ENABLING INDEPENDENT COMMUNICATION FOR FPGAS IN HIGH PERFORMANCE COMPUTING

A THESIS SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
IN THE FACULTY OF SCIENCE AND ENGINEERING

2019

Joshua Lant
School of Computer Science

Contents

Word count 54081

List of Tables

List of Figures

The University of Manchester

Joshua Lant

Doctor of Philosophy

Enabling Independent Communication for FPGAs in High Performance Computing

June, 2019

The landscape of High Performance Computing is changing, with increasing heterogeneity, new data-intensive workloads and ever tighter system power constraints. Given these changes there has been increased interest in the deployment of FPGA technology within HPC systems. Traditionally FPGAs have been of limited use to the HPC community. However, there have been many architectural advances in recent years; hardened floating-point operators and on-die CPUs, greater on-chip memory capacity, increased off-chip memory bandwidth but to name a few. These advances have brought the opportunity to more readily exploit the FPGA's efficiency and flexibility in HPC. Unfortunately there are still a number of research problems to be solved in order to allow this to happen. In this thesis we tackle one such problem; regarding the interconnect and its relation to the system architecture.

The interconnect must have several key properties in order to satisfy the demands of large, data-intensive applications and take advantage of dataflow processing for FPGA based HPC. It must ***(i)*** allow for tight coupling between FPGA and system memory in both local *and* remote nodes. This is required to enhance the performance of a number of key workloads which exhibit irregular memory access patterns. ***(ii)*** It must allow for the FPGA to issue and process network transactions without any CPU intervention. This is required for high performance inter-FPGA communication and independent scaling (disaggregation) of the FPGA resources. ***(iii)*** The interconnect must maintain its key properties of scalability and reliability; required for HPC systems but at odds with the other primary requirements.

In this thesis we present a novel Network Interface solution implemented entirely within the fabric of the FPGA, which attempts to address all of these competing factors. The Network Interface allows for a system architecture which better supports distributed FPGA processing within a shared, global address space. It provides hardware primitives to support RDMA and shared-memory transfers over a lightweight, custom network protocol. It allows for direct inter-FPGA communication without any CPU intervention; supported via a hardware-offloaded, reliable *and* connectionless transport layer.

The microarchitecture of the Network Interface and transport layer are detailed, as well as a number of performance enhancements which reduce the latency and increase the achievable throughput of the system. We assess the consistency issues and network errors which can occur, and show how the Network Interface is able to support Out-Of-Order packets from the network. In the latter part of the thesis we show the benefits of direct inter-FPGA communication for dataflow processing when compared with a software based transport, and demonstrate how we can estimate the expected performance of such a system for network-bound processing.

Declaration

No portion of the work referred to in the thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright Statement

Acknowledgements

There are of course a multitude of people to thank for their support and guidance in helping me to reach this point. The first of these is obviously my Mother and Father, who provided me with all the advantages in life that so many people in this world are not afforded. I thank them, along with my sister Sian for their continued love and support.

I must also thank my Auntie Christine for providing me with the money to purchase my first laptop. Undoubtedly this enabled me to explore computer music more deeply, helping me find my undergraduate degree. My late Uncle Christopher was also a source of great inspiration to me; easily one of the most intelligent and interesting people I have ever known. I miss him dearly and I wish I were able to share this work with him as it would have interested him deeply.

I would also like to thank my old friends Lee and Al. They both provided me with a lot of inspiration during my most formative years. I am very lucky to have been able to learn so much from them at such a young age. Although computer architecture is not a pursuit my teenage self would have ever envisaged for his future, without Lee and Big Al I would likely be on an entirely different path!

Thank you to all my friends, and house-mates past and present over the past three and a half years. They have kept me sane and having them all in my life has brightened it immensely. Special mention must be given to Sophie Hayter, who welcomed me into her life with open arms and who provides me great joy. I must also give special thanks and apologies to my other long suffering house-mates; Nelly and Josh. Living with a PhD student cannot be easy at times, particularly one with my temperament. I thank them for their patience and love.

Thank you to my beautiful girlfriend Stef. She has supported me so much over the past year and a half it is beyond words. She has motivated me to continue

working and has taught me to view the world in a completely different way. She is a remarkable human being.

I must now thank all of my colleagues over the years from the APT group and beyond. Particularly my workmates Andrew Attwood, Jose Pascual and Caroline Concatto. It was a joy to be able to work closely with them. I had a lot of fun and learned many things. Our discussions were of immeasurable help in focusing my work, with their guidance possibly preventing me from throwing in the towel at certain points. I must also thank Fabian Chaix. He was a great friend during my time on Crete and has given me many fond memories. His knowledge is deep and he is a real workhorse of an engineer. Thanks as well go to John Mawer who helped me with a lot of technical problems during the earlier stages of my time at Manchester.

Last and *in no respect* least, I must thank my supervisor Javier Navaridas. He has taught me so much about the world of academia, and has guided me through my PhD with attentiveness; providing me assistance when it was needed most and keeping me on track. Our discussions are always fruitful, and it has been a pleasure to learn from him. I am deeply grateful for everything he has done for me, and without hesitance I would recommend him as an advisor to anyone! Thank you.

Chapter 1

Introduction

In recent years there have been two great changes which have affected the way systems architects must think about future technology within High Performance Computing. The most obvious and devastating of these is the breakdown of Dennard scaling [1]. After around 2004 the single threaded performance of the processor has seemingly plateaued. Continued transistor scaling has not been able to deliver the same energy scaling due to the effects of rising current leakage. A consequence of this has been that scale-out and the use of accelerators has become the new norm for increasing performance. As such the power consumption of the largest machines has become a major issue, thwarting further scalability of traditional HPC architectures. Energy efficiency is now without doubt the number one concern of the HPC community, and achieving the target of 1 exaFLOP (10^{18} floating point operations per second) at 20MW [2] requires efficiency to be squeezed from every level; from applications down to transistor technology.

The second great change is that we have now entered the *Fourth Paradigm* of scientific discovery [3]. Modern applications are moving from computational science into the era of big-data analytics The rapid growth of data-intensive workloads has instigated a convergence between data centre and HPC technology, and techniques such as hyper-converged storage are increasingly used to bring data closer to compute resources. The price of computation has been falling dramatically for decades, to the point that the energy required for on-chip data movement is now significantly higher than that of a double precision floating point operation [4]. As such, architects must now focus on reducing data movement as a method for reducing

power consumption.

Given this current focus on reducing power consumption and given the changing profile of HPC workloads, we see that the currently dominant architectural model of CPU coupled with bus-attached GPU accelerator has fundamental issues and will not be able to provide for the scaling and efficiency requirements of future systems. GPUs undeniably provides unbeatable performance for compute intensive workloads with regular memory accesses due to its high memory bandwidth. Despite this the GPU has some drawbacks; not only are high end GPUs growing notably more power hungry [5], but their performance in certain domains is limited. In applications exhibiting irregular parallelism, lower computational intensity, or close interaction with the CPU or network the GPU can be seriously hindered, and thus requires alternative architectures. Coupling this with the fact that the PCIe attached GPU model foments large amounts of data movement, it is reasonable to question the scalability of such architectures when considering energy efficiency.

There has been growing interest recently in the feasibility of using FPGAs for general HPC systems, and in FPGA technology in general. With Intel's acquisition of Altera[1], IBMs strategic partnership with Xilinx[2], and as major players such as Amazon[3] and Microsoft [6] have introduced FPGAs into their data centres, it seems that this interest is increasingly being shown in industry as well in academic circles. FPGAs are highly regarded for their performance-per-watt over GPU technology [5], and their high performance for workloads able to take advantage of fine-grained, deep pipelining and dataflow processing [7]. Unfortunately there are a number of obstacles regarding programmability, reliability, system architecture, libraries etc. which currently prevent FPGA technology from being exploited to its full potential within the HPC domain [8]. Overcoming one such obstacle, with regards to the interconnection network and its relation to the system architecture, is the main goal of the work detailed in this thesis.

[1]Intel to Acquire Altera- `https://www.hpcwire.com/off-the-wire/intel-to-acquire-altera/`, accessed April 2019.

[2]IBM and Xilinx at SC15- `https://www.anandtech.com/show/9790/ibm-xilinx-sc15-collaborating-for-better-powerfpga-system-integration`, accessed April 2019.

[3]Amazon EC2 F1 Instances- `https://aws.amazon.com/ec2/instance-types/f1/`, accessed April 2019.

1.1 FPGAs for HPC

In order to make efficient use of FPGA technology within HPC systems the FPGA must be upgraded from its status as a mere coprocessor, and be viewed as the main compute element within the system, capable of issuing and processing network data (for distributed FPGA computing), and capable of global, shared access to system memory (to allow CPU and FPGA to use fine grained parallelism, making several application domains far more viable [9]).

One of the main issues with FPGAs is that the off-chip memory bandwidth is very limited when compared with GPUs [10]. However, the main aim in reducing data movement is to reduce these off-chip memory accesses, instead using algorithms which seek to maximise the reuse of on-chip memory, keeping data close to the compute. Many of the issues with mapping algorithms onto current FPGAs is the data-sets can be much larger than the available on-chip memory of the FPGA. It is this reason why having the FPGA tightly coupled with the network is so vital for FPGA based HPC systems, enabling the efficient use of distributed FPGA resources.

Using distributed FPGAs requires a high performance interconnect with which FPGA resources can communicate directly with one another, without the additional overheads of issuing network transactions via the CPU. There are many solutions which exist that allow for FPGAs to communicate with one another. However, current interconnect solutions are typically limited in one of the following ways:

- They provide only simple point-to-point connections between FPGAs; limiting available topologies and scalability of solutions to those typically situated within a single rack or chassis [11].

- They require CPU intervention to issue transactions to the network. This causes additional latency/bandwidth overheads, with additional buffering, system calls or control information required.

- They do not provide tight coupling to system memory, reducing the ability of the system to take advantage of fine grained parallelism over distributed FPGAs or proper coordination between CPU and FPGA, inhibiting certain

workloads such as N-body problems and FFTs [9].

- They do not guarantee reliable transfer, so are not amenable for production-level systems.
- Their hardware offloaded transport layers require per-connection state information/buffering to be stored within the hardware, limiting scalability or degrading performance by excessive connection setup/teardown.

1.1.1 Our Interconnect Solution

The main aim of this thesis is to provide a custom interconnect solution which aims to provide functionality which overcomes each of the limitations listed in Section 1.1. The first step to achieving this is to design a custom Network Interface (NI) which provides hardware primitives to support a hybrid MPI+PGAS programming model. Other work has discussed the need for custom interfaces, and the pitfalls of using off-the-shelf solutions in an FPGA based environment [12]. The desire to provide direct support for MPI via an RDMA (Remote Direct Memory Access) mechanism stems from its complete ubiquity, and any HPC application modified from prior work is almost guaranteed to use MPI in the medium to long term, regardless of the demonstrated scalability and performance issues [13], [14]. We provide support for PGAS-like communications by providing a direct mechanism to read/write into remote memory using a transparent load/store operation from the CPU or FPGA fabric [15]. In encapsulating and extending the system bus protocol to work over the network, there is no change from the user perspective in accessing remote memory from local memory (other than the Non-Uniform Memory Access effects of additional latency).

