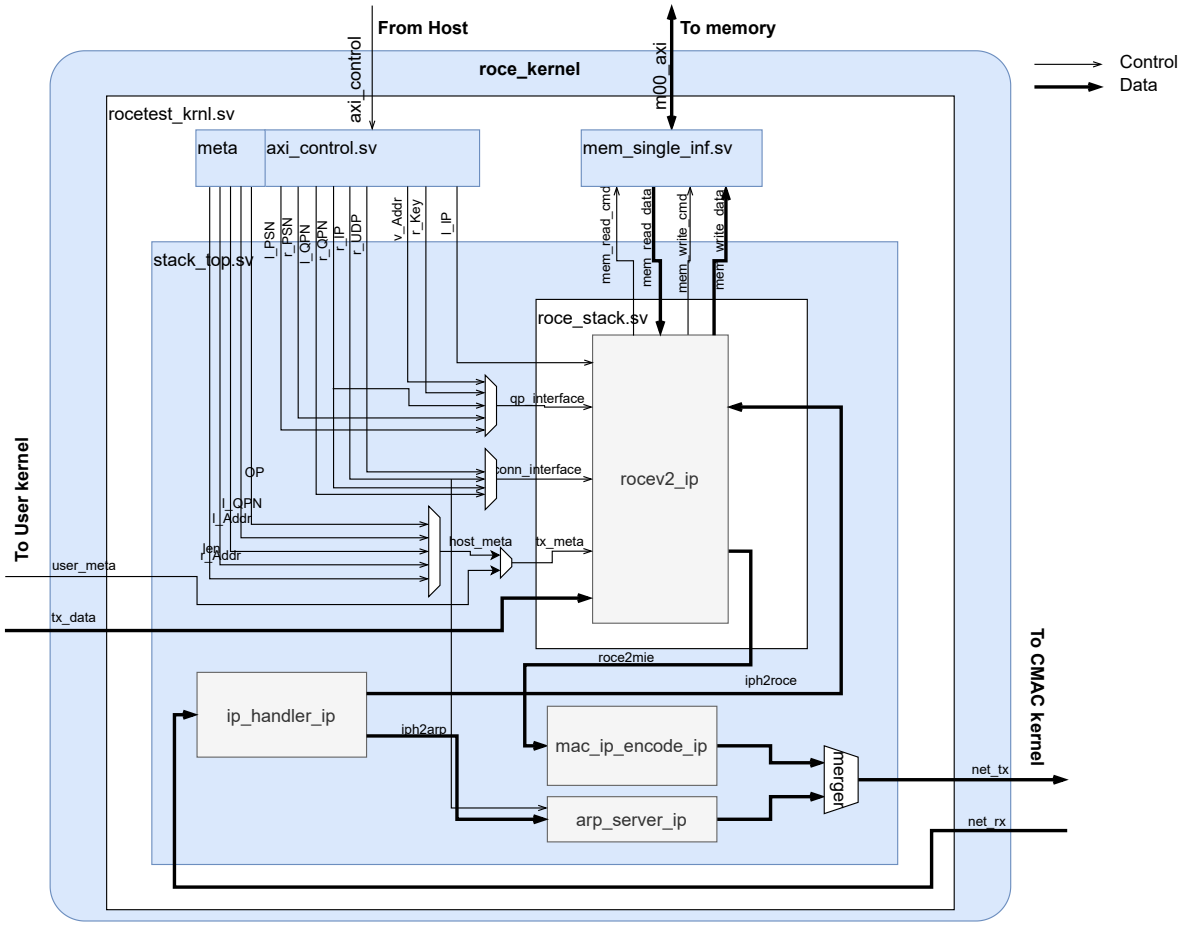


Figure 4.3: The block diagram and interfaces of designed RoCE kernel.

4.2.1. Kernel Hierarchy

As mentioned above, we used the Vitis design flow to develop the RoCE kernel. After defining the host interfaces, stream ports, and memory interfaces using Vitis RTL Kernel Wizard, the top-level module will be automatically generated with an HDL file. For the RoCE kernel it is the `rocetest_krnl.sv`. The control logic to pass host parameters to kernels is also generated along with it, which is the `axi_control.sv`. Whenever the host enqueues the RoCE kernel, parameters like the addressed and QP numbers will be transferred to the FPGA through an AXI Light interface. The `axi_control.sv` then extracts the parameters sequentially. These are all done before the kernel start signal is asserted for the RoCE kernel. So that the kernel is already able to use those parameters once the kernel is triggered to start.

The interface to read and write data to HBM is handled by the `mem_signle_inf.sv`. It is the same as what is used in [10]. The memory operations are handled in an aligned fashion. All the data blocks to be transmitted have a size of 64 bytes (i.e. 512 bits). This helps to improve the utility of the memory bandwidth. The memory transfer speed can degrade significantly if the data have small and floating lengths. The `mem_signle_inf.sv` also interfaces with the `rocev2` IP directly. Memory read or write commands lines and data ports are shown in the figure as well.

The `stack_top.sv` includes the core of our RoCE v2 stack. The four IPs are instantiated and connected in it. Almost all the control logic for the RoCE kernel functionalities is written in this file. Parameters are combined into meta signals for the IPs. State machines to control the operation modes are also developed here.

Finally, the `roce_stack.sv` is a simple wrapper for the `rocev2` IP. It controls whether the IP will be instantiated or not. It also parameterizes the data width for the `rocev2` IP.

4.2.2. Kernel Interfaces

As mentioned in Chapter 2, three types of interfaces can exist for each kernel. The RoCE kernel has them all. These interfaces are illustrated in the arrows in Figure 4.4. The name of each port is also shown on each arrow.

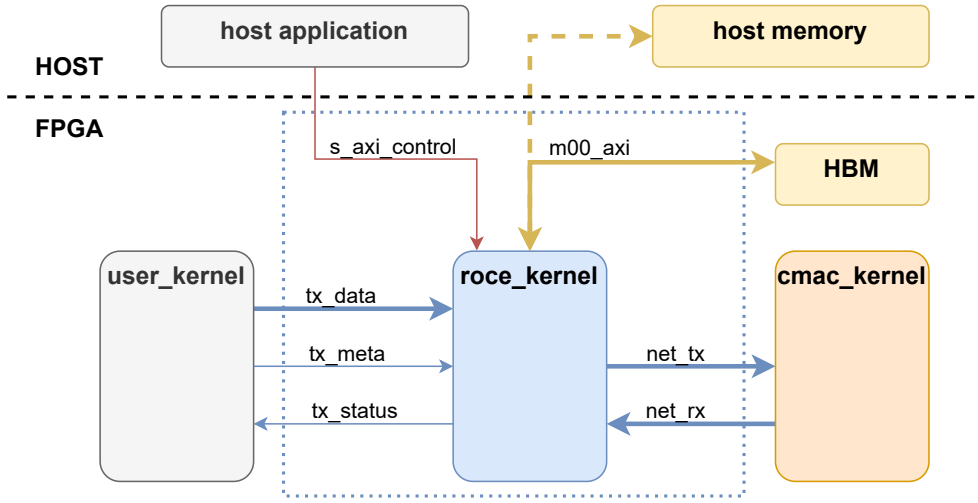


Figure 4.4: The interfaces for the RoCE kernel.

The register interface is needed to accept host arguments through the `s_axi_control` AXI Lite slave interface. It is shown as the red arrow. The details of these arguments can be found in Table 4.1. The first 8 arguments are parameters that define the QPs and RC connection. They are required for any operation modes. They should be exchanged between the two endpoints before any data transfer. Arguments with ID 9 to 12 are used for RDMA requests which are essentially the Working Queue Elements. Then there is a memory pointer `mem_ptr` pointing at the address of HBM. It is acquired by the host application when it allocates that memory.

Lastly, the `debug` argument is currently used to pass some control arguments to the RoCE kernel. The fields of `debug` is listed in Table 4.2. There are three operation modes for this RoCE kernel for different use cases and should be correctly set. More details will be discussed shortly. The board

ID	Name	Port	Size (bytes)	Type
0	rPSN	s_axi_control	0x4	uint
1	lPSN	s_axi_control	0x4	uint
2	rQPN	s_axi_control	0x4	uint
3	lQPN	s_axi_control	0x4	uint
4	rIP	s_axi_control	0x4	uint
5	lIP	s_axi_control	0x4	uint
6	rUDP	s_axi_control	0x4	uint
7	vAddr	s_axi_control	0x8	ulong
8	rKey	s_axi_control	0x4	uint
9	OP	s_axi_control	0x4	uint
10	rAddr	s_axi_control	0x8	ulong
11	lAddr	s_axi_control	0x8	ulong
12	len	s_axi_control	0x4	uint
13	debug	s_axi_control	0x4	uint
14	mem_ptr	m00_axi	0x8	int*

Table 4.1: Host argument parameters for the RoCE kernel.

Field	Width	Position	Description
Mode	2	[1:0]	The operation mode of this stack
Board number	2	[3:2]	The number to distinguish FPGA boards
Interval	12	[15:4]	The time interval between two RDMA requests, ranging from 0 to 4096 cycles
Reserved	16	[31:16]	Reserved for other purposes, not used so far

Table 4.2: Description for the `debug` field of the designed RoCE kernel.

number is used to distinguish FPGA boards for both user and the networking. MAC address should be unique for each network end node. The MAC address for this RoCE kernel is related to the board number to ensure no repeat. The interval is a field used only for test mode to avoid the situation where RDMA requests are issued too fast that the memory operations may have not finished. The unit is in cycles, so the interval is ranging from 0 to 4096 cycles. The other bits of the `debug` argument are not used and are left reserved for potential future use.

A memory-mapped interface is also used for the RoCE kernel to communicate with the HBM on FPGA. It is shown as the double-ended yellow arrow named `m00_axi`. It is an AXI4 memory-mapped Master interface. The HBM on Alveo U280 FPGAs is divided into 32 segments with 256 MB each. The segments are called pseudo channels (PCs). The bandwidth of a single PC is 14.375 GB which corresponds to 115 Gbps. This is already enough as it's higher than the targeted 100 Gbps networking bandwidth. Hence, our `m00_axi` is configured to be connected to one of the PCs, more specifically, the `HBM[0]`.

For the Alveo U250 platform, there is no HBM bank on FPGA. The memory interface should be connected to the `HOST[0]` to enable host memory access. Since we use the U280 FPGAs as default, the interface to host memory for U250 boards is shown by the dashed arrow.

The memory interface module we used is the same as that in EasyNet. It implements an Xilinx AXI DataMover to perform the conversion between the streaming interface and the memory-mapped interface. However, we find the memory write direction of this module could get stuck during evaluation. This is the reason why only a small amount of data can be successfully transmitted.