The second, and most important way in which we provide for those properties of the network described above is by creating a novel transport layer which is implemented within the fabric of the FPGA. The CPU is not involved in network communications between FPGA resources whatsoever, and the solution provides two entirely separate methods for reliable transfer, depending on the primitive invoked (RDMA transfer or shared-memory operation). Doing this enables some beneficial properties of the interconnect. The first is that the shared-memory operations and

RDMA transfers have very different requirements, and our solution is tailored to increase the performance of these two types of operation. Shared-memory operations typically require lower latency as they comprise messages required for control or synchronization, whereas RDMA transfers are used to saturate the high-speed links which the network uses for data transfer and so require higher throughput.

The second benefit which arises from this is the fact that we allow for a connectionless transport layer within the hardware, which is able to offer guaranteed delivery of messages. What we mean by this is that no state information needs to be held in the Network Interface, the connection information is implicit within the transfer as all network transactions are effectively memory read/writes. Typically to provide reliability requires a connection based transport, with persistent state information required to be held within the Network Interface. In our solution we only maintain transient information regarding outstanding operations which have left the sender. We are able to do this by separating the transport layers, having one for shared-memory operations in which retransmission information can safely be held within the Network Interface. We then provide an entirely separate transport for reliable RDMA transfer. In this instance one which is more scalable as the information is held in its source memory location, rather than being copied into retransmission buffers within the Network Interface.

In the latter part of the thesis we deal with several issues which arise in terms of the performance of the system. These issues are primarily surrounding overlapping computation and communication, and present themselves owing to two factors. The first is that the transport mechanism and lack of retransmission buffers prevents overwriting of the data at the source before the transfer is complete. The second is the fact that our network allows for out-of-order packet delivery, which makes it difficult for the receiver to track the state of RDMA transfers. We provide a mechanism to allow the receiver to acknowledge transfers without the sender issuing a notification, thus reducing the latency of these transfers.

1.2 Contributions

A summary of the contributions of this thesis are as follows:

- The design and implementation of a novel microarchitecture for a Network Interface which provides direct hardware support for both PGAS/NUMA and RDMA transfers, over a custom network protocol. The architecture has been designed specifically to support High Performance Computing within FPGA based systems. We show the benefits that arise from providing two separate communication mechanisms, given that the latency for small, low latency transfers over the global shared-memory path is over 25% lower than using RDMA.

- The design and implementation of a fully hardware-offloaded, reliable *and* connectionless (in the Network Interface) transport layer, which provides end-to-end packet retransmissions for the transfer of data over a custom network protocol. This novel solution separates out traffic classes, providing a lower latency, high performance solution for global shared-memory accesses, and a more scalable solution for RDMA transfers.

- We address several limitations in the baseline system, and propose several enhancements to the architecture. ***(i)*** Direct receive-side notifications which eliminate a round trip packet transmission for lower latency RDMA transfers. ***(ii)*** Segmentation of large transfers in order to allow for more efficient communication/computation overlap. ***(iii)*** Early acknowledgement for system bus transactions, preventing pipeline stalling and maximising RDMA transmission throughput. These have been implemented within the Network Interface and are specifically designed to alleviate the issues we see from out-of-order packet delivery from the network. An analysis of one of these performance enhancements– the receive-side notifications– shows that this mechanism reduces the latency of smaller RDMA transfers by around 20% when compared to the baseline solution.

- Our system allows the FPGA fabric to issue reliable network transactions for both shared-memory and RDMA operations. We show how our system therefore allows distributed FPGA resources to communicate with one another directly, avoiding CPU intervention completely and providing a more suitable model for dataflow style processing. We show that our architecture has two

fundamental properties, which we argue are both required for the uptake of FPGAs within HPC systems, and that these two properties have not been seen together in any prior architecture. The solution must provide tight coupling to the main memory system of the CPU, and be available for access by local *and* remote resources. Secondly the FPGA resources must be completely decoupled from the CPU with regards to the network; capable of issuing and processing its own network transactions. Aside from this architectural advancement in disaggregating the CPU and FPGA resources, our results show that the performance in terms of data processing throughput and latency is enhanced by up to 8.6% and 29% respectively, when compared with a software based transport mechanism.

- We show how the results of our experiments into distributed FPGA resources can be used to estimate the performance and resource consumption of network communication bound processing within an FPGA. Our results show that we can achieve comparable performance to other recent work in the literature [11], while providing a much more sophisticated and scalable networking solution.

- An analysis into the implications of network errors, specific architectural features, and out-of-order packet delivery on data consistency within our system. We show how inconsistencies can arise, and the ways in which we are able to mitigate against these within the Network Interface and within our application model.

- An analysis of different link-level and end-to-end fault tolerance mechanisms, comparing the performance of these for the proposed system-wide network. We conclude that for our case targeting a modern *Dragonfly* topology, there is little requirement to provide link-level fault tolerance strategies, as the performance in the normal case is heavily degraded.

1.3 Publications

Below is a list of publications from the author. Those with the author name in **bold text** are those in which the material is directly related to the work presented within this thesis:

Journal Articles

- **Joshua Lant**, Javier Navaridas, Mikel Luján, and John Goodacre. "Toward FPGA-Based HPC: Advancing Interconnect Technologies." *Accepted 2019, To appear in IEEE Micro.*

- **Joshua Lant**, Caroline Concatto, Andrew Attwood, Jose A. Pascual, Mike Ashworth, Javier Navaridas, Mikel Luján, and John Goodacre. "Enabling shared memory communication in networks of MPSoCs." *Concurrency and Computation: Practice and Experience*: e4774.

- Jose A. Pascual, Joshua Lant, Caroline Concatto, Andrew Attwood, Javier Navaridas, Mikel Luján, and John Goodacre. "On the effects of allocation strategies for exascale computing systems with distributed storage and unified interconnects." In *Concurrency and Computation: Practice and Experience*: e4784.

Conferences

- **Joshua Lant**, Javier Navaridas, Andrew Attwood, Mikel Luján, and John Goodacre. "Enabling Standalone FPGA Computing." *IEEE 26th Biennial Symposium on High Performance Interconnects (HOTI) 2019*.

- Javier Navaridas, Joshua Lant, Jose A. Pascual, Mikel Luján, and John Goodacre. "Design Exploration of Multi-tier interconnects for Exascale systems." In *International Conference on Parallel Processing (ICPP)*, pp. 49, ACM 2019.

- **Joshua Lant**, Andrew Attwood, Javier Navaridas, Mikel Luján, and John Goodacre. "Receive-Side Notificaton for Enhanced RDMA in FPGA Based

Networks." In *International Conference on Architecture of Computing Systems*, pp. 224-235. Springer, Cham, 2019.

- Caroline Concatto, Jose A. Pascual, Javier Navaridas, **Joshua Lant**, Andrew Attwood, Mikel Lujan, and John Goodacre. "A CAM-Free Exascalable HPC Router for Low-Energy Communications." In *International Conference on Architecture of Computing Systems*, pp. 99-111. Springer, Cham, 2018.

- Jose A. Pascual, Joshua Lant, Andrew Attwood, Caroline Concatto, Javier Navaridas, Mikel Luján, and John Goodacre. "Designing an exascale interconnect using multi-objective optimization." In *2017 IEEE Congress on Evolutionary Computation (CEC)*, pp. 2209-2216. IEEE, 2017.

- Jose A. Pascual, Caroline Concatto, Joshua Lant, and Javier Navaridas. "On the Effects of Data-Aware Allocation on Fully Distributed Storage Systems for Exascale." In *European Conference on Parallel Processing*, pp. 725-736. Springer, Cham, 2017.

Workshops/Other

- **Joshua Lant**, and Javier Navaridas. "Direct Communication between Distributed FPGA Resources." In *Emerging Technology (EMiT) Conference*, pp. 16-19, 2019.

- **Joshua Lant**, "Reliable Communication in Networks of FPGAs" In *Towards Exascale HPC systems: co-design and technology development within the EuroEXA, ExaNeSt, ExaNoDe and EcoScale projects*, European HPC Summit Week 2018.

1.4 Architectural Overview and Thesis Structure

This Section contains an overview of the architecture of the entire Network Interface and system design which is presented within this work. Each of the relevant components on the following diagrams is numbered and briefly introduced in the listings. This can be used to direct the reader to the relevant location within the thesis, in which the respective component is discussed in detail.

1.4.1 Network Interface Design

Figure 1.1 shows the microarchitecture of the Network Interface (NI). Each of the labelled components are as follows:

1. **Module Interfacing-** The interfacing to the NI is discussed at various points within the thesis. Section 3.3.1 and Appendix C give an introduction and brief tutorial on the memory-mapped AXI protocol. Section 3.3.2 discusses the packet format used for the wider network. Section 3.3.4 discusses the segregation of traffic classes within the NI.

2. **Packetization-** Section 3.3.2.1 briefly discusses the bridging mechanism used to encapsulate AXI into a format suitable for the network. A detailed description of the controllers which have been developed to perform the bridging function is given in Appendix D.

3. **Inbound Messages-** Inbound messages require storage of metadata in order to build the response packets. Request packets add to the table and as responses are seen locally they are removed. This is discussed in Section 3.3.3.

4. **Retransmission of Shared-Memory Operations-** Shared memory operations are stored in the NI until acknowledged, the mechanism supporting this is shown in Section 5.2.2.

5. **Retransmission of RDMA Operations-** The reliability layer for RDMA operations is more complex, requiring the reforming of partial operations into new ones. This is shown in Section 5.2.3.

6. **Early Acknowledgements-** We implement a system of Early Acknowledgements for write operations in order to reduce the performance limitations of shared-memory operations. This has significant implications for the consistency model of the system. This is analysed in Chapter 4. The design of the mechanism is shown in Section 6.1.

7. **Registered Receive for RDMA-** In order to enhance the performance of RDMA operations, and enable them to be truly one-sided, we introduce a mechanism to post receive notifications directly from the receive-side NI. This is opposed

to waiting for notification of completion from the sender. This is discussed in Section 6.2 and analysed in Section 6.4.

8. **Operation Segmentation-** The DMA engine sits externally to the NI, with RDMA work items being issued through the NI to enable us to manipulate and track the operations. The mechanism for RDMAs is presented in Section 3.3.4.2. To better overlap communication and computation, we can segment large RDMA transfers into smaller ones. This is discussed in Section 6.3.

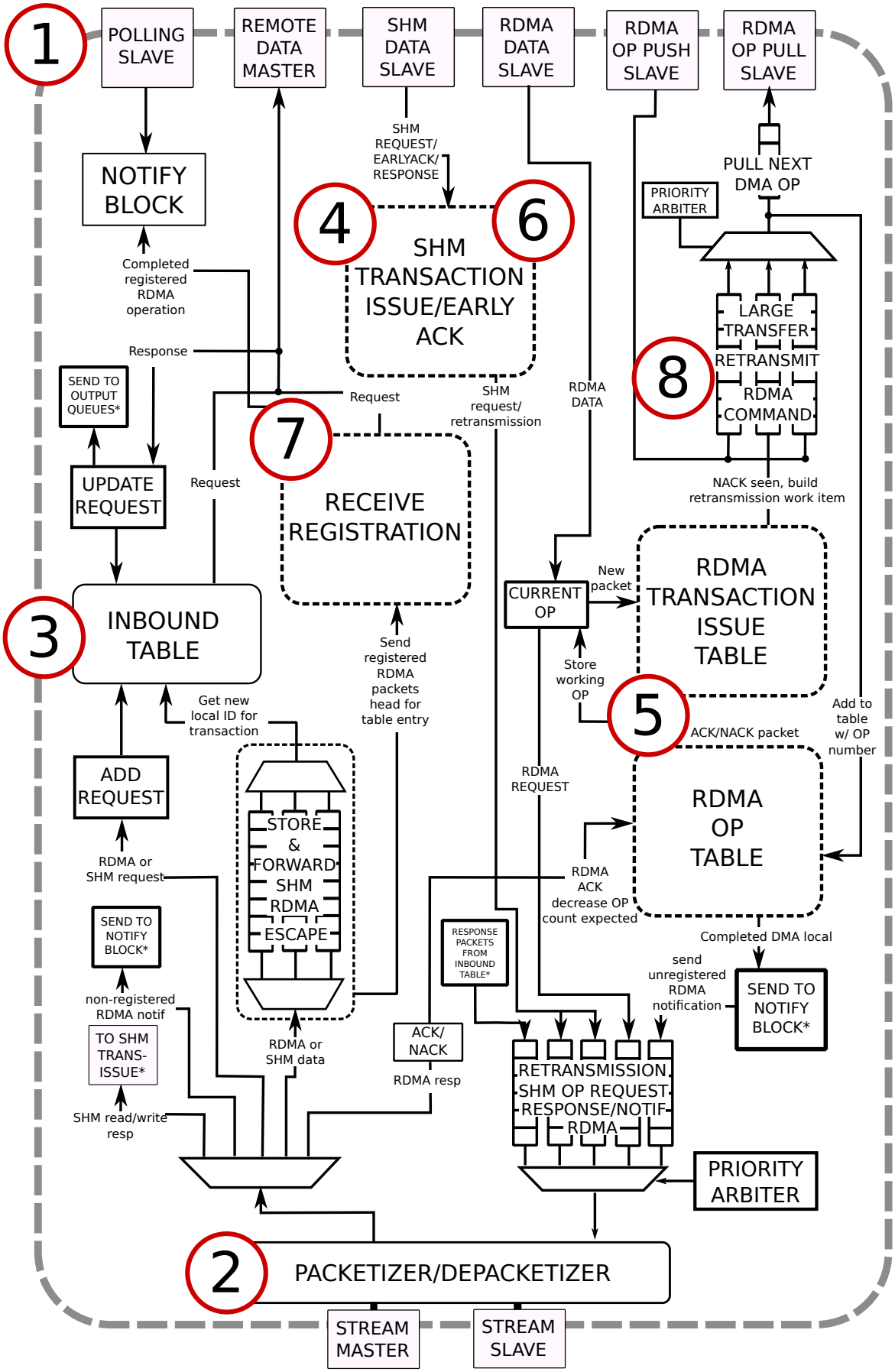


Figure 1.1: Complete Network Interface designed and implemented within this thesis.

1.4.2 System Level Design

Figure 1.2 shows an overview of the full system design implemented within a modern SoC device. Each of the labelled components are as follows:

1. **Network Interface-** The basic design of the Network Interface and associated relevant information is contained in Chapter 3.

2. **DMA Engine-** The DMA engine we currently use is a simple off-the-shelf IP from Xilinx [16], the basics are discussed in Section 3.3.4.2. We perform a basic analysis of the DMA functionality in Section 3.4.3, extending this in Section 6.1. As the DMA engine requires additional setup overheads, we provide a completely segregated data/control path through the NI, additionally supporting remote load/store operations. The benefits of doing this on small, latency critical packets is shown in Section 3.4.1.

3. **Network Switch-** The switch for the network is described in Section 3.1.3. As a consequence of the switch functionality out-of-order packet delivery is permitted from the network. The ramifications of this are detailed in Chapter 4.

4. **MAC/PHY and Transceivers-** We use the standard 10G Aurora MAC/PHY from Xilinx for much of the analysis in this work [17]. This is discussed in Section 3.2.3.

5. **Processing System Elements (CPU)-** The use of this particular processor (ARM Cortex-A53) has implications for the design in a number of ways. This is discussed in 3.2.2 and in Chapter 4.

6. **Accelerator Logic-** While the scope of this work does not extend to the performance of real-world distributed acceleration of applications, we analyse the potential performance that can be extracted from our solution for purely network bound communications in an application. This work is detailed in Chapter 7.

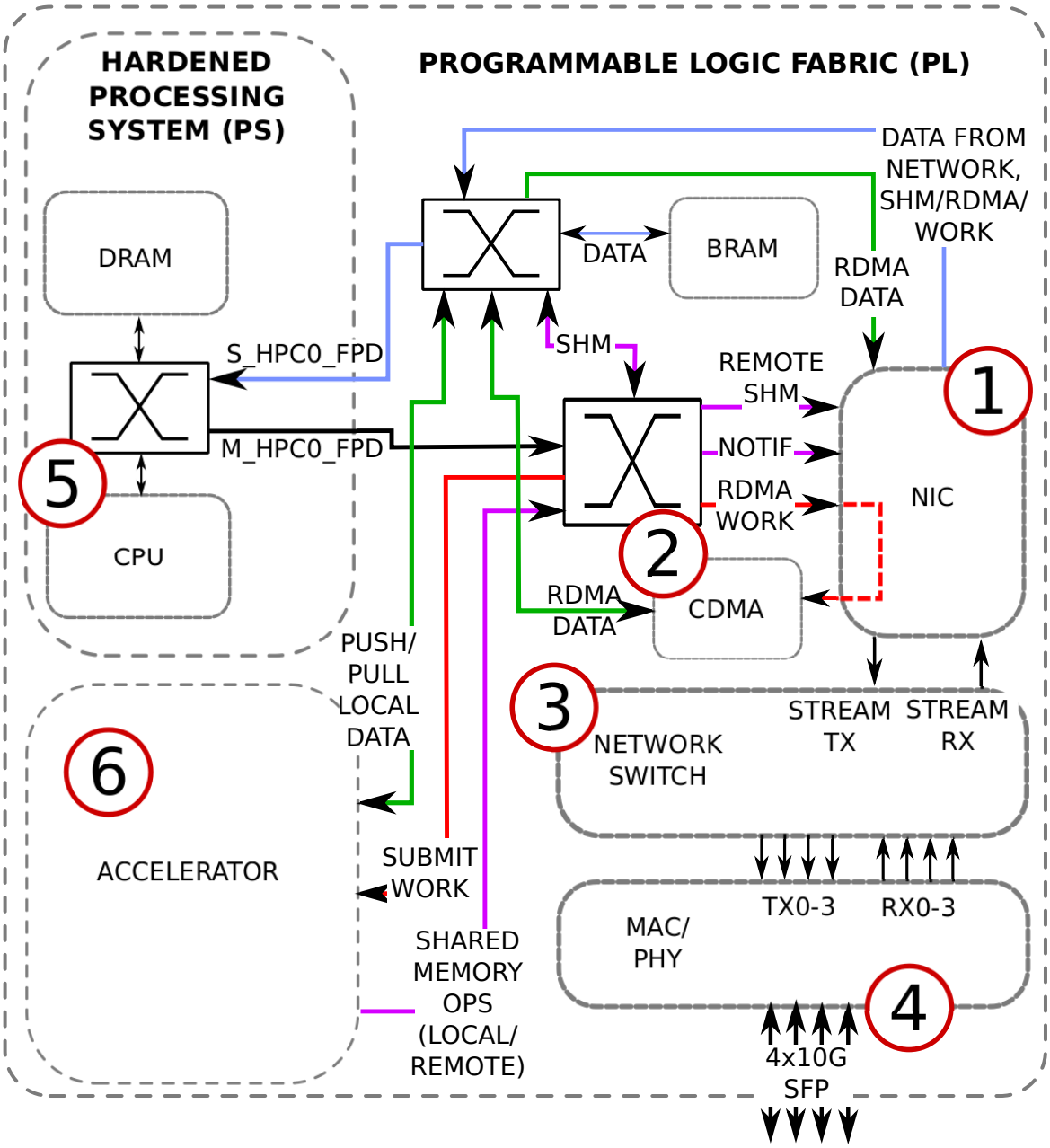


Figure 1.2: System level design of the full networking stack implemented on a Xilinx Ultrascale+ MPSoC device.

1.4.3 Thesis Outline by Chapter

- **Chapter 1** gives an introduction to our work and the motivation for pursuing FPGA based High Performance Computing.

- **Chapter 2** provides a background into many aspects of High Performance and Reconfigurable Computing, workloads, interconnects and programming models. It shows the ways in which the work presented within this thesis goes beyond currently available solutions, providing the required functionality for properly exploiting FPGAs within HPC.

- **Chapter 3** shows the basic Network Interface design and provides basic performance metrics for the two programming models targeted by the system (MPI+PGAS). It also provides an overview of the target system, topology, switching and interfacing, in order to give an appreciation of the framing of this thesis and rationale behind certain design decisions.

- **Chapter 4** explains the application and consistency model for the system, the network errors and inconsistencies that can occur and the mitigation we provide against them.

- **Chapter 5** details the microarchitecture of the hardware-offloaded, connectionless, reliable transport layer we have developed as part of the thesis. We discuss the issues surrounding its implementation, and at the end of the Chapter an analysis is provided into possible fault tolerance strategies for the network.

- **Chapter 6** details the microarchitecture of three performance enhancements we have added to the Network Interface. Experiments are provided which demonstrate the efficacy of these.

- **Chapter 7** provides a discussion of the reduced complexity we see in the control and data path when using our hardware-offloaded transport layer to allow distributed FPGA resources to communicate with one another. We demonstrate how latency and throughput are enhanced over a software based transport layer, and show how the results can be used to gain insight into the

possible computational power of arbitrary accelerator blocks constrained by network communication.

- **Chapter 8** gives a brief discussion on the thesis as a whole, and provides a discussion into the future directions for research and the ongoing work which would be required to enable production use of the Network Interface.

- **Appendix A** gives a brief overview of the projects in which the context of this work is placed. Introduced are a set of EU Horizon 2020 projects tasked with the design and implementation of system prototypes for future exascale class systems, specifically targeting novel low-power architectures.

- **Appendix B** provides an overview into the manner in which we provide access to the physical hardware within a user-space application without requiring the use of an IO Memory Management Unit. We show the creation of small windows of memory is performed using the `mmap()` system call, and then we discuss the limitations on addressing within the target device in general,

- **Appendix C** provides a short tutorial on the ARM's AXI interface standard. This is the system bus protocol used to interface between FPGA and CPU resources, and the overview is provided as an appreciation of certain elements of the NI design require a basic knowledge of the AXI protocol.