Finally, there are several streaming interfaces for the RoCE kernel to stream data from and to other kernels, as shown in the blue arrows in Figure 4.4. Two AXI Stream slaves ports are exposed by the RoCE kernel to the User kernel to receive the user's RDMA requests and data. An optional AXI Stream master port is connected to the User kernel as well for potential feedback from the RoCE stack. For example, the RoCE kernel can notify the user when package dropping or NAK is happening in the network stack. It is currently not used as we only used very straightforward User kernels for simple benchmarks. On the other side, the RoCE kernel exposes an AXI Stream master port and an AXI

Stream slave port to the CMAC kernel for the transmit and receive paths.

4.2.3. Operation Modes

The RoCE kernel has three operation modes, which can be selected by host provided parameter. The 2 lowest bits of parameter `debug` is used to indicate the operation mode.

In mode 0 what we called the **default mode**, the RoCE kernel will only be written with its queue pair parameters and connection information. No RDMA operation meta is passed to the RoCE kernel from the host so that it works passively. However, this mode does not mean that the kernel can only work as a responder. There can be also RDMA operation issued by the upper stream User kernel. The User kernel can contain any user application logic with any RDMA operations. It can also provide the data for RDMA WRITE operations. The RoCE kernel in mode 0 simply handles the incoming traffic as well as the User kernel commands or data. This is the mode that we recommend and should be used in most cases.

Mode 1 is a **test mode** where the RoCE kernel will sequentially execute RDMA READ, RDMA WRITE, and RDMA READ operations. A successful read-write-read test verifies both RDMA READ and RDMA WRITE operation functions. The target remote memory block is first read out, so we know if the read operation fetches the correct data. Then the same remote memory block is overwritten by the new data we specified in local memory with an RDMA WRITE operation. Finally, the last RDMA READ operation ensures that the remote memory block is indeed overwritten by the RDMA WRITE operation. Note that there should be a certain time interval between RDMA READ and RDMA WRITE operations to the same memory address. Otherwise, the control logic for memory access will hang before the last operation is finished. This might lead to a deadlock. The interval is configurable through the Interval field in Table 4.2.

Mode 2 is called the **host mode** where the RDMA operations are issued by the host. The User kernel shouldn't request any RDMA operation or send any data to the RoCE kernel to avoid conflicts. Instead, a dummy User kernel should be used in this mode. In this case, the host code will provide one OpCode for the RoCE kernel to execute. Because of the Alveo execution model, host parameters can only be written to the kernel when it's enqueued. So that only one RDMA operation can be issued at a time. To request more operations, multiple executions of the host code should be performed.

4.2.4. The `rocev2` IP

The internal structure of the `rocev2` IP can be seen in Figure 4.5. The red boxes frame the scope of each layer. We can see clearly that the RoCE v2 stack already includes the UDP and IP layers. The Reliable Connection transport layer service is implemented. The `rocev2` IP is actually a stack of `ib_transport_protocol` IP, `UDP` IP and `IP` IP. The internal details for it can be found in the Ph.D. thesis of Sidler [27] and will not be discussed here. What matters in our work is the interfaces of this `rocev2` IP and how it works.

On the bottom side, this IP communicates with the networking which is the Ethernet. Two 512-bit wide AXI Stream data ports are used for receiving and transmitting. Data that go through these ports are the Internet Protocol packets.

On the top side, the `rocev2` IP interfaces with memory and accepts the RDMA requests. The RDMA operations require many direct memory accesses. There are two 512-bit wide AXI stream data ports as well as two customized AXI meta ports (includes 64-bit wide address and 32-bit wide length fields) for memory read and write. In our kernel, these memory ports are connected to the `mem_signle_inf.sv` and then communicate with the HBM on FPGAs. As for the RDMA requests, they are wrapped as an `tx_meta` AXI Stream. The description and position of each field in the `tx_meta` is listed in Table 4.3. The width is in bits and the positions show which bits are used for each field.

The `OP` code mentioned here in the `tx_meta` is different from those OpCodes in Chapter 3. The `OP` here are abstracted into a higher level in the HLS code inside InfiniBand RC transport service. There are no separated `RDMA READ response First`, `RDMA READ response Middle` or `RDMA READ response Last` for a single RDMA READ operation. Only the general `RDMA READ` and the `RDMA WRITE` are used in this IP. Specific OpCodes for each packet are handled internally by the InfiniBand transport service IP and are opaque to the outside. This helps to simplify the user application or the User kernel design greatly. There are several other OP codes defined in this IP as well but we will not cover them. The `len` field is 32-bit wide, which corresponds to the message length ranging from 0 to 2 Gigabytes. With all the fields in `tx_meta` specified and accepted, the `ib_transport_protocol`

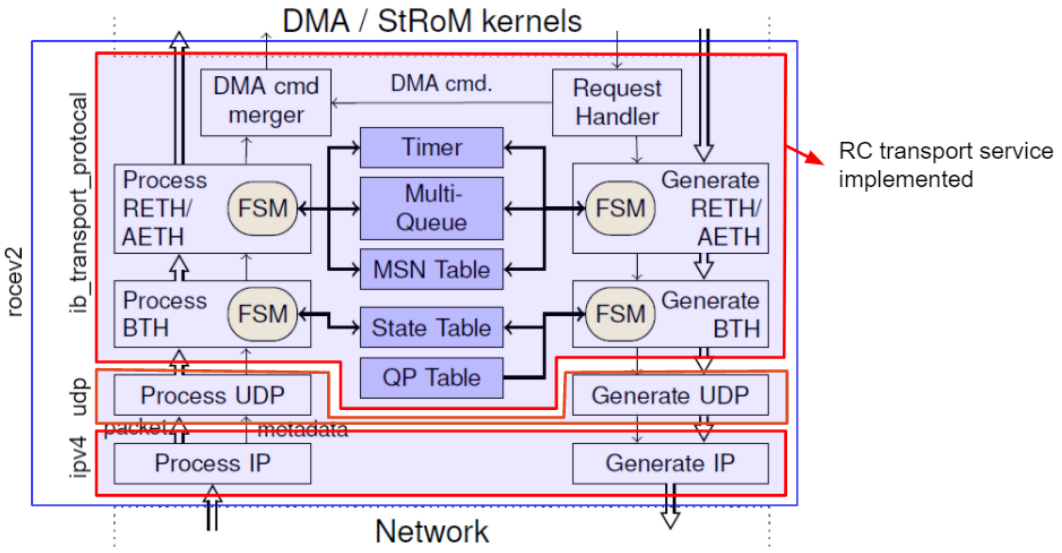


Figure 4.5: The internal structure for `rocev2` IP. Protocol header are processed in a pipelined fashion. Taken and modified from [29].

Field	Width	Position	Description
OP	3	[2:0]	Abstracted operation code: 0 - RDMA READ, 1 - RDMA WRITE, other not used
lQpn	24	[26:3]	The local QPN indicates which QP is used for this operation request
lAddr	48	[74:27]	The local virtual memory address to read data from or write data to
rAddr	48	[122:75]	The remote virtual memory address to read data from or write data to
len	32	[154:123]	The message length for this operation, ranging from 0 to 2^{31} bytes

Table 4.3: Description for the fields of the `tx_meta` AXI Stream for `rocev2` IP

IP will start generating the RDMA request packets as the flow on the right side of Figure 4.5.

Field	Width	Position	Description
state	3	[2:0]	Queue Pair states, only READY_RECV is used so it's always 3'b010
rQPN	24	[26:3]	The remote QPN indicating which QP is targeted
rPSN	24	[50:27]	The current remote PSN, used for the RC service to generate new PSN for each request
lPSN	24	[74:51]	The current local PSN, used to compare if a incoming RDMA packet is valid and in order
rKey	16	[90:75]	The remote protection key for the local end point to access remote memory
vAddr	48	[138:91]	The address that indicates where the virtual address locates locally, vAddr = 0 means local data are provided by `tx_data` stream rather than from memory

Table 4.4: Description for the fields of the `qp_interface` AXI Stream for `rocev2` IP

Other than that, information like the local and remote queue pair numbers and packet sequence numbers should be passed to the IP prior to any RDMA operation. They are fed into the InfiniBand transport service IP through two AXI Stream interface ports named as `qp_interface` and `conn_interface`. The parameters required to identify one QP are aggregated into `qp_interface`, while the parameters required to represent a connection are combined as the `conn_interface`. The detailed explanations for every field of them are listed in Table 4.4 and Table 4.5. Note that the total bit width needed is 155 bits for `tx_meta` and 139 bits for `qp_interface`. But the `tx_meta` is defined as 160 bits wide while `qp_interface` is 144-bit wide so as to become a multiple of 8 bits.

The `state` field of `qp_interface` tells the local QP what state it should be at. In total there are six internal QP states including `RESET`, `INIT`, `READY_RECV`, `READY_SEND`, `SQ_ERROR`, `ERROR`. However, only the `READY_RECV` is used in our design, which is sufficient for all cases. The `rPSN` and `lPSN` should coordinate very well so as to successfully executes RDMA operations. If the `rPSN` is written wrong, all the egress packets will carry the wrong PSN. Thus leading to all packets being either

dropped or unacknowledged by the remote responder.

Worth mentioning that the vAddr field controls whether the RDMA WRITE data is provided by a block of local virtual memory or through the tx_data AXI Stream. If the vAddr is set to 0, the local memory is ignored and the ib_transport_protocol IP will use the data streamed by the tx_data port. Otherwise the vAddr indicates the real address of the local memory which can be directly accessed by the ib_transport_protocol IP.

Field	Width	Position	Description
lQPN	16	[15:0]	The local QPN indicating which QP is targeted
rQPN	24	[39:16]	The remote QPN indicating which QP is connected to the local QP
rIP	128	[167:40]	The remote IP address, supports both IPv4 with 32 bits and IPv6 with 128 bits. Since IPv4 is applied, only [167:136] is used and other bits set to 0.
rUDP	16	[183:168]	The remote UDP port number

Table 4.5: Description for the fields of the conn_interface AXI Stream for rocev2 IP

For the conn_interface, all fields are necessary information to construct and identify one connection. The rIP field is 128-bit wide which supports both IPv4 and IPv6. Since in this project only the IPv4 is applied, only bits [167:136] are used. Bits [135:40] are set to 0.

The ib_transport_protocol IP is stacked over the UDP and the IP IPs to form the entire rocev2 IP. To conclude the input interfaces, three steps are needed for the rocev2 IP to issue an RDMA operation.

1. Provide the qp_interface to create local Queue Pair.
2. Provide the conn_interface to establish connection from one local QP to one remote QP.
3. Provide the tx_meta to execute an RDMA operation.