- **Appendix D** details the state machines and controllers used to bridge between the on-chip system bus protocol and the custom network protocol used for off-chip communications.

Chapter 2

Background and Related Work

In this Chapter we introduce many of the concepts and works revolving around High Performance Computing systems, and their communication models and protocols which have culminated in the work presented in this thesis. The 5 main topics we discuss are:

1. We begin with a brief history on the trends of HPC systems which leads us to conclude that the FPGA is a viable and attractive target for acceleration within future machines.

2. We then discuss the sorts of workloads/algorithms which modern HPC present. This informs the role of the FPGA within these future systems. We show that it is essential for architectures to move beyond a simple coprocessor model for accelerators, and view the FPGA as the main compute element within the system.

3. Then we look at the programming and communication models which are typical within HPC systems and FPGA programming. We show that providing hardware support at the Network Interface for both RDMA and direct remote shared memory accesses is a necessary design choice to provide high performance and simpler programmability.

4. Following this we will provide an overview of previous FPGA based HPC and data centre clusters and show that they do not fit all of our requirements,

either in terms of the proposed architecture, or in terms of the network capabilities. We indicate how our proposed system differs from those presented, thus providing an alternative approach over the current state of the art.

5. Finally we will introduce and evaluate a number of common and bespoke interconnect solutions for HPC systems. We conclude that a new custom interconnect solution is required to provide better exploitation of distributed FPGA resources in a HPC system with the FPGA viewed as the main compute element. We show that a hardware-offloaded transport layer is necessary to enable direct inter-FPGA communications. We argue that simpler interconnect solutions will not enable complex topologies with reduced diameter, and that common HPC network protocols are unsuited for FPGA based implementations. We also show that a connectionless transport layer is the only way to ensure scalability, and that a reliable *and* connectionless solution is possible given a globally shared address space.

2.1 Trends in High Performance Computing

There are many events and innovations throughout the years which have contributed to the changing landscape of supercomputing since the term was coined for the then-mighty CDC-6600 [18]. As we shall discuss in this section, over the years the primary concern of the architect has shifted from computation to communication costs. Advancements in compute have outstripped those of data movement as the power-wall has been faced. At the end of this Section we will show that new architectural approaches need to be sought to enable HPC systems to continue advancing forwards.

2.1.1 Vector Machines

Through the 1970's and 1980's the vector machine ruled supreme [19]. Seymour Cray's adage that two strong oxen would always be preferable to 1024 chickens for ploughing fields seemed irrefutable; particularly given the relative failure of the first massively parallel machine, the ILLIAC-IV [20]. The machines of this age

extracted large *data-level parallelism* by using single instructions to take arrays of data vectorized from memory to work on them in parallel, and typically employed deep pipelining.

The rationale behind these machines was the fact that computation was very expensive. The cost of computation dominated the overall time for program completion, and so loop unrolling into vectors of concurrent operations reduced the time for computation dramatically. In these early days communication was effectively free.

2.1.2 Massively Parallel Machines

As computational requirements continued to grow, the use of small numbers of very large and powerful nodes became untenable, with the last of the supercomputers based on a vector architecture, NEC's Earth-Simulator[1], claiming number 1 on the TOP500 spot until the end of 2004. Performance barriers were reached due to limits on the speed of the transmission lines within the systems [21]. The use of a single shared memory space could no longer be scaled effectively, so distributed memory and message passing parallel programming techniques became the norm. The development of machines such as the Connection Machine CM2/5 [22], Intel Paragon [23], Cray T3E [24] and the ASCI Red/White [25] necessitated the use of more interesting topologies such as hypercubes, tori and fat-trees in order to scale beyond the limits of the simple crossbars which came before.

The first TOP500 machine to use a standard commodity Intel processor was the ASCI Red. Along with this development and driven by the standardization of message passing techniques for distributed memory systems, as well as the increasing power of consumer grade processors, large clusters of standard commodity machines begin to emerge; Beowulf clusters [26]. Systems such as these democratized supercomputing, bringing the costs of systems down significantly, causing an industry shift toward large volumes of lower cost, standardized components.

[1]We say *based* as it should also be classed as a parallel machine, as it used parallel processing among its interconnected nodes, with shared-memory scaled only within the node.

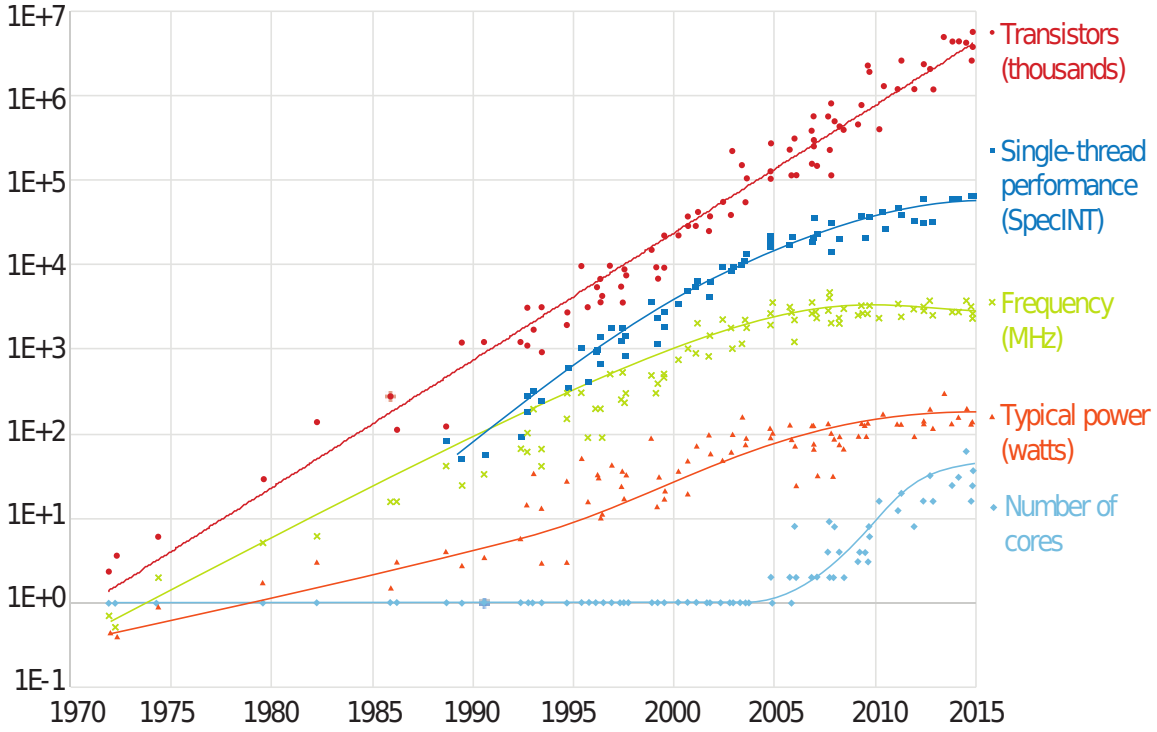


Figure 2.1: Shows the effects of post-Dennard scaling on ballooning core counts, taken from [28].

2.1.3 Many-Core and Heterogeneous Computing

Through this period until the mid-2000s single threaded performance grew very rapidly, with aggressive techniques to exploit greater *instruction level parallelism*; ever deeper pipelining, superscalar architectures, out-of-order execution, branch prediction and speculative execution, all alongside dramatically increasing clock rates. This trend could not last however, and around 2004 Dennard scaling began to break down [1]. The effects of exponentially rising current leakage that accompany transistor scaling meant that energy-density in the chip no longer remained constant. As a consequence clock speeds have stalled (see Figure 2.1) and because transistor numbers still (for now) roughly track Moore's Law the era of dark silicon has emerged [27].

Chip manufacturers began to pursue multi-core and many-core architectures as a method of exploiting *thread-level parallelism*. As this became the only way to gain greater performance in systems, the core and component count in the largest

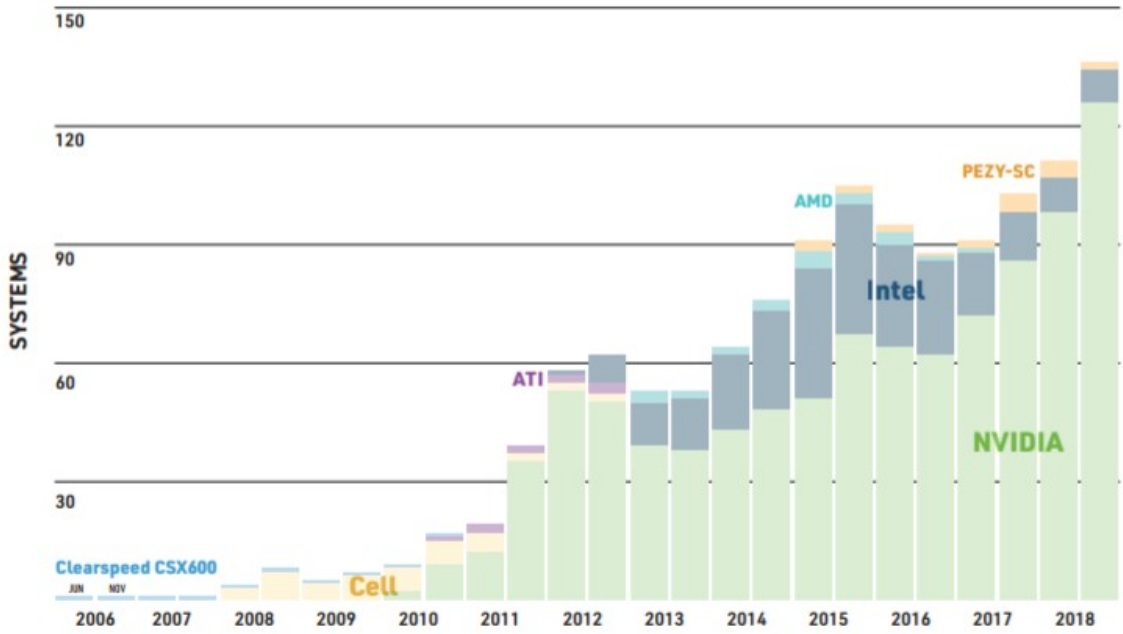


Figure 2.2: The rise of GPU accelerators within the TOP500 list. Image taken from The Next Platform[2].

machines exploded, and with it the overall power consumption. The first machine with sustained petaflop performance, the IBM Roadrunner, totalled 122,400 cores [29] when it knocked the Blue Gene/L [30] from the top spot in June 2008. Both of these machines favoured much higher volumes of modestly powered CPUs or coprocessor/accelerator cores, as opposed to fewer, large cores (Roadrunner used the IBM Cell as coprocessing elements for function offload from the main Opteron CPUs, as well as for standalone compute tasks [29]).

As the desire for ever more performance outstripped the need for simplistic programming models, the Roadrunner gave a taste of the heterogeneous CPU+accelerator solutions that have followed, and have now become fairly typical. The use of GPU acceleration has become commonplace in the largest machines, and the Intel Xeon Phi (originally designed from a GPU architecture) and Nvidia Tesla currently feature in a substantial number of the world's top machines. Figure 2.2 shows this rise in GPU accelerated computing within the TOP500 over time.

.

[2]The Widening Gyre of Supercomputing- `https://www.nextplatform.com/2018/11/12/the-widening-gyre-of-supercomputing/` , accessed April 2019.

2.1.4 Energy Efficiency and Data Movement

Unfortunately, the massive developments in performance that have been afforded to GPUs in recent years has come at a price. The power consumption of the GPU has increased dramatically along with its floating point performance and memory bandwidth [5]. While high-end GPUs are shown to be more efficient than using a CPU alone for computation, this gap is narrowed by appropriate optimizations [31] and their performance scalability must be questioned in relation to their current configuration within HPC architectures.

The main issue with these more traditional forms of architecture is the fact that many still view the GPU accelerator as a coprocessor for the CPU. While they may allow for limited inter-GPU communication directly between a subset of the accelerators using protocols such as NVLink[3], such limitations can cause additional data copying and decreased locality depending on the workload.

The current number 1 on the TOP500 list (November 2018), the IBM Summit, uses Nvidia Volta GV100 GPUs, and is currently the third most efficient system in the world, topping 14.6 GFLOP/Watt. However, despite this the system uses $\approx$9.8MW of power. Simply scaling such a system up to exascale (10^{18} FLOPS) would require around 70MW. This is well over the acceptable 20-30MW budget for an exascale machine, and infeasible in terms of both the sheer infrastructure requirements and operating costs alone.

For many years the main focus of HPC architects was to reduce the cost of computation. Data movement was not of primary concern. However, reductions in computational cost along with continued transistor scaling has far outstripped any advances in reducing the cost of data movement. We have reached a point where reducing data movement is *the* key factor for improving energy efficiency. Figure 2.3 shows that at an 11nm process the cost of a single double precision float is less than that of on-chip movement.

Given this shift toward the importance of energy efficiency, and the fact that reducing data movement is now the key to reducing power consumption in these

[3]For example in the IBM Summit- `https://www.olcf.ornl.gov/for-users/system-user-guides/summit/summit-user-guide/`, accessed April 2019.

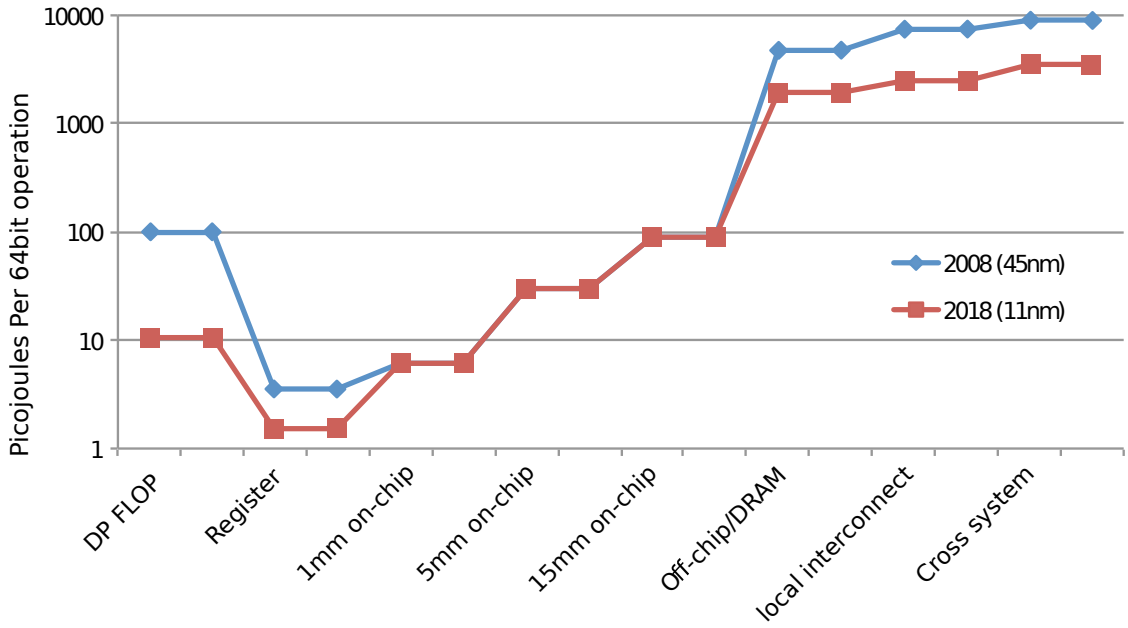


Figure 2.3: Cost of a double precision FLOP has now been reduced below the level of on-die data movement [4].

systems, the GPU's high off-chip memory bandwidth and raw floating point performance capability (see Figure 2.3) looks less important for many workloads than using techniques to reduce the power consumption by increasing data locality. In order to increase the efficiency of HPC systems, new architectures and interconnect technologies must therefore be sought...

2.1.5 FPGA

Given the discussion above FPGAs should be viewed as a promising candidate for increasing the energy efficiency of heterogeneous HPC systems. In [5] a range of example application domains are provided which show that in general the FPGA is far more power efficient than the GPU. In scenarios where performance reaches comparable levels we can say with fair certainty that the FPGA will be more efficient than the GPU, given the dramatically reduced operating frequency of the FPGA; and thus the reduced wattage. High end GPUs can consume as much as 300W[4]. FPGA power consumption is more difficult to measure (being highly

[4]NVIDIA TESLA V100 SXM2 GPU Accelerator- `https://images.nvidia.com/content/technologies/volta/pdf/tesla-volta-v100-datasheet-letter-fnl-web.pdf`, accessed April 2019.

implementation dependent) but works show them typically operating at a maximum of around 40W [32]. Table 2.1 shows a comparison of the operating wattage from a number of studies which compare the performance and/or efficiency of FPGAs against GPU accelerators (methods of power evaluation are discussed in detail in [33]). From this we can see clear performance/watt advantages in using FPGAs for a number of applications.

FPGA vendors are now becoming aware of this possible penetration into the HPC/data centre arena[5]. They are slowly addressing the gap between GPUs and FPGAs, attempting to create a more suitable environment for their exploitation. We are now starting to see advanced memory systems such as High-Bandwidth Memory (HBM) [48] integrated on the same package [49]. As well as this the floating point performance of FPGAs is increasing, with the addition of hardened floating point blocks [50], and more DSP capability within the architecture. All of these advances are key to enabling the FPGA to be fully exploited in the HPC domain.

As well as this, with shrinking transistor sizes FPGA vendors have been able to significantly increase the amount of distributed on-chip memory, providing further tools with which the FPGA can use to widen the von Neumman bottleneck [51]. Given that memory bandwidth is the limiting factor for a number of HPC workloads, reducing the number of off-chip DRAM accesses and storing data on-chip and closer to compute is not only desirable for reduced power consumption, but also as a method of increasing performance. One of the main justifications for the use of FPGA technology is the ability to use novel algorithms and custom memory layouts to optimize the number of accesses. Increasing the available on-chip memory and using multiple FPGAs is a key method to enable larger datasets to be stored closer to the compute. Numerous works have shown optimizations which can be used to increase the computational efficiency of FPGA based computing, using approaches which are not as readily exploited on the GPU. In the following Section we will review some of the work which shows the sorts of workloads which can exploit the FPGA, and we will identify the system architecture which is required to best facilitate this.

[5]Intel Launches FPGA Accelerator Aimed at HPC and HPDA Applications- `https://www.top500.org/news/intel-launches-fpga-accelerator-aimed-at-hpc-and-hpda-applications/`, accessed April 2019.

Table 2.1: Survey of comparative studies between FPGA and GPU efficiency and performance.

Study	FPGA (Watts)	GPU (Watts)	Notes on Performance
A comparison of CPUs, GPUs, FPGAs, and massively parallel processor arrays for random number generation [34].	30	178	FPGA has 3X higher performance.
Acceleration and energy efficiency of a geometric algebra computation using reconfigurable computers and GPUs [35].	8.3	170	GPU ≈2-8x higher throughput, FPGA 18.7x lower latency.
A comparative study on ASIC, FPGAs, GPUs and general purpose processors in the gravitational n-body simulation [36].	5-30	122-148	(multiple FPGA and GPU configurations) GPU 1.4x-2.8x higher performance.
BLAS comparison on FPGA, CPU and GPU [37].	≈10	≈128	FPGA performance comparable, at 2.7-293x higher energy efficiency.
Comparing performance and energy efficiency of FPGAs and GPUs for high productivity computing [38].	79-103	71-85	(multiple FPGAs on a Convey coprocessor) about 2x more efficient than GPU.
Highly parameterized k-means clustering on FPGAs: Comparative results with GPPs and GPUs [39].	15	59	FPGA 2-6.7x higher performance over GPU.
An energy efficient FPGA accelerator for monte carlo option pricing with the heston model [40].	35-40	148-310	FPGA with laptop runs 2.4-4.2x slower than GPU+server, consumes 1.7-2.5x less energy.
Floating-point mixed-radix FFT coregeneration for FPGA and comparison with GPU and CPU [41].	96-109	142-236	CPU+FPGA achieves 1.2/2.06 GFLOP/W, CPU+GPU achieves 5.5/3.2 GFLOP/W for medium/large size FFTs. .
Accelerating a random forest classifier: Multi-Core, GPGPU, or FPGA? [42].	11-13	≤225	FPGA ≈26.4-31.2x more energy efficient.
Comparison of Processing Performance and Architectural Efficiency Metrics for FPGAs and GPUs in 3D Ultrasound Computer Tomography [43].	35/35	35/235	FPGA 6-8X higher performance/FPGA within 10% of GPU performance when GPU power consumption is limited.
Optimization schemes and performance evaluation of smith-waterman algorithm on CPU, GPU and FPGA [44].	25	190	FPGA performance 3.4x higher than GPU.
High performance biological pairwise sequence alignment: FPGA versus GPU versus cell BE versus GPP [45].	100-139	70-126	FPGA has 23x higher performance per watt over GPU.
A comparison of FPGA and GPU for real-time phase-based optical flow, stereo, and local image features [46].	5.5	244	FPGA 4.6x higher performance per watt, but 9.58x lower absolute performance.
A Performance and Energy Comparison of Convolution on GPUs,FPGAs, and Multi-Core Processors [47].	40	≈250	FPGA and GPU performance comparable.

2.2 HPC Workloads

Following on from our discussion of the advancements in HPC system architectures and the requirements for reducing power consumption, we will show some application domains and workload characteristics where FPGA architecture can outperform the GPU. In doing this we further reiterate the possibility for FPGAs to emerge in next generation HPC solutions.

2.2.1 Current Suitability

Traditionally FPGAs have been viewed as excellent architectures in areas where irregular memory accesses are exhibited, or where dataflow type processing is possible, making use of much deeper pipelining and fine grained parallelism. While the memory bandwidth for GPUs is incredibly high, and is certainly important for a number of high performance applications, there are many applications which are far more sensitive to memory latency, messaging rate and overlap. These have been overlooked in the development of GPU architectures [52].