Note that the rocev2 IP has an extra mechanism to ensure data alignment. This helps to improve the memory throughput. The actual payload size of the rocev2 IP is limited to 1408 bytes rather than the 1460 bytes for a full MTU ($1500 - 20 - 8 - 12 = 1460$). Because the data width is 512 bits in most parts of the RoCE stack including the memory interface, the payload size is set to be multiple of 64 bytes. Thus, that there will not be unaligned data blocks. Unaligned memory operations can degrade the throughput significantly [10]. 1408 is the maximum payload size to ensure this.

The rocev2 IP is functioning but not an production-level implementation. There are still many functionalities to be added or optimized. Before developing the RoCE stack on FPGAs, we already investigated and tested this IP carefully. The work can be found in the Github repository [3].

Several bugs have been fixed. For example, we fixed the width parameters for several header fields. We also corrected the order of the local and remote UDP port parameters for the RDMA operation meta. In this IP, the UDP port is set to a default value equal to 0x12b7. However, this problem is crucial for two end nodes whose UDP ports are not the same.

Besides, we also provide TX and RX test cases with test packets. The test setups such as the addresses, QPNs and OpCodes are discussed in detail with all the information explained. Both RDMA READ and RDMA WRITE operations are included to perform a read-write-read test. The packets do not contain Ethernet headers but all other upper level headers such as IP, UDP, BTH, and RETH. Detailed explanations for all types of headers and a sample packet are provided in the README file.

4.2.5. TX/RX Path

First, let's look at the transmit and receive paths for ARP packets. As mentioned above, remote information needs to be mutually confirmed by the two RoCE v2 stack end nodes of a connection. The remote IP address is not only used by RDMA packets but also used to generate an ARP request. The remote MAC address should be acquired and stored to the arp_server IP before any RDMA packets can be sent out.

[3] https://github.com/hcxxstl/fpga-network-stack/tree/master/hls/rocev2

On the rising edge of the start signal for this RoCE kernel, an ARP request will be triggered immediately for this purpose. The `arp_server` IP then generates an ARP request packet containing the remote IP address to ask for its MAC address. The packet will go through a packet merger, which is a 2 in 1 out Xilinx AXI interconnect IP. And it will finally move forward to the CMAC kernel. After the remote end node has received the ARP request and responded, the ARP response packet will be received through the `net_rx` AXI Stream port. The `ip_handler` IP will then determine this packet should go to the ARP server rather than the RoCE stack. With the remote MAC address received, the ARP server will create an entry of an IP-MAC address pair. On the responder side, the ARP packets also go through the same RX or TX path. The only difference is that the ARP server will accept the ARP request and reply with an ARP response packet including its MAC address.

As shown in Figure 4.3 the TX path for RDMA packets is similar to ARP packets but only one more step. Packets generated by the `rocev2` IP will be first sent to the `mac_ip_encoder` IP. The encoder will ask for the `arp_server` what the remote MAC address is. The ARP server then searches in its mapping table to find the corresponding MAC address and reply to the encoder. `mac_ip_encoder` then insert the MAC address and forward this RDMA packet. After going through the merger, the packet will be sent out with the `net_tx` AXI Stream port. The RX path for RDMA packets is the same as that of ARP packets. Ingress packets will be diverted to the `rocev2` IP rather than the ARP server by the IP handler according to their packet headers.

As we have two modes to issue RDMA operations for the RoCE stack, another merger is used for the network stack. In the host mode, RDMA operation parameters provided by the host are combined into the `host_meta`. While in the default mode, RDMA meta is provided by the User kernel through `user_meta`. The `host_meta` and `user_meta` are merged as the `tx_meta` for our `rocev2` IP.

4.3. User and CMAC Kernel

4.3.1. User Kernel

The User kernel is the kernel where user-defined applications are. It is the upper stream kernel that issues RDMA commands to the RoCE kernel. It can be designed for any application that needs RDMA networking. The interfaces of the User kernel is shown in Figure 4.6.

This kernel also works in sequential execution mode as it should be triggered by the host program. It is clocked at 250 MHz in order to coordinate with the RoCE kernel.

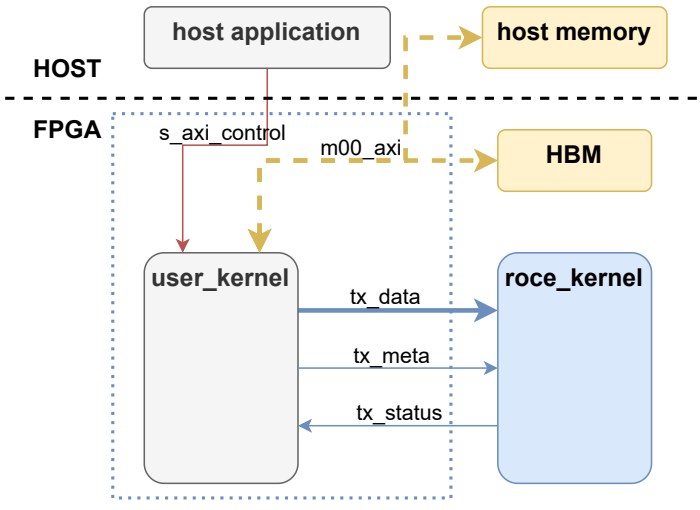


Figure 4.6: The interfaces for the User kernel.

Arguments can be passed to the User kernel through a register interface by the host application. The User kernel can issue RDMA commands by sending operation meta through the `tx_meta` AXI interface. It is 256-bit wide to comply with the Vitis kernel stream interface restriction. So that only the lower bits are used. The meta should comply with the structure stated in Table 4.3. Note that the meta includes the local QPN which is part of the QP context. This local QPN should be given by the host

application when it is established by the host. The `tx_data` AXI interface is only used when `vAddr` is set to 0. The RDMA WRITE operation can then use this data stream provided by the User kernel as the data source. There is another extra AXI Stream slave interface to the RoCE kernel, which is the `tx_status`. This is an optional interface that is used to acquire internal values of the RoCE kernel when needed. This is not used in this thesis project but is open for any usage. It could be useful for some applications that need feedback from the networking stack.

Besides, the User kernel can of course create a memory interface to FPGA global memory like the HBM or to host memory. They are illustrated by the yellow dashed arrows. This means the host data can be pre-processed by the User kernel before being sent to the RoCE network stack. Applications whose data processing can be offloaded to FPGAs will benefit from this. This is known as in-networking processing. But this is only viable for RDMA WRITE operation. If we want the ability of processing data before they are written to host memory for the RDMA READ operation, a memory interface should be added between the RoCE kernel and the User kernel. This can be a future work to improve the kernel design.

Three User kernels have been developed to demonstrate and test this RoCE network stack.

As mentioned previously, a dummy kernel should be used to connect the RoCE kernel when its operation mode is host mode. It's important to correctly de-assert the valid signals for both `tx_data` and `tx_meta` AXI signals to avoid metastability. The developed **roce_dummy_krnl** does this job. It is very simple with no logic but only these wire assertions.

A **roce_read_krnl** is developed to iteratively generate RDMA READ commands for the RoCE kernel. It is used as the RDMA READ benchmark test kernel. A 32-bit host argument `debug` is used to pass necessary parameters to control the RoCE read kernel. The fields for `debug` are explained in Table 4.6.

Field	Width	Position	Description
lQPN	24	[23:0]	The local QPN indicating which QP is used for this operation
len	5	[28:24]	The message length of one RDMA operation
cnt	3	[31:29]	The total number of RDMA operations to be executed

Table 4.6: Description for the `debug` field of the designed RoCE read or write User kernel.

The lowest 24 bits are reserved for the local QPN necessary for the RDMA operation meta. The `len` field takes up 5 bits and tells the read kernel how many bytes of data should be read in one RDMA READ operation. The actual message length is calculated as 2^{len} bytes. So the supported message length is from 1 to 2^{31} bytes (2 GB). The highest 3 bits are used as the `cnt` field. It determines the number of RDMA operations be executed. The actual number is calculated as 2^{cnt}. The supported number is ranging from 1 to 128. In combination, the total message size to be read from the remote memory is up to 256 GB per User kernel execution.

Because this RoCE read kernel is only developed for benchmarking, there is no other host arguments designed. The `lAddr` and `rAddr` are handled by the kernel itself. `rAddr` is always set to zero so that the same block of memory is read repeatedly. While the `lAddr` is increased by the message length iteratively. Thus, this RoCE read User kernel will read the same message on the remote end node and store them to the local memory successively.

A **roce_write_krnl** is designed in the same way. The same `debug` argument is used to pass the host parameter to the User kernel. The only difference is that the operation code, the local addresses, and remote addresses are set reversely.

4.3.2. CMAC Kernel

With the User kernel and RoCE kernel, the RoCE v2 network stack is already functioning. The last step for it to connect to the real Ethernet fabric is the CMAC kernel. The CMAC kernel we used is the same as what in the TCP/IP stack project [10]. So the details will not be discussed in this thesis.

The CMAC kernel works as the free-running execution mode with no host arguments. It has only two interfaces for TX and RX path which are 512-bit wide AXI Stream ports. Another thing that should be understood is that this CMAC kernel uses the Xilinx UltraScale+ Integrated 100G Ethernet Subsystem

IP as its core. A license of this IP is required to generate the bitstream for this entire RoCE stack design. This IP works at a frequency of 322 MHz. With Clock Domain Crossing (CDC) implemented internally, the CMAC kernel can be easily connected to the RoCE kernel clocking at 250 MHz.

The CMAC kernel is explicitly assigned to SLR2 on Alveo boards. As mentioned in Section 2.3, this means that the MAC layer is placed closer to the physical network ports. This configuration is important to avoid timing problems.

5 Evaluation

In this chapter, evaluation for the designed RoCE v2 network stack for FPGAs will be presented and discussed. This chapter is organized as follows. First, the XACC cluster is explained briefly. Then the benchmark setup and results will be presented in detail. Finally, the evaluation results for our RoCE network stack are compared with the TCP/IP network stack.

5.1. Evaluation Platform

The evaluations for our RoCE v2 network stack are performed on the Xilinx Adaptive Compute Clusters at ETH Zurich. As shown in the Figure 5.1, the cluster has a well-configured networking fabric.