In a recent survey [9] the suitability of FPGAs for use within HPC systems is analysed. They use the Berkeley classification for HPC applications [53], which identifies 13 different *dwarves* (patterns of communication and computation) of HPC workloads. Using this prior classification they assess their suitability for execution within FPGAs, GPUs or on traditional multi-cores based upon the results of prior literature. The Berkeley paper shows that over half of the application types they classify have their performance limited by memory-latency, as opposed to memory-throughput. This sort of work is beneficial for FPGAs as they are able to make better use of custom memory layouts and reduce the external memory accesses, thereby reducing memory latency.

They show that irregular memory access patterns combined with high levels of arithmetic computation already provide for more efficient FPGA based solutions over CPU or GPU solutions. Workloads such as stencil codes are suited to the FPGA due to the high volume of on-chip memory, reducing DRAM accesses [54]. Other computations such as sparse matrix-matrix/matrix-vector multiplication are very good for implementing on the FPGA, as they feature a relatively low FLOP count

per memory access, meaning that on-chip storage for values is preferable in this situation (as in [55]).

They find that the main limitation with FPGAs is currently in their floating point capabilities, so algorithms containing high arithmetic computation combined with regular memory access patterns are best performed on GPUs (dense matrix-matrix/matrix-vector multiplication, FFTs N-body simulations etc.).

2.2.2 Advancing System Architectures

Despite these limitations, in [9] they show works which suggest that the evolution of the FPGA beyond a simple coprocessor solution–integrating hard-core CPUs with coherent access to the FPGA fabric–will allow for the effective acceleration of workloads with more complex memory handling. Examples include graph traversal algorithms and branch-and-bound problems. This inference is backed up by recent work [56] which examines pointer-chasing on modern FPGA substrates with direct, shared-memory access between tightly-coupled accelerator and CPU. They conclude that memory access latency is the main area of concern for these sorts of problems, and that interleaving memory access over several concurrent traversals can alleviate this problem.

Extending beyond this tight memory coupling is the idea the FPGA should be viewed as the main, standalone compute element within the system, capable of accessing the network as well as the main memory system. They argue for the use of custom Network Interfaces (as is also argued in [12]) to make better use of the high speed serial links in cluster setups, suggesting that with suitable work distribution FPGAs may eventually be able to compete with GPUs even on dense linear-algebra problems where the GPU excels.

There have been several prior works towards this goal; upgrading the FPGA to a more central role within the system, eliminating the CPU entirely from computation (or at least forcing all data movement through the FPGA. Examples of this sort of architecture are shown to readily accelerate cryptographic functions [57] or map-reduce type operations [58] within the network for example. Forcing all network traffic through the accelerator is known as a *bump-in-the-wire* architecture and implementations have been shown to be effective in enabling pools of FPGA resources

to be used for scaled acceleration [6]. This is discussed further in Section 2.4.

2.2.3 Accelerator Optimization Techniques

There are two main areas in which the performance and/or energy efficiency can be improved in a system which utilizes FPGA based technology which GPU based accelerators typically cannot utilize (aside from techniques such as deeper pipelining etc. leading to greater instruction level parallelism):

- Custom memory layouts, reducing the number of DRAM accesses or off-chip communications.
- Using custom data types, reduced precision and memory compression for reducing the volume of data being moved around the system.

2.2.3.1 Efficient Memory Use

There has been a lot of studies into improving data locality in HPC systems generally, using a variety of techniques and tools [59]. Several works have recently been able to show the efficacy of some of these techniques within the context of FPGAs.

In [55] they perform a set of experiments on optimized iterative solvers for large matrix-vector calculations. They first present an optimized GPU implementation which reduces memory accesses by using redundant computation. Instead of passing all the intermediate values between neighbouring cores, they simply compute some of them multiple times in multiple cores; trading off additional compute for *reduced* memory access frequency. However, in their FPGA based implementation they make use of the greater abundance of distributed on-chip memory and routing resources. They implement small FIFOs to pass the intermediate information between different processing elements, and are able to eliminate the memory accesses for these intermediate values altogether. They show that for randomly generated matrices they can get between 0.3-4.4x speedup over the optimized GPU implementation.

The authors of [60] introduce FPDeep; a framework for mapping Convolutional Neural Network Problems onto multi-FPGA networks. Their solution uses deeply

pipelined processing to spread computation over distributed FPGAs, providing a dataflow type model. In using this fine-grained pipelining their solution requires no external memory, utilizing only the on-chip memory of the FPGA. As a result they report 7.6x higher energy efficiency over a Titan X GPU with comparable performance, and 4x higher energy efficiency than a prior FPGA implementation.

In [61] they show how three common signal processing tasks can be optimized by increasing locality. They show that Gaussian Mixture models can benefit from increasing the number of processing elements, in order to place greater numbers of smaller memories closer to the compute. In a Window Filtering kernel they show how they use *line buffers* to store information from neighbouring processing elements. In many instances this data may be required for further computation, in which case they can use the data immediately rather than retrieving the values repeatedly from main memory. In an FFT they reduce the cost of memory accesses by changing the order of computation, reducing the size of the memories needed for early stages of computation. Finally they propose *Continuous Hierarchical Memory*, a modification to the architecture of on-chip memory in an Altera device, which allows the block memory to be used in one of 3 configurations in a hierarchy, reducing the energy spent on memory accesses for large memory, thus enhancing the energy efficiency.

In [62] genomic sequencing is accelerated using FPGAs. They identify techniques to enable an FPGA coprocessor (implemented on the Convey HC1) to mitigate against memory inefficiencies. The *Read Mapping* portion of their algorithm requires many non-contiguous memory accesses. These are optimized in the Convey's coprocessor by utilizing the scatter-gather DIMMs and its tight coupling to system memory (attached to the host via the Intel Front-Side Bus). These sorts of optimizations are simply not possible on the multi-core machine due to limitations of the memory controller (which is customized in the FPGA solution), and they show performance increases of around 189x compared with a single CPU solution.

In [63] they accelerate a *3D Reverse Time Migration* algorithm; a key component of which performs stencil operations. The performance of these operations is enhanced by customizing the block RAM of the FPGA in order to allow for better

data reuse on the FPGA. Typically these kernels require a lot of buffering and suffer from many cache misses which arise from accesses to the rows and columns of 3D arrays. However, they optimize the problem to fit within the on-chip storage of the FPGA by decomposing the 3D arrays into a series of smaller 2D problems, achieving a performance-scalable solution as FPGA size increases.

2.2.3.2 Reduced Precision and Custom Data Types

The GPU's high performance for floating-point operations is well established. However, there are numerous workloads which do not require such precision. The ability for FPGAs to exploit completely custom data types allows it to reduce the overheads for computation and communication dramatically. The majority of this work is for Deep Learning algorithms, which are able to tolerate reduced accuracy in results.

In [64] they assess the use of ternary weights and full single precision neurons in a sparse general matrix-matrix multiplication (GEMM). Comparing against a GPU implementation using the same reduced precision data type they deliver 60% better performance and 2.3x increase in performance/watt. The algorithm produces results with an accuracy within 1% of those using full precision data types.

The work in [65] is the first to implement a Binarized Neural Network on FPGA. They use common high level synthesis tools (Vivado HLS) to demonstrate the efficacy of these tools for increasing designer productivity. Their results demonstrate efficiencies in the design, showing a 4.5-15.2x more efficient use of FPGA resources than standard Convolutional Neural Network (CNN) implementations on FPGA. (Although they admit that this comparison may not be completely fair.)

2.3 Programming Models and Interfaces

Having shown the potential benefits that FPGAs can bring to future HPC systems for a number of viable workloads, we now assess existing programming models in use within existing HPC systems and accelerators. We will show the models which may be amenable to High Performance Reconfigurable Computing (HPRC).

2.3.1 MPI

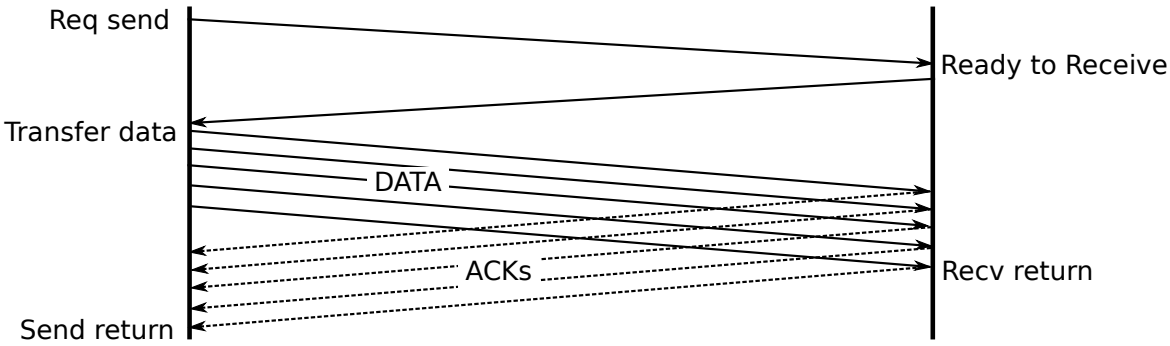
MPI (Message Passing Interface) is the ubiquitous standard for message passing communications, used for over two decades within the parallel computing community [66]. Today it is used in virtually every HPC application running on distributed systems. MPI gained popularity due to its high performance, which comes at the cost of exposing the parallelism heavily to the applications programmer, making task mapping, data distribution and (in the case of one-sided communication) synchronization explicit.

MPI communications are typically formed of two-sided send and receive operations; for which both blocking (`MPI_Send()` / `MPI_Recv()`) and non-blocking (`MPI_Isend()` / `MPI_Irecv()`) versions are available. Introduced as part of the standard in MPI 2.0 (and heavily updated in MPI 3.0 [67]) one-sided (RMA) Remote Memory Access (`MPI_Put()` and `MPI_Get()`) operations are also supported. Standard two-sided operations synchronize implicitly, as shown in Figure 2.4a, requiring the post of an `MPI_Recv()` before the data can actually be placed in the receiver's memory, indicating the receiver is ready. This model is as opposed to one-sided communication, which requires the initialisation of a *synchronization epoch* for transfer of data into abstract objects called *windows*. These are used to specify regions of memory for remote operations to target (Figure 2.4b).