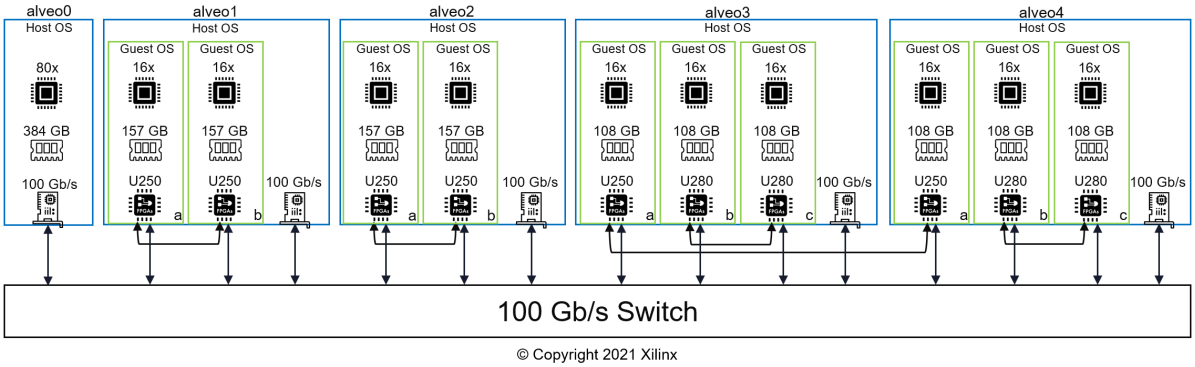
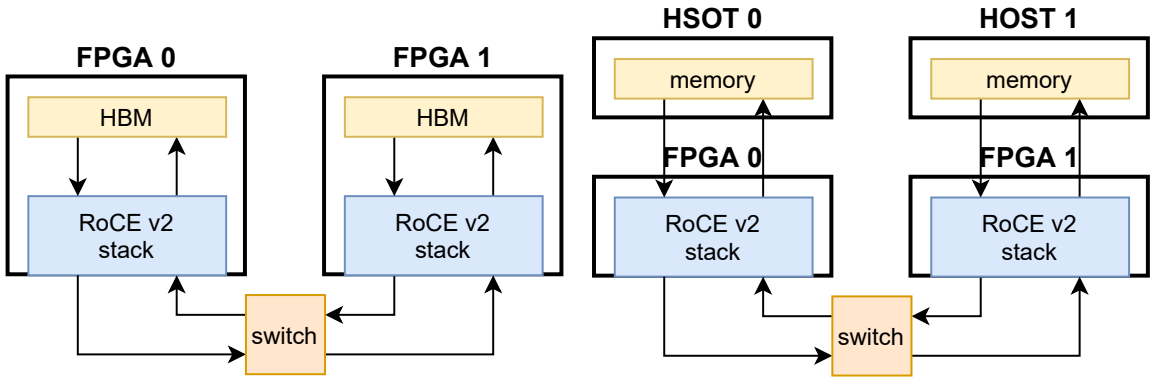


Figure 5.1: The networking configuration for XACC cluster at ETH Zurich. Taken from XACC documentation page. Taken from[1].

There is one develop node `alveo0` and four deployment nodes `alveo1 - 4`. The develop node is only used for developing and running implementations. It does not have any FPGA attached to it. Each deployment node has multiple Alveo FPGAs connected. Each FPGA is handled by a separate operating system. The tools for the Vitis shell are only enabled on the deployment nodes.

As for the networking infrastructures, all the hosts and FPGAs are connected to the same 100 Gbps Ethernet switch. So it is very convenient for the developers to test the network stack on any device. For example, we can easily perform FPGA to FPGA or FPGA to host communication tests. Besides that, two FPGAs on the same deployment node are connected directly without a switch in between. This is useful for local FPGA to FPAG communication that does not need to go to the Internet and requires ultra-low latency. However, we did not use this setup. Because we consider that there are only limited use cases where two RDMA endpoints are always connected to each other. This can be beneficial only for applications that have an unchanging data path. It also makes it impossible for applications to have data partitioned in multiple nodes since no data shuffle can be done without a connection to a centralized network switch. In a more common computing cluster, computing nodes are connected to network switches rather than connected in pairs for better flexibility.

[1] `https://xilinx.github.io/xacc/ethz.html`



(a) The test setup for Alveo U280 FPGAs with HBM as memory.

(b) The test setup for Alveo U250 FPGAs with host memory access.

Figure 5.2: The schematic diagrams of the board to board test setup.

In our evaluation, the RoCE network stack is deployed on the Alveo U280 FPGAs. We test it on the `alveo3b/c` and `alevo4b/c` nodes. The schematic diagram for our tests setup is illustrated in Figure 5.2a. Two Alveo U280 FPGAs are programmed with the RoCE network stack to work as the two RDMA endpoints. They are connected with 100 Gbps Ethernet fabric with a switch in between.

A similar evaluation could also be performed for U250 FPGAs with host memory access as shown in Figure 5.2b. For the Alveo U250 setup with host memory access, we have only verified the functionality on the `alveo2a/b` nodes but have not performed any evaluation for it.

5.2. RDMA Benchmark

Before performing the benchmarks, the correctness of the designed network stack should be guaranteed. Because it requires many efforts to debug the implementation on FPGAs, we performed validation step by step throughout the development procedure as explained at the beginning of Chapter 4.

	Host 0	**Host 1**
User kernel	read	dummy
Board number	0	1
RoCE kernenl mode	0	0
rPSN	`200000`	`000000`
lPSN	`000000`	`200000`
rQPN	`100000`	`000000`
lQPN	`000000`	`100000`
rIP	`0b01d4e1`	`0b01d4e0`
lIP	`0b01d4e0`	`0b01d4e1`
rUDP	`12b7`	`12b7`
vAddr	`00000001`	`00000001`
rKey	`0000`	`0000`

Table 5.1: The necessary parameters that used to establish RDMA RC model connection for our benchmark tests.

The necessary parameters that are used to establish RDMA connection are listed in Table 5.1. The values of these parameters that we used in our benchmark for the two FPGA end nodes are also provided. Because these arguments are assigned and passed to FPGAs by the host executable, they are listed under Host 0 and Host 1. Note that the parameter values in the lowest section are all in hexadecimal format as they are written in monospace font. The IP addresses are `10.1.212.224` and `10.1.212.225` in normal format. They are set as such to avoid having conflicts with any existing machine while being within the subnet of the XACC ETH Zurich cluster.

As discussed previously, these parameters in Table 5.1 should be exchanged between the two

RDMA endpoints using other communication methods before the RDMA connection can be established. In our test, the pre-exchange of information is neglected and they are assumed to be known by the two hosts. Thus, a pair of two host codes are developed with each representing one endpoint. The local and remote addresses, PSNs, and QPNs are already written accordingly in the host files to simplify the test.

After setting up the RDMA connection with these parameters, RDMA operations can be performed between the two FPGA endpoints, namely FPGA 0 and FPGA 1 as illustrated in Figure 5.2. The two boards are distinguished by the board number argument. This board number is also used to generate different MAC addresses for each FPGA hardware. The operation modes of the RoCE kernel on both FPGAs are the default mode so that RDMA operations can be issued by the User kernel.

The FPGA 0 is used as the local requester with RoCE read User kernel triggering the RDMA commands. It uses either a READ or a WRITE User kernel. While the FPGA 1 works as the remote responder with a dummy User kernel connected. In both of the RDMA READ and WRITE benchmark, only one RC connection is used. This means only one pair of QP is set up for the two end nodes. The message size is varied from 64 B to 4 MB to evaluate the throughput for our RoCE stack.

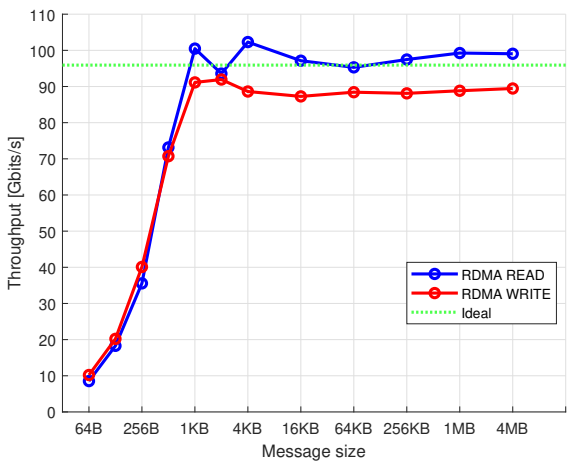
5.2.1. Throughput

One of the most important metrics for a network stack is throughput. Our RoCE v2 network stack on FPGAs is targeting 100 Gbps bandwidth. Since the `rocev2` IP does not provide the internal state to its interface, the kernel itself can't determine when the RDMA operations are finished. Thus, the throughput is measured on FPGA boards through Xilinx ILA IPs. Clocks are inserted into the RoCE kernel so that the total data transmission time can be measured by calculating the cycle count difference. Because the measurement process is quite cumbersome, each test is only performed twice. The value on the figure shows the average of two tests.

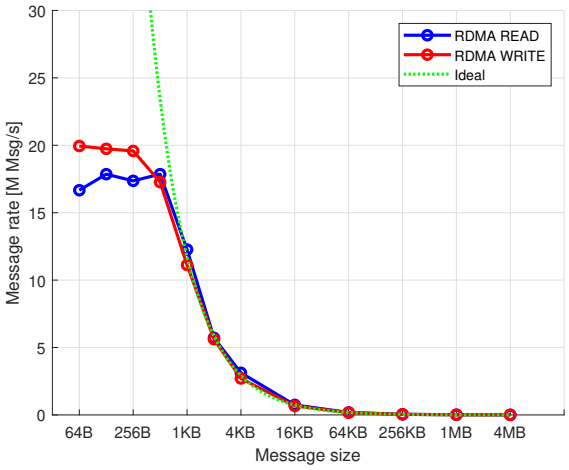
$$\text{Throughput} = \frac{8 \cdot Len \cdot Cnt \cdot f}{Cycle} \tag{5.1}$$

$$\text{Message Rate} = \frac{Cnt \cdot f}{Cycle} \tag{5.2}$$

The throughput is calculated by Equation (5.1). The Len is the same as the Message length used in the RDMA meta. It is in bytes, so a factor of 8 is multiplied to convert to the bits unit for throughput. The Cnt is the total number of messages to be read, which is also a parameter in Table 4.2 for the User kernel. The $Cycle$ stands for the total cycle count for the entire RDMA READ or WRITE operation. And the f represents the frequency of this RoCE kernel, which is 250 MHz. Message rate is another metric for this benchmark that is related to the throughput. It shows how many messages the network stack can transfer in one second. It is determined by Equation (5.2).



(a) Throughput of RDMA READ benchmark.



(b) Message rate of RDMA READ benchmark.

Figure 5.3: The throughput and message rate of the RDMA READ benchmark.

The benchmark results are shown in Figure 5.3. The left side Figure 5.3a shows the throughput while the right side shows the message rate. Note that the unit for message rate is a million messages per second. The throughput in the figure is actually the goodput, which counts only the payload without any header overhead. The red dash lines in both figures indicate the ideal case considering an MTU size of 1500 for a 100 Gbps network fabric. In this ideal case, the Ethernet packet has a length of 1518 bytes including the Mac header and CRC. Excluding the inevitable IP, UDP, BTH headers, and the ICRC field, there are 1456 bytes left for the payload. The ideal goodput for the RoCE v2 structure is thus 95.9 Gbps on 100 Gbps network as calculated in Equation (5.3).

$$\text{Ideal Goodput} = 100Gbps \cdot \frac{1500 - 20 - 8 - 12 - 4}{1500 + 14 + 4} = 95.9Gbps \tag{5.3}$$

From the throughput result, we can see that the designed RoCE v2 stack can achieve the full bandwidth when the message size is larger than 1 KB. When message size is smaller the throughput is significantly lower. A straightforward explanation for this bandwidth degradation would be due to the packet header overhead. When the message size is smaller than the length of a single Ethernet packet, the percentage of header overhead will become larger in the network. But this is not the case. For example, the header overhead contributes at most 50.7% for an RDMA READ response packet with 64 bytes payload. While the actual goodput is lower than 10% of the network capacity. This packet header overhead surely plays a significant role to decrease the throughput for smaller message sizes, the message rate is the direct reason for through reduction. For message sizes smaller than 512 bytes, the throughput curve follows a quadratic pattern. This means a linear relationship between the throughput and message size as the horizontal axis is logarithmic. A similar conclusion can be drawn from the message rate as well in Figure 5.3b. When message sizes are smaller than 512 bytes, the message rate remains steady around 17.5 million messages per second for READ and 20 million messages per second for WRITE operation. This corresponds to 13 - 14 cycles for each message (or each packet, since message sizes are small to fit in one packet).

Another important question arises here on what is the limiting factor of this message rate. This is not a limitation of the packet processing speed of the `rocev2` IP because the IP can process each packet in 5 cycles [29]. In his work, the message rate is restricted by the speed of issuing RDMA commands by the host. While we use the User kernel rather than the host to generate RDMA operation commands. The RoCE read kernel in our design can generate one RDMA command every 7 cycles. The real restriction in our work is introduced by the memory interface. It can only consume one memory operation about every 14 cycles. The reason behind this remains unclear and needs further investigation for the memory interface. As the message size grows over 1 KB, the message rate restriction will be taken over by the network bandwidth.

5.2.2. Latency

Besides the throughput, latency performance is also vital for network stacks. The connection establishment is out of the scope of RoCE architecture. What we are interested in is the actual delay of data flowing through the network stack. The network fabric outside our RoCE stack is not the focus.

The total RDMA READ delay is counted in cycles from the cycle when an operation is issued by the User kernel through `user_meta` until the final byte of the data has crossed the interface between the RoCE kernel and the HBM. The total RDMA WRITE delay is counted from the cycle when an operation is issued to the cycle when the (last) ACK packet is consumed by the RoCE stack. The results can be fond in Figure 5.4.

As we can see in the figure, the round trip latency is around 4 - 4.5 microseconds for our RDMA READ operations and about 3.5 - 4 microseconds for the RDMA WRITE operations. These results are also the average value of two tests for each message size. The latency difference for message size between 64 B and 1 KB is not significant. This is because that only one packet is needed for each operation. The same amount of cycles are spent to process them no matter the packet length. As message size grow larger than one packet size, the latency is approximately linear to the number of packets. For each additional package, there will be an additional 25 cycles for READ operations and 40 cycles for WRITE operations.

Other than that, we also measured the latency for each internal component. It is measured by the Vivado Hardware Manager with the ILA inside the RoCE kernel. For the RDMA READ operation, the

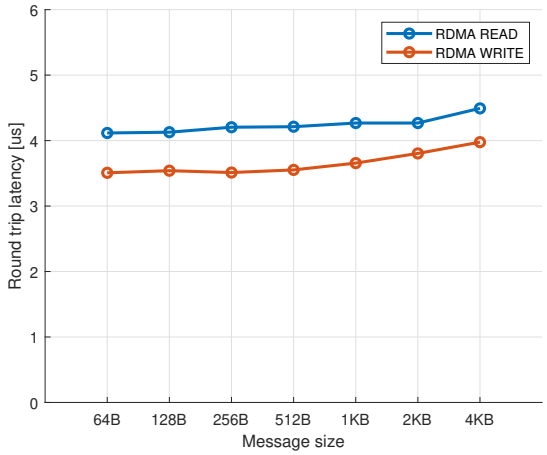
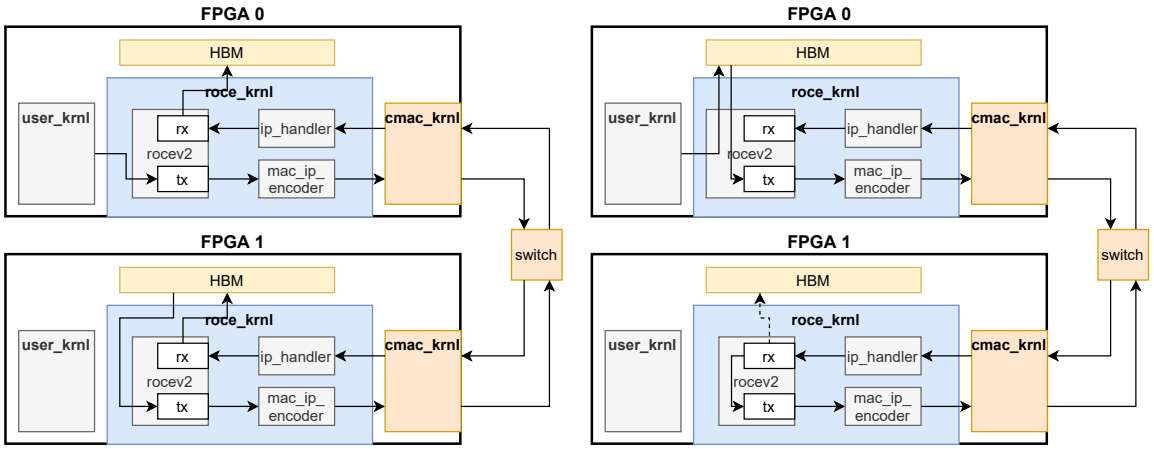


Figure 5.4: The latency of the RDMA READ and WRITE benchmarks.



(a) The entire path of the RDMA READ operation to be measured for latency.

(b) The entire path of the RDMA WRITE operation to be measured for latency.

Figure 5.5: The comparison for the throughput and message rate to related work.

latency of the full path is measured as illustrated in Figure 5.5a. For the RDMA WRITE operation, it is measured as shown in Figure 5.5b.

For the READ operation, the full path starts from the User kernel. First, the RDMA operation meta is sent to the RoCE kernel. After it is processed by the RoCE TX path, the RDMA READ request packet is generated and is sent to the MAC encoder to become a complete Ethernet packet. The RDMA READ request then walks through the CAMC kernel and network fabric and reaches the remote FPGA. After being processed by the IP handler and RoCE RX path, a memory read request will be sent to the remote HBM. Once the remote data is read out to the RoCE kernel, it will be packaged by the RoCE TX path as well as the MAC encoder into Ethernet packets. These RDMA READ response packets then travel through the switch to the local FPGA. After being processed by the IP handler and RoCE RX path, the data is finally be written to the HBM on the local FPGA.

The data path for the RDMA WRITE operation is similar to the READ operation except for the memory parts. On the local side, HBM is first accessed to acquire the data to be written. Then the WRTIE packet is generated and sent to the other FPGA. On the remote side, the delay of memory writing is not included. So the arrow is drawn in the dashed line. This is because the delay introduced by remote memory does not account for our round trip. The remote FPGA then generates ACK package and sends it back. The path ends when ACK is acquired by the local `rocev2` IP.

Each of the steps is carefully measured through the waveform captured by the Vivado Hardware Manager. The delay distribution of each part in the entire read path is shown in Figure 5.6. As the measurement is very cumbersome, two representative tests are performed. One reads 64 bytes and

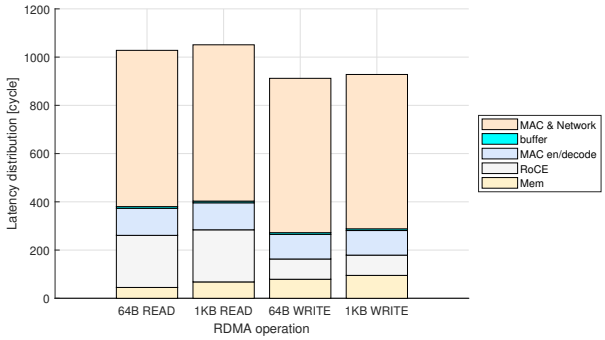


Figure 5.6: The delay distribution of each part in the entire data path for the RDMA READ operation.

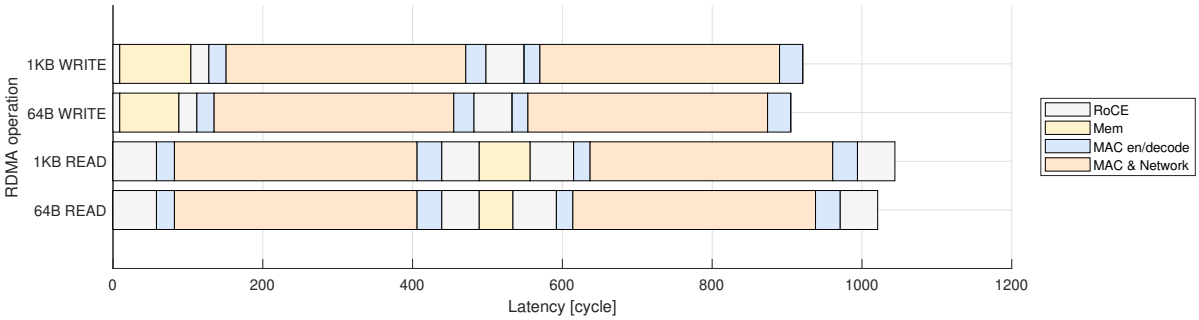


Figure 5.7: The delay timeline for the RDMA READ and WRITE operation on 64B and 1KB data.

the other uses 1 KB message size. Note that the sum of all parts of this delay distribution does not exactly match the total delay in Figure 5.4. The delay here is measured by comparing the time when the first packet arrives at each point. The delay for each module is determined by the difference between the cycle when the first rising edge arrives at the module and when it arrives at the next module.