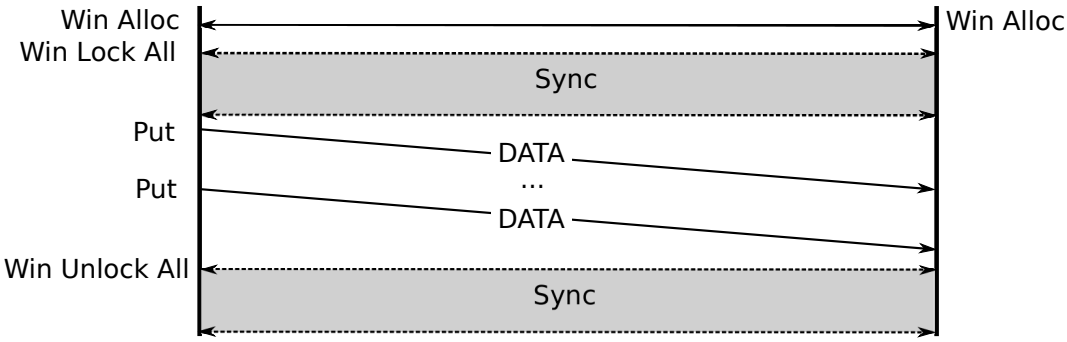
Any new HPC system produced in the near to medium term future is almost certain to use MPI as one of its communication paradigms. This is not only because of the sheer volume of legacy MPI code but also the lack of suitable alternatives. However, it is clear that continuing to scale MPI into the future is untenable, as many-core architectures and increasing heterogeneity have exposed additional levels of finer-grained parallelism to the user [68]. While MPI is highly efficient for the CSP [69] (Communicating Sequential Processes) model for parallelism, modern architectural features are not easily represented or exploited in MPI[7] and shared memory cannot be utilized within MPI ranks on the same node.

[6]MPI Performance Topics- `https://computing.llnl.gov/tutorials/mpi_performance/`, accessed April 2019.

[7]While technologies such as NVIDIA's GPUDirect can be used to expose accelerator buffers to MPI [70], it is unclear as to whether this could be extended beyond a host-accelerator architectural model, as the GPU kernels cannot act as MPI processes in and of themselves.



(a) Two sided MPI send and receive operation. Note this is a rendezvous implementation which requires pre allocation of space in the receive buffer; for short messages implementations may use an eager protocol[6].



(b) One-sided MPI put operation, shows the window allocation which must first occur, followed by a lock on the window. Following this data can be freely transferred until the window has been unlocked.

Figure 2.4: One sided and two sided MPI operations.

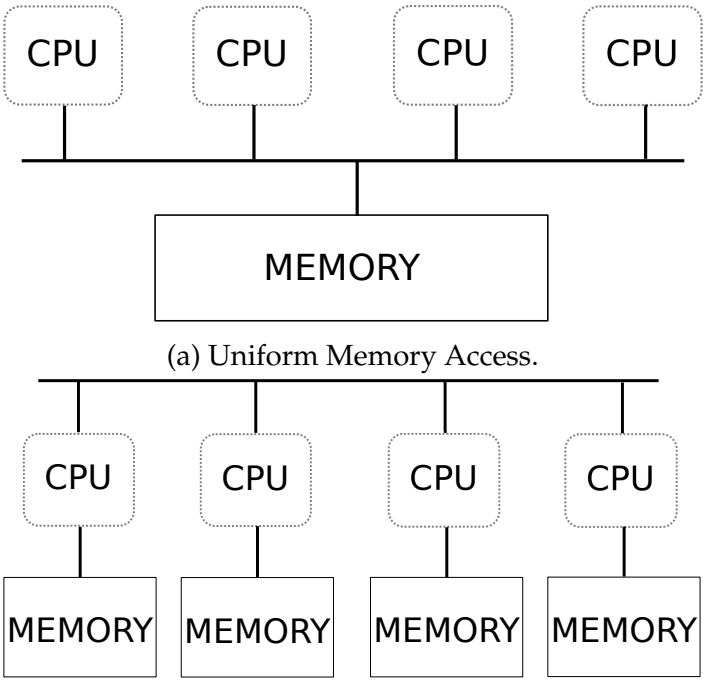
Due to this limitation of MPI on modern multi- and many-core machines, many HPC applications today run MPI+X (a hybrid model; where X refers to some sort of programming support threads[8]). This allows for the use of the shared memory within a node and message passing across the distributed memories for inter-node communication, and is very popular due to the standard model for HPC architectures over the past 20 years. (This model consisting of many interconnected nodes, with multiple processors per-node, and multiple cores per processor.)

2.3.2 OpenMP

OpenMP[9] is an API for writing multithreaded applications within a shared-memory environment. It has created a standardized method of performing SMP

[8]Compilers and more: MPI+X- https://www.hpcwire.com/2014/07/16/compilers-mpix/- accessed April 2019.

[9]For an OpenMP tutorial, see- `https://computing.llnl.gov/tutorials/openMP/`, accessed April 2019.



(a) Uniform Memory Access.

(b) Non-Uniform Memory Access.

Figure 2.5: Types of shared memory node architecture which can be exploited by OpenMP.

(Symmetric Multi-Processing), and allows for a much higher level and more portable way of multi-processing than *pthreads* for example. OpenMP consists of a set of compiler directives that can be used to simply divide computational tasks and distribute loop iterations over multiple threads of execution. It can be used in a multi-core Uniform Memory Access (UMA) and/or multi-processor Non-Uniform Memory Access (NUMA) node architecture (see Figure 2.5), but cannot be used in a distributed memory environment.

As OpenMP on its own is solely used for intra-node communications, its continued use within the HPC community will have little bearing on the wider networking solution we propose within this thesis. It can also traditionally not be used to program and exploit parallelism within FPGA accelerators, however recent works are attempting to address this:

Developed at Barcelona Supercomputing Centre (BSC), OmpSs [71] is a programming model which aims to extend OpenMP with new directives to allow for asynchronous parallelism to be exploited (implemented within the OpenMP standard as of OpenMP 4.0 [72]). By expressing tasks and data dependencies explicitly with *#pragmas*, a dataflow type execution can be expressed within sequential code.

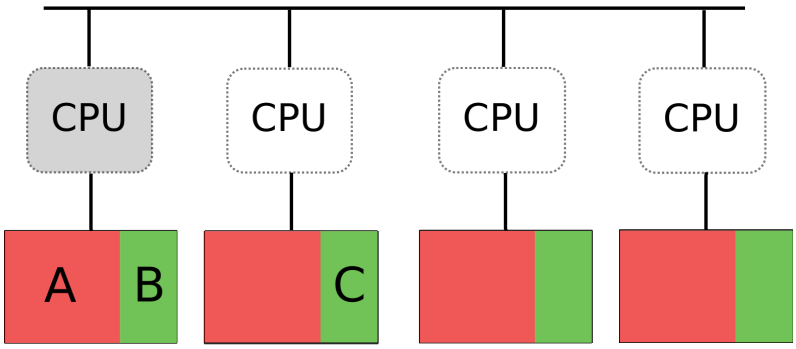
Heterogeneity can be exploited using a *target* directive, making it simple to interface with FPGA accelerators. While this is a step forward in heterogeneous parallel programming it does not allow for distributed accelerator resources to be exploited, only those within the local shared-memory of the processor [72]. The target cannot be a remote distributed accelerator.

2.3.3 Distributed Shared-Memory (NUMA/PGAS)

Initially proposed and developed around same time as the MPI standard, Partitioned Global Address Space (PGAS) languages aim to combine the performance and data locality provided by MPI with the much simpler programmability of a shared memory model. PGAS languages such as Unified Parallel C (UPC) [73], Co-Array Fortran [74], and Titanium [75] provide language extensions for explicit parallel programming to C/C++, Fortran and Java respectively. Whereas other PGAS languages such as X10 [76] or Cray's Chapel [77] have been built from scratch. Library based implementations of the PGAS model also exist, such as OpenSHMEM [78].

PGAS languages allow for *private* memory, visible only to the local process, and *shared* memory which is made accessible to some or all of the processors within the system. They also introduce the concept of places (memory affinity), which is exposed to the programmer in order to allow them to explicitly give the compiler information about which data is where on the system. Since the data is either *local* or *remote*, the relative speed of memory access (NUMA awareness) is known using a cost function, where remote accesses are obviously more expensive. Accesses to remote memory can therefore be performed by simple references while ensuring that differentiation between local and remote memory is explicit. Of course this is only a general description and the many PGAS languages provide different models of execution, access cost, data distribution and access [79]. An example of memory access and affinity using UPC as an example is shown in Figure 2.6.

There has been a lot of recent interest in the potential for the PGAS model within future HPC systems as means of overcoming the challenges associated with MPI, and the restrictions with OpenMP [80]. As the memory accesses are naturally one-sided, PGAS models more easily eliminate issues with overlapping computation




```
int A;              /* private local variable */
shared int sv[4];   /* variables in shared space (1 per thread) */
int B = sv[0];      /* access shared variable with local affinity */
int C = sv[1];      /* access shared variable with remote affinity */
```

Figure 2.6: Example of different remote and local variable accesses in UPC.

and communication [81], increasing performance. There are also major concerns regarding the memory footprint of MPI. As memory scaling has not kept pace with the scaling number of cores, the memory per-core has been decreasing. Since many of MPI's functions and data structures scale linearly with the number of processes, this problem is amplified as the number of cores and processes increases [13].

Adoption of PGAS languages has been slow however, due to a lack of direct hardware support for the communication primitives that PGAS requires; the ability to write directly into global address spaces. This is as well as a lack of significant enough performance benefits to merit the arduous task of rewriting legacy code. Although, this is not to say that there are not clear benefits to the PGAS model over message passing. Several works have shown the benefits which arise from one-sided communication, which the PGAS model is much more suited for than MPI [81], [82].

Due to these facts, many believe that future systems will use PGAS models in a hybrid setup to compliment MPI[10], as this will allow programmers to exploit the benefits to scalability and programmability that PGAS can offer without having to rewrite entire code bases [83]. This shift is evident from the push to further develop the one-sided communication capabilities afforded in MPI 3.0. However, there are advantages to using dedicated languages for PGAS support over the library based MPI. One main advantage is the simpler programmability of PGAS languages when

[10]Melding Hyperscale and HPC to Reach Exascale- `https://www.nextplatform.com/2016/08/10/melding-hyperscale-hpc-reach-exascale/`, accessed April 2019.

compared to MPI's RMA (Remote Memory Access) semantics; requiring the need for explicit communication management within MPI [14].

The first such hybridization of MPI and a PGAS language came in [14], where they present a programming model to support MPI and UPC. Given that the MPI implementation is as a library, and the UPC language is an extension to C, they can compile their code normally and simply link to the MPI library. Their hybrid MPI+UPC shows far superior scaling over a simple global shared space. Under normal circumstances the number of non-local memory references is detrimental to the performance of UPC on its own, particularly as the number of processors scales. However, using the MPI+UPC approach they are able to replicate the data over multiple nodes as opposed to distributing it, and so they increase the number of local accesses.

In [84] they measure the performance of a "Concurrent Search" kernel from the Graph 500 benchmark[11]. They compare a standard MPI implementation against their hybrid MPI+PGAS runtime over Infiniband, and show a 59% reduction in the execution time over the standard MPI model. In this instance the communication patterns are identical for both hybrid and standard MPI, showing the benefits of using a one-sided communication strategy. Their solution also highlights the positive scaling characteristics of using a hybrid MPI+PGAS model.