Starting from the bottom, the memory read operations take up 45 cycles for 64 bytes data and 68 cycles for 1 KB data. Here we only count in the memory read because we cannot measure the memory write delay. For the RoCE processing latency, both TX and RX paths are accounted for. The total latency for this module remains steady around 216 cycles for READ operations and 84 cycles for WRITE operations. This means the RoCE stack itself (excludes memory accesses) introduces 0.864 us latency for READ and 0.336 us for WRITE operations. Because the `rocev2` IP is implemented in a fully pipelined design, message size does not affect this processing delay. Similarly, the MAC address encoding and the IP decoding have a combined latency of 110 cycles that is almost constant. And several buffers are contributing to 7 cycles of delay in total. Finally, the largest amount of delay is spent on the physical cable, switch, and the MAC layer. The measured delay for these parts is around 650 cycles, which is about 2.6 us. The only part where the difference appears is the memory read and write procedures. The delay increases approximately linearly as the message length grows.

We also plot the delay timeline for these tests in Figure 5.7. All of the four stacked bars can be seen as divided into five segments. From left to right, they are local FPGA, network, remote FPGA, network, and local FPGA.

For RDMA WRITE operations, the local FPGA spends most of its time accessing data from memory. After data is transferred to the remote FPGA, it generates the ACK packets within 100 cycles. Finally, the ACK is processed by the local FPGA. For RDMA READ operations, the local FPGA directly generates READ request packets without memory access. So the delay is very small at the local FPGA side. Then, the remote FPGA has to read data from memory and send them back, which takes up more than 200 cycles for 64 B data. In the end, the response packets travel through the network and are processed by the local FPGA.

5.2.3. Resources

The resource utilization for our RoCE v2 network stack on Alveo U280 boards can be seen in Table 5.2. The resource used by each kernel is presented. The total resources on a U280 FPGA are also listed

for comparison.

	LUTs		Registers		BRAMs	
Total	1302720	100.00%	2605440	100.00%	2016	100.00%
Full	212694	16.33%	388560	14.91%	698	34.62%
READ/WRITE User kernel	4203	0.32%	6346	0.24%	11.5	0.57%
Dummy User kernel	86	0.01%	121	0.00%	0	0.00%
RoCE kernel	92424	7.09%	178970	6.87%	459	22.77%
CMAC kernel	13418	1.03%	53313	2.05%	25.5	1.26%

Table 5.2: The resource utilization for our RoCE v2 network stack on Alveo U280.

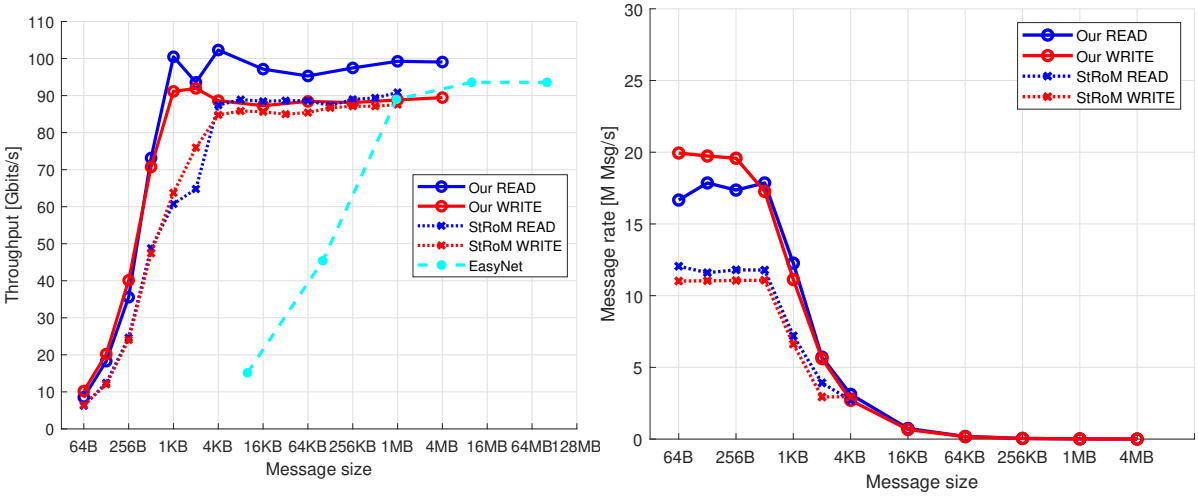
A full implementation including the READ User kernel, the RoCE kernel, and the CMAC kernel as well as the necessary peripheral logic occupies less than 20% CLB resources (LUTs and Registers) and less than 35% block memory (BRAMs). Note that the full utilization includes not only the kernels in the dynamic region but also the resources in the static region. The RoCE kernel took up the most percentage in all three types of resources. The small footprint of the RoCE kernel and CMAC kernel allows a large space for the User kernel. Very complex user applications can be fit into the rest of Alveo FPGAs.

Since utilization is not the focus of our work. No optimization is applied such as the implementation strategies provided by Vivado. Hence, the resource utilization could be even better.

Two examples of the Alveo device utilisation for this RoCE stack can be found in Appendix A. The design takes up less than one third of the dynamic region of Alveo U280.

5.3. Comparison to Related Work

In this section, a brief comparison between our designed RoCE v2 stack and some of the related work targeting at 100 Gbps network stack will be presented. As mentioned above, the 100 Gbps TCP/IP stack of EasyNet [10] is a starting point of our project. We did the development for the same FPGA platform and tested them on the same XACC cluster. It would be interesting to have a comparison between TCP/IP stack and RoCE network stack. Besides, David Sidler [29] also presented their measurements for the StRoM RDMA NIC (Network Interface Controller) running at 100 Gbps. We used the same HLS IPs but target different FPGA platforms. Their results are also compared.



(a) Throughput of our work, StRoM and EasyNet.

(b) Message rate of our work and StRoM.

Figure 5.8: The comparison for the throughput and message rate to related work.

Throughput

The comparison of throughput for our work, the StRoM RoCE stack, and the EasyNet TCP/IP stack is shown in Figure 5.8a. We can find that with smaller message sizes, the RoCE stacks have significantly higher throughput than the TCP/IP stack. Our work approaches 100 Gbps with a message size equal

to 1 KB. The StRoM reaches 100 Gbps with a 4 KB message size. While 1 MB data size is required for the TCP/IP stack to saturate the 100 Gbps bandwidth. Comparing our RoCE stack to the StRoM, the throughput is always higher. For smaller message sizes, this is due to the higher RDMA operation issuing rate. In our design, the operation is issued by the User kernel on FPGA rather than by the host CPU. This higher issuing rate means a higher message rate, which leads to better throughput results. For larger message sizes, the difference is possibly due to two reasons. First, the small amount of test data and few repeated tests in our measurement result in insufficient confidence in our results. Second, the DMA bandwidth in StRoM is almost the same as network bandwidth. This means that the memory interface is also a limiting factor for the stack. While there is limited data being accessed in our measurements, the impact from the memory interface is not shown.

The Figure 5.8b shows the comparison of message rate between our stack and the StRoM. A higher message rate is achieved in our work because the RDMA requests are generated by our User kernel on FPGAs, which is very fast. While for StRoM, the requests are issued by the host through the PCIe interface, which limits the message rate. As message size increases, the message rate for both implementations is restricted by the network bandwidth.

Latency

The latency of RDMA READ and WRITE operations are compared between StRoM and our work. As shown in Figure 5.9, the total latency for our RoCE stack on Alveo FPGAs is almost the same as the StRoM NIC.

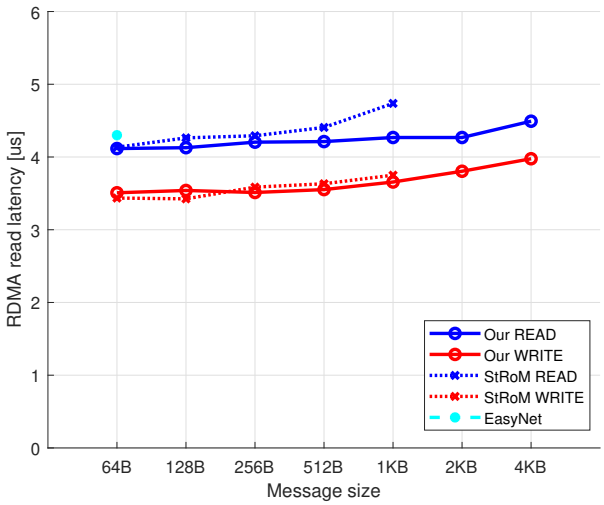


Figure 5.9: The comparison for the latency to related work.

However, this does not mean the latency is the same. First, there is no switch between the two endpoints of the StRoM evaluation. FPGAs are connected directly in his evaluation setup. To have a better comparison, we performed extra delay tests to make up for the difference. The `alveo4a` and `alveo4b` are used with their QSFP port 1 hooked to the CMAC kernel. So that we can test our RoCE stack between direct connected FPGAs. The results show that the switch in the ETH Zurich XACC cluster introduces 0.84 microseconds delay on average for one way. The RDMA operation data path includes Ethernet switch involvement twice. This means there is about 1.68 us or 37% of the 4.5 us delay is spent on the Ethernet switch.

Second, the memory read and write latency in our evaluation did not include the time spent in HBM. We only measured the delay until the last byte has passed the memory interface. The time from data leaves RoCE kernel to it is written to the HBM is unknown in our evaluations. The delay of memory access on the remote response side in 10 Gbps StRoM NIC [27] is around 1.4 us. Since its clock frequency is at 156.25 MHz, we should convert it to the cycle count which is about 219. This is significantly higher than in our stack. It is reasonable because we use on-board HBM rather than the host memory. Besides that, the ICRC inside the `rocev2` IP is ignored in our implementation. While it is enabled in StRoM. The generation and validation for ICRCs also contribute to several cycles of delay. The latency differences in memory interface and CRC calculation should explain the extra 1.68

us delay for the StRoM.

Although there is no TCP `receive` latency measurement provided in EasyNet, there is an RTT (Round Trip Time) measured for a ping-pong test carrying 64 bytes of data. The RTT between two TCP endpoints on FPGAs is 4.3 us [10]. It is shown as the only green point in Figure 5.9. Comparing the latency of 64-byte data transfer, our RoCE stack achieves 8.3% less delay than the TCP/IP stack. This advantage is not significant because the TCP/IP stack is entirely offloaded to FPGAs. With data provided by the user kernel on FPGAs, there is no memory copy overhead in the host CPU. The advantage of RDMA is thus not shown up. If we perform a benchmark for RDMA stack and TCP/IP stack with data provided by the host application, the latency difference should become larger.

6

Conclusion and Future Work

6.1. Conclusion

With the rapid development of big data and cloud computing, there is a growing demand for high bandwidth and low latency networking. Evolving technologies like RDMA have become popular among data centers. In this thesis project, we focused on RDMA network stack implementation for FPGA hardware. We have explored the three research questions mentioned in Chapter 1 and provided the following to answer them.

1. We investigated the alternative solutions for RDMA network stack for FPGAs and developed a 100 Gbps RoCE v2 stack on Alveo FPGAs.
2. We developed benchmark kernels and measured throughput and latency for the designed RDMA stack on FPGAs.
3. We made a comparison between our designed RDMA stack and existing RDMA and TCP/IP stack implementation in terms of throughput and latency.

First, through a detailed comparison of different RDMA architectures, transport layer models, a preferred configuration for the RDMA stack is selected for FPGA implementation. In this work, we showed that the RC transport service is most suitable for a general-purpose network stack in terms of its functionality and reliability. As for the architecture, RoCE v2 introduces the least packet header overhead and requires relatively little deployment effort compared to InfiniBand, RoCE v1, and iWARP. A comprehensive packet header fields comparison is carried out for several types of RDMA packets between each architecture. This is to our knowledge the most accurate header comparison for RDMA packets. Based on the alternative analysis, we developed an open source RoCE v2 network stack using existing IPs in the Xilinx Vitis flow. The RoCE stack can include a customized User kernel for user applications on FPGAs. The stack itself occupies only around 10% of the total resources on U280 FPGAs, which allows very complex user logic to be developed.

Then regarding the last two questions, the designed RoCE stack is evaluated and compared with related work. The design is deployed and evaluated on Alveo U280 FPGAs on the XACC cluster where 100 Gbps networking is enabled. With limited test data, the designed RDMA stack achieves 100 Gbps throughput when the message size is larger than 1 KB. A significant throughput advantage for RDMA stack over TCP/IP stack can be observed for data size smaller than 1 MB. The latency between two FPGAs with a switch in between is around 4 us for RDMA READ operation and 3.5 us for RDMA WRITE operation on 64 bytes data, which are both lower than the RTT of the TCP/IP stack.

6.2. Future Work

The designed RoCE v2 network stack is now functioning but a lot of further optimizations or functionality enhancements can be done in the future. Some of the directions we have considered are as follows.

Host memory access evaluation
The RoCE kernel currently has direct access to the HBM which is on FPGA resources. This configuration can not show the advantage of RDMA networking over other protocols as there is no real host memory access. To enable direct memory access to the host memory, the memory interface should be changed. We have found Several possible solutions for it. First, the host memory access capability provided by Xilinx through PCIe Slave Bridge[1]. This Slave Bridge exposes an AXI4 memory-mapped interface `HOST[0]` to the FPGA dynamic region for memory access. The bridge handles the other side, i.e. the host CPU, through a PCIe connection. As mentioned in Chapter 4, the U250 platform is already supported. This host memory access can be easily done by modifying the kernel linking configuration file. But we have not evaluate the performance for it. On both the Alveo U250 PCIe 3x16 platform and the U280 platform with PCIe 4x8, the theoretical PCIe throughput is around 126 Gbps [42, 43]. The actual memory throughput for U250 PCIe 3x16 measured by XRT tools is about 75-95 Gbps on the XACC cluster. This memory bandwidth is lower than the networking bandwidth, which means that there will be throughput degradation for this setup. Another direction is to investigate the QDMA shell for Alveo U250 FPGAs. The QDMA shell can transfer data as a stream rather than memory mapping. Although this might have lower bandwidth, it is still worth checking out. With host memory access benchmarks, we hope to see latency superiority for RDMA stack over TCP stack.

Scalability investigation
The scalability of our designed RoCE stack has not been investigated. The `rocev2` IP supports thousands of QPs which means huge scalability. Our stack is thus theoretically capable of multiple RDMA connections. This is important for applications that require more inter-connectivity. It is also important to investigate the impact of growing connection counts on throughput, latency, and resource utilization.

XRC transport service
Speaking of scalability, upgrading RC service to the Extended Reliable Connection transport service could also be beneficial. As mentioned in Section 3.1, the XRC requires much fewer QPs for a more complex interconnection between many processes on multiple nodes. This work should be done with HLS as an addition to the `fpga-network-stack` project[2].

User kernel memory interface
As mentioned in Chapter 4, the User kernel can only send data to the RoCE kernel but not receive data from it. This means in-networking processing is only possible at the sender FPGA. To enable such processing on the receive FPGA, there should be an interface for data transmission from the RoCE kernel to the User kernel to allow incoming data to be processed before being stored to the memory.

Fletcher integration
One of the triggers of this project is in the context of Apache Arrow data format[3]. It is a standardized columnar format of data for any language and platform. This uniform format avoids the serialization and de-serialization step when data is exchanged between different systems with different languages and databases. Built on top of Arrow there is the Arrow Flight data transport framework[4]. It supports Arrow format data to be efficiently transformed through Google's Remote Procedure Call (gRPC) which is based on TCP/IP. RDMA network stack is thus a potential plus for Arrow Flight because of its low overhead and CPU bypass feature. As shown in our results, the throughput and latency could both benefit from an RDMA stack. Fletcher [20] is a framework to integrate FPGA using Apache Arrow format. It provides tools to generate the interface as well as an accelerator kernel template for Arrow

[1] `https://xilinx.github.io/XRT/2020.2/html/sb.html`
[2] `https://github.com/fpgasystems/fpga-network-stack`
[3] `https://arrow.apache.org/`
[4] `https://arrow.apache.org/blog/2019/10/13/introducing-arrow-flight/`

data on FPGAs. With Fletcher integrated into the User kernel of our RoCE network stack, we can present a complete framework for Arrow Flight over RDMA with FPGA accelerators.

Big data applications
The RDMA network stack is designed for efficient communications for data exchange between high-performance data centers. As networking is becoming one of the bottlenecks for big data processing, this RDMA stack has great potential to address this problem. Big data applications such as genomics analysis are often pipelined for higher processing throughput. Our work can be integrated into such use cases to improve their performance.

A

Appendix: Alveo Device Utilisation

The device utilization of our RoCE v2 stack with an RDMA READ User kernel on Alveo U280 FPGAs is shown in Figure A.1. The RDMA WRITE User kernel has the same design as a READ kernel, with only differences in several internal parameters. So the device utilization remains the same.

The device utilization of the stack with a dummy User kernel on Alveo U250 FPGAs is shown in Figure A.2. The dummy kernel contributes to nearly zero extra utilization as it do not contain any logic. Hence, this figure shows the device utilization of the RoCE kernel and CMAC kernel.

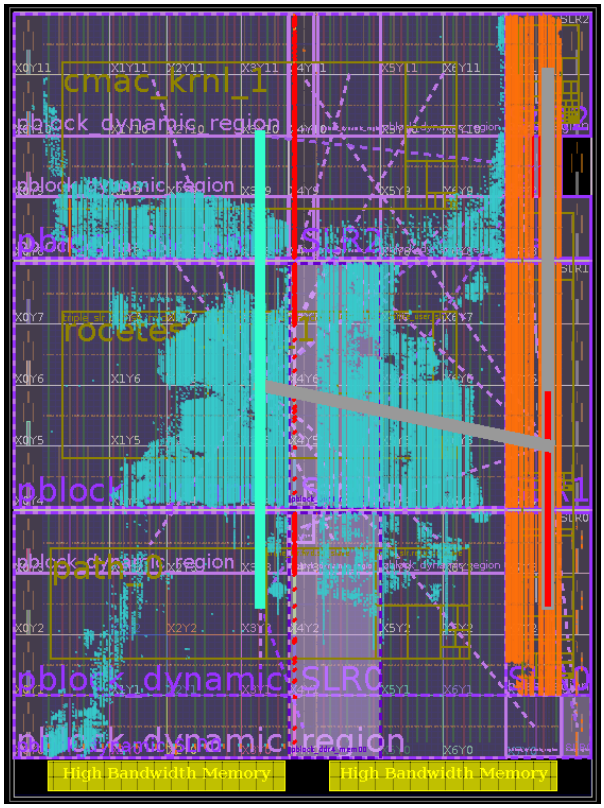


Figure A.1: Alveo U280 device utilisation for the RoCE stack with an READ/WRITE User kernel

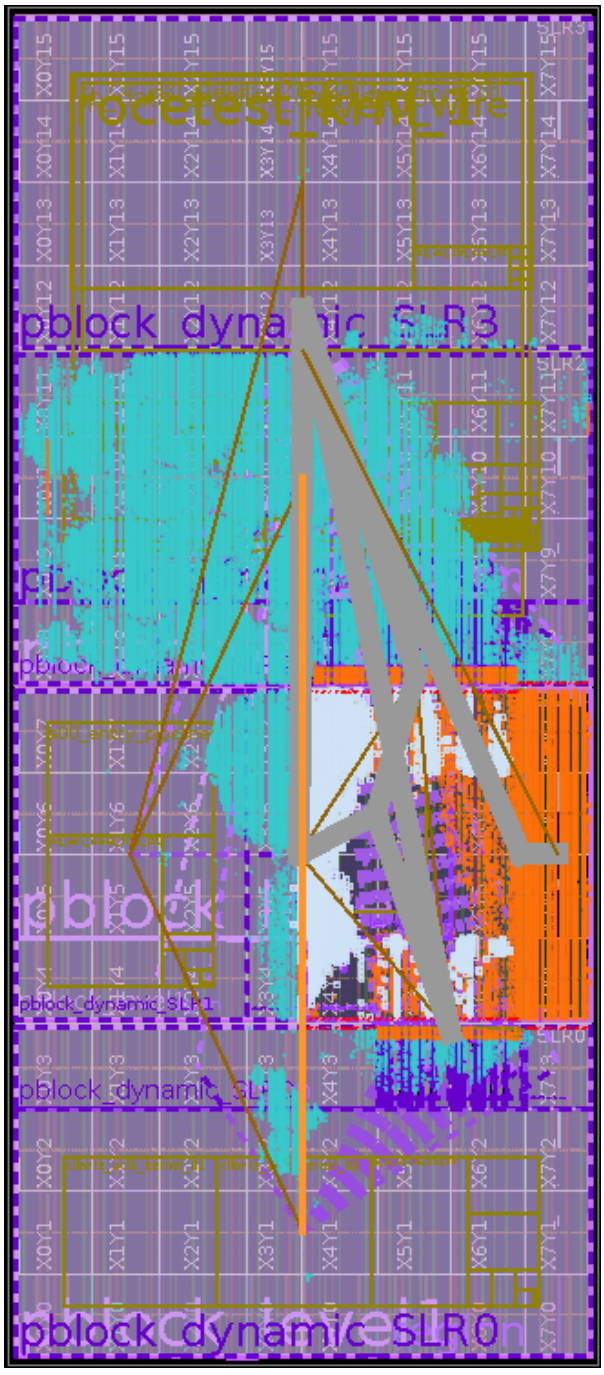


Figure A.2: Alveo U250 device utilisation for the RoCE stack with a dummy User kernel.

Bibliography

[1] Networking — The Linux Kernel documentation. URL https://linux-kernel-labs.github.io/refs/heads/master/labs/networking.html.

[2] Nikolaos Alachiotis, Simon A. Berger, and Alexandros Stamatakis. Efficient PC-FPGA Communication over Gigabit Ethernet. In *2010 10th IEEE International Conference on Computer and Information Technology*, pages 1727–1734, June 2010. doi: 10.1109/CIT.2010.302.

[3] InfiniBand Trade Association. InfiniBand™ Architecture Specification Volume 1: Release 1.4, 2020.

[4] Serhat Nazim Avci, Zhenjiang Li, and Fangping Liu. Congestion aware priority flow control in data center networks. In *2016 IFIP Networking Conference (IFIP Networking) and Workshops*, pages 126–134, May 2016. doi: 10.1109/IFIPNetworking.2016.7497228.

[5] Chelsio Communications. The Case Against iWARP. Technical report, Chelsio Communications, 2015.

[6] Philippe Coussy and Adam Morawiec. *High-level synthesis*, volume 1. Springer, 2010.

[7] A. Dollas, I. Ermis, I. Koidis, I. Zisis, and C. Kachris. An open TCP/IP core for reconfigurable logic. In *13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'05)*, pages 297–298, April 2005. doi: 10.1109/FCCM.2005.20.

[8] W. Feng, P. Balaji, C. Baron, L.N. Bhuyan, and D.K. Panda. Performance characterization of a 10-Gigabit Ethernet TOE. In *13th Symposium on High Performance Interconnects (HOTI'05)*, pages 58–63, August 2005. doi: 10.1109/CONECT.2005.30. ISSN: 2332-5569.

[9] Dror Goldenberg. InfiniBand Technology Overview. page 39, 2007.

[10] Zhenhao He, Dario Korolija, and Gustavo Alonso. EasyNet: 100 Gbps Network for HLS. 2021. doi: 10.3929/ETHZ-B-000487920. URL http://hdl.handle.net/20.500.11850/487920. Medium: application/pdf,7 p. accepted version Publisher: ETH Zurich.

[11] Fernando Luis Herrmann, Guilherme Perin, Josue Paulo Jose de Freitas, Rafael Bertagnolli, and Joao Baptista dos Santos Martins. A Gigabit UDP/IP network stack in FPGA. In *2009 16th IEEE International Conference on Electronics, Circuits and Systems - (ICECS 2009)*, pages 836–839, December 2009. doi: 10.1109/ICECS.2009.5410757.

[12] Yi-Mao Hsiao, Ming-Jen Chen, Kuo-Chang Huang, Yuan-Sun Chu, and Chingwei Yeh. High speed UDP/IP ASIC design. In *2009 International Symposium on Intelligent Signal Processing and Communication Systems (ISPACS)*, pages 405–408, January 2009. doi: 10.1109/ISPACS.2009.5383815.

[13] Yong Ji and Qing-Sheng Hu. 40Gbps multi-connection TCP/IP offload engine. In *2011 International Conference on Wireless Communications and Signal Processing (WCSP)*, pages 1–5, November 2011. doi: 10.1109/WCSP.2011.6096913.

[14] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using RDMA efficiently for key-value services. In *Proceedings of the 2014 ACM conference on SIGCOMM*, SIGCOMM '14, pages 295–306, New York, NY, USA, August 2014. Association for Computing Machinery. ISBN 978-1-4503-2836-4. doi: 10.1145/2619239.2626299. URL https://doi.org/10.1145/2619239.2626299.

[15] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Design Guidelines for High Performance {RDMA} Systems. pages 437–450, 2016. ISBN 978-1-931971-30-0. URL https://www.usenix.org/conference/atc16/technical-sessions/presentation/kalia.

[16] Jiuxing Liu, Jiesheng Wu, and Dhabaleswar K. Panda. High Performance RDMA-Based MPI Implementation over InfiniBand. *International Journal of Parallel Programming*, 32(3):167–198, June 2004. ISSN 1573-7640. doi: 10.1023/B:IJPP.0000029272.69895.c1. URL https://doi.org/10.1023/B:IJPP.0000029272.69895.c1.

[17] Wassim Mansour, Nicolas Janvier, and Pablo Fajardo. FPGA Implementation of RDMA-Based Data Acquisition System Over 100-Gb Ethernet. *IEEE Transactions on Nuclear Science*, 66(7): 1138–1143, July 2019. ISSN 1558-1578. doi: 10.1109/TNS.2019.2904118. Conference Name: IEEE Transactions on Nuclear Science.

[18] Sándor Molnár, Balázs Sonkoly, and Tuan Anh Trinh. A comprehensive TCP fairness analysis in high speed networks. *Computer Communications*, 32(13):1460–1484, August 2009. ISSN 0140-3664. doi: 10.1016/j.comcom.2009.05.003. URL https://www.sciencedirect.com/science/article/pii/S0140366409001078.

[19] Michael Oberg, Henry M Tufo, Theron Voran, and Matthew Woitaszek. Evaluation of RDMA Over Ethernet Technology for Building Cost Effective Linux Clusters. page 13, 2006.

[20] J. Peltenburg, J. van Straten, L. Wijtemans, L. van Leeuwen, Z. Al-Ars, and P. Hofstee. Fletcher: A Framework to Efficiently Integrate FPGA Accelerators with Apache Arrow. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pages 270–277, September 2019. doi: 10.1109/FPL.2019.00051. ISSN: 1946-1488.

[21] J. Postel. User datagram protocol. STD 6, RFC Editor, August 1980. URL http://www.rfc-editor.org/rfc/rfc768.txt. http://www.rfc-editor.org/rfc/rfc768.txt.

[22] Jon Postel. Internet protocol. STD 5, RFC Editor, September 1981. URL http://www.rfc-editor.org/rfc/rfc791.txt. http://www.rfc-editor.org/rfc/rfc791.txt.

[23] Jon Postel. Transmission control protocol. STD 7, RFC Editor, September 1981. URL http://www.rfc-editor.org/rfc/rfc793.txt. http://www.rfc-editor.org/rfc/rfc793.txt.

[24] R. Recio, B. Metzler, P. Culley, J. Hilland, and D. Garcia. A remote direct memory access protocol specification. RFC 5040, RFC Editor, October 2007.

[25] Mario Ruiz, David Sidler, Gustavo Sutter, Gustavo Alonso, and Sergio López-Buedo. Limago: An FPGA-Based Open-Source 100 GbE TCP/IP Stack. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pages 286–292, September 2019. doi: 10.1109/FPL.2019.00053. ISSN: 1946-1488.

[26] Alexander Shpiner, Eitan Zahavi, Omar Dahley, Aviv Barnea, Rotem Damsker, Gennady Yekelis, Michael Zus, Eitan Kuta, and Dean Baram. RoCE Rocks without PFC: Detailed Evaluation. In *Proceedings of the Workshop on Kernel-Bypass Networks*, KBNets '17, pages 25–30, New York, NY, USA, August 2017. Association for Computing Machinery. ISBN 978-1-4503-5053-2. doi: 10.1145/3098583.3098588. URL https://doi.org/10.1145/3098583.3098588.

[27] David Sidler. *In-Network Data Processing using FPGAs*. PhD thesis, ETH Zurich, 2019-09.

[28] David Sidler, Gustavo Alonso, Michaela Blott, Kimon Karras, Kees Vissers, and Raymond Carley. Scalable 10Gbps TCP/IP Stack Architecture for Reconfigurable Hardware. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 36–43, Vancouver, BC, Canada, May 2015. IEEE. ISBN 978-1-4799-9969-9. doi: 10.1109/FCCM.2015.12. URL http://ieeexplore.ieee.org/document/7160037/.

[29] David Sidler, Zeke Wang, Monica Chiosa, Amit Kulkarni, and Gustavo Alonso. Strom: Smart remote memory. In *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020. doi: 10.1145/3342195.3387519.

[30] Charles E Spurgeon. *Ethernet: the definitive guide*. ” O’Reilly Media, Inc.”, 2000.

[31] József Sütő and Stefan Oniga. FPGA implemented reduced Ethernet MAC. In *2013 IEEE 4th International Conference on Cognitive Infocommunications (CogInfoCom)*, pages 29–32, December 2013. doi: 10.1109/CogInfoCom.2013.6719258.

[32] Mellanox Technologies. RoCE vs. iWARP Competitive Analysis. White paper, Mellanox Technologies, 2017. URL https://www.mellanox.com/related-docs/whitepapers/WP_RoCE_vs_iWARP.pdf.

[33] Brian Tierney, Ezra Kissel, Martin Swany, and Eric Pouyoul. Efficient data transfer protocols for big data. In *2012 IEEE 8th International Conference on E-Science*, pages 1–9, October 2012. doi: 10.1109/eScience.2012.6404462.

[34] Tomohisa Uchida. Hardware-based tcp processor for gigabit ethernet. *IEEE transactions on nuclear science*, 55(3):1631–1637, 2008.

[35] Jean-Philippe Vasseur and Adam Dunkels. Chapter 6 - transport protocols. In Jean-Philippe Vasseur and Adam Dunkels, editors, *Interconnecting Smart Objects with IP*, pages 63–74. Morgan Kaufmann, Boston, 2010. ISBN 978-0-12-375165-2. doi: https://doi.org/10.1016/B978-0-12-375165-2.00006-5. URL https://www.sciencedirect.com/science/article/pii/B9780123751652000065.

[36] D. Waitzman. Standard for the transmission of ip datagrams on avian carriers. RFC 1149, RFC Editor, April 1990.

[37] D. Waitzman. Ip over avian carriers with quality of service. RFC 2549, RFC Editor, April 1999.

[38] Felix Winterstein, Samuel Bayliss, and George A. Constantinides. High-level synthesis of dynamic data structures: A case study using Vivado HLS. In *2013 International Conference on Field-Programmable Technology (FPT)*, pages 362–365, December 2013. doi: 10.1109/FPT.2013.6718388.

[39] *10 Gigabit Ethernet Media Access Controller v15.1 LogiCORE IP Product Guide*. Xilinx, 2018.

[40] *Xilinx Embedded Target RDMA Enabled NIC v1.1 LogiCORE IP Product Guide*. Xilinx, 6 2018.

[41] *AXI Ethernet Lite MAC v3.0 LogiCORE IP Product Guide*. Xilinx, 2020.

[42] *Alveo U200 and U250 Data Center Accelerator Cards Data Sheet*. Xilinx, 5 2020.

[43] *Alveo U280 Data Center Accelerator Card Data Sheet*. Xilinx, 2021.

[44] *UltraScale+ Devices Integrated 100G Ethernet Subsystem v3.1 Product Guide*. Xilinx, 2021.