2.3.4 FPGA Programming Techniques

Whilst we have discussed common programming techniques for the CPU, we have yet to discuss the FPGA programming techniques which will likely be employed. Traditionally the main hurdle in the uptake of FPGAs for HPC (or any other domain) has been programmability, with traditional Register Transfer Languages (RTL) having a steep learning curve, long development time and requirements for specialist knowledge of the hardware. Over the past few years there has been dramatic advances in the capability of High Level Synthesis (HLS) techniques, with many new languages (BSV [85], Chisel [86], System-C [87], MaxJ) and toolchains (Vivado HLS [88], Intel's HLS compiler [89], FCUDA [90]) being introduced. The vast majority of these HLS tools target Verilog/VHDL as their output, requiring the

[11] Graph 500 Benchmark- `https://graph500.org/`, accessed April 2019.

use of standard synthesis tools to create the final bitstream for the hardware. Their development is regarded as vital for increasing programmer productivity, and its uptake has caused RTL to be viewed as the assembly language of the hardware world; with respect to it's abstraction level.

OpenCL [91] is an open source framework for programming heterogeneous systems, and is one of the most established methods for programming GPUs. Altera (now Intel) have pushed for OpenCL as the preferred way to create FPGA based accelerators, with an SDK and compiler supporting development. However, OpenCL (like many other HLS solutions) offers only a host/device model of programming. The code is split into two portions; the *host* code which runs on the CPU, and the *device* or *kernel* code which runs on the accelerator. An API allows the host code to use the hardware kernel, loading data into the hardware for execution and then transferring the data back out to the host once computation has been completed.

This model therefore does not typically extend beyond the scope of a single node, meaning that the use of distributed acceleration resources requires additional memory copying and CPU intervention. However, there are works attempting to address this. The ECOSCALE project [92] is one such example. They aim to extend OpenCL and create a runtime system which will enable the aggregation of multiple hardware devices, allowing multiple accelerators to work as a single compute element spanning multiple FPGAs. Groups of hardware blocks form workers, whose memory is formed into a Partitioned Global Address Space (PGAS), allowing communication via direct load/store operations or via DMAs.

Vivado HLS is another common tool from Xilinx. As opposed to OpenCL, which provides an API to manage calls between the host and accelerator kernel, Vivado HLS uses a set of pragmas and directives, which indicate to the compiler how standard C/C++/System-C should be compiled into regular RTL. It provides abstractions for data type, algorithm and interface, and is used to develop IP blocks which can then be integrated into other designs. Using OpenCL for the kernel development (within Xilinx tools) can actually use Vivado HLS to generate the IP for the kernel, if the kernel is being synthesised directly from C/C++.

The main concern with respect to the development of our interconnect is with respect to the interfacing methodology, and to ensure that these standard tools do

not require heavy modifications in order to enable the FPGA logic to initiate network transfers. We will discuss this in the following Section.

2.3.5 Extending Models to FPGA

Given the discussion above, we see that facilitating both MPI and PGAS like communication models will be incredibly useful for future HPC systems. The vast majority (if not all) modern interconnect solutions intended to run MPI over the top do so using Remote Direct Memory Access (RDMA) as the primary vehicle for communication. Infiniband [93], iWarp [94], RoCE (RDMA over Converged Ethernet) [95], Blue Gene [96], Tofu [97], Cray's Gemini [98] etc. all facilitate MPI communications via the use of RDMA, thus reducing latency and freeing the CPU from the burden of communication. It is necessary and desirable to support this in the Network Interface (NI), and multiple examples of RDMA enabled networking solutions already exist for FPGA, showing the viability of the model [99]–[101].

As we have identified, one of the main barriers for uptake of the PGAS model is the fact that direct hardware support is often lacking in most common interconnect solutions. Attaching the Network Interface as a PCIe peripheral device and requiring DMA initiation precludes the use of transparent memory accesses being issued through the network, which could be used to improve the latency of transfer dramatically for small messages.

While RDMA can offload much of the processing for network transfer to the NI, for small transfers it still provides unnecessary overheads when compared with a simple load/store operation issued to a global, distributed address space [15]. Smaller transfers are more likely to be latency critical transfers such as synchronization and control messages, whereas larger data transfers can be left to RDMA. If we know that the latency critical messages will be small, reducing the latency of smaller messages becomes paramount. The difference in latency between an RDMA transfer and a direct message of a single word length can be shown thusly:

$$T_{RDMA} = T_{storeN} + T_{DMAInit} + T_{MR} + T_{transfer} + T_{ack}$$

$$T_{SharedMemoryOp} = T_{store1} + T_{transfer} + T_{ack}$$

Where *MR* is the word read from main memory, *DMAInit* is the time taken to initiate a work item to the DMA engine from the NI, and *store* is the time taken to write a given number of words into the NI, where N is the number of words in an RDMA work item (transfer descriptor).

Providing direct hardware support for PGAS memory requests can massively improve the performance of such systems. Cray's work on their Gemini Interconnect shows this, where they provide a *Fast Memory Access* mechanism which permits the use of native CPU stores into the NI. They indicate that for short messages the speedup when using one-sided communication can be as much as 5-10x [82]. The Aries Interconnect is an enhanced version of Gemini, and is discussed in greater detail in Section 2.5.5.1.

We therefore propose a similar method to that in the Cray Gemini/Aries interconnect [102], where small low latency transfers have a special, faster path through the NI. Upon initialization a process is allocated a descriptor, which when written to will send the data to the address held by the descriptor. Our mechanism differs from this in that we simply accept read and writes to any address through the system, and a transaction destined for a remote location requires translation in an MMU (Memory Management Unit), as opposed to holding static addresses in these pre-allocated descriptors.

Older work at Cray on their T3D shared memory system [103] performs similar functionality, where they separate the methods of communication based upon the volume of data being transferred. They use a standard shared memory, global address space method for small transfers, but once the transfer size reaches a certain threshold they use a *Block Transfer Engine*, which is essentially a DMA device. The threshold for using this engine is 7900B, which is the point where the latency for sending through the normal channel becomes greater than the initialization time for the Block Transfer Engine.

Other previous work allowing this direct shared-memory communication is presented in [104], where they have developed a Network Interface which allows local load/store transactions in the HyperTransport protocol to be forwarded and sent over a custom network to be received by remote nodes. By using the HyperTransport protocol directly over the network and between the network controller

and host they avoid any bridging or protocol conversion. They maintain coherence within the node, but by not allowing global coherence their solution forms an ideal interface for PGAS models.

With regards to the FPGA, in order to direct inter-FPGA communication between local and remote accelerator logic the use of hardware support within the Network Interface to allow for RDMA and simple load/store transactions makes a lot of sense; for programmability and performance. This sort of model provides a natural way for accelerators to orchestrate work among themselves, as these models of communication are frequently used between local CPU and accelerator in more traditional architectures.

For example, Xilinx's Vivado HLS tool [88] used to synthesise C code into RTL fits this model well. It provides a *memcpy()* function which basically acts as a DMA, pulling data in and pushing it back out when a descriptor is passed to it. It also enables simple memory mapped accesses as pointer dereferencing, meaning that the FPGA fabric can easily pass transfer descriptors to remote accelerators if every component within the system is viewed as a global, shared memory location. Having the whole network rely on direct memory addressing for global communications (as opposed to using IP addresses for example, or via PCIe transactions) means that the performance is enhanced as no protocol conversion is needed, and no additional memory copies are required. It also ensures that the minimum of extension to traditional FPGA programming models will be required, as OpenCL Kernels or Vivado HLS is perfectly capable of forming hardware with this functionality already.

2.4 FPGA Clusters

There are examples of the use of FPGAs as accelerators within HPC systems dating back over a decade. The vast majority of these systems are limited however; providing limited scope for use in much larger scale systems or beyond a very narrow set of applications. This stems from a number of historical issues. One of these is the comparatively limited capacity of older FPGAs, preventing the development of more complex on-chip networking solutions and cementing the bus-based coprocessor model. Another reason is that only recently has the development of mixed

hard-core CPUs and FPGA on the same die become available, providing much tighter coupling between these resources and eliminating the need for costly and latent off-chip protocols between them. A third reason is the programmability issues surrounding FPGAs. Traditionally these devices were inaccessible to software developers, requiring dedicated hardware engineers to write custom RTL. However, there have been many recent advances in High Level Synthesis techniques which have reduced this burden somewhat.

In this Section we will review a series of solutions for HPC clusters, and demonstrate that they are not viable for our requirements. We frame this around the system architecture; which is only now reaching a point where we are able to have both tight coupling with system memory as well as a completely standalone solution where the FPGA is regarded as a full peer within the network. Both of which we deem essential for the uptake of FPGAs within the HPC arena.

2.4.1 Early Examples

Early examples of systems including FPGA technology include the SRC-6, the Cray XD1, and the SGI-Altix. These machines were the first to attempt to overcome the problems with system integration between the FPGA and host CPU. The Cray XD1 contains an FPGA connected to the host via a simplified HyperTransport interconnect, and contains a separate memory bank with closer proximity to the FPGA than the host memory [105]. The SRC-6 has a board containing two FPGAs connected to the host via a separate network. The FPGAs cannot communicate directly with the host memory, however work can be distributed between the two FPGAs to expand the capabilities of the accelerator [106]. SGI's Altix, containing their RASC (Reconfigurable Application Specific Computing) module [107] is among the first systems to bring tight coupling between the host and accelerator memory systems, giving shared NUMA access via the NUMAlink [108] system interconnect. Notwithstanding the scalability issues of a fully coherent shared memory system, this early system allows for completely disaggregated scaling and tight coupling between CPU and accelerator components using the reconfigurable element as a separate module. Unfortunately FPGA technology and programmability at the time was far more limited and the potential benefits of this early architecture

were never fully explored. This is especially true as the reconfigurable element is supplied as a separate module; the system was mainly sold and run as a standard non-accelerated cluster.

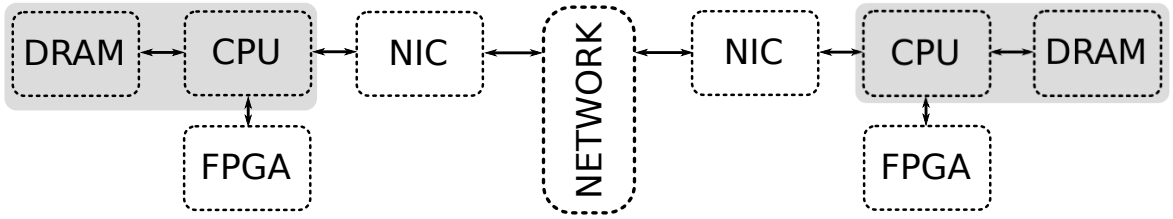
2.4.2 Bus-Based Coprocessor

A key barrier in the development and exploitation of distributed FPGA resources within the context of HPC systems is the traditional use of the FPGA as a mere co-processor [109], loosely coupled to the CPU and network resources—attached via PCIe or other equivalent bus-based interconnect (see Figure 2.7a). This architectural model exacerbates the limited off-chip memory bandwidth of the FPGA by distancing the accelerator from the main memory hierarchy of the CPU. In addition it severely limits the feasibility of dataflow processing among distributed FPGA resources because they depend on the CPUs for performing network transactions.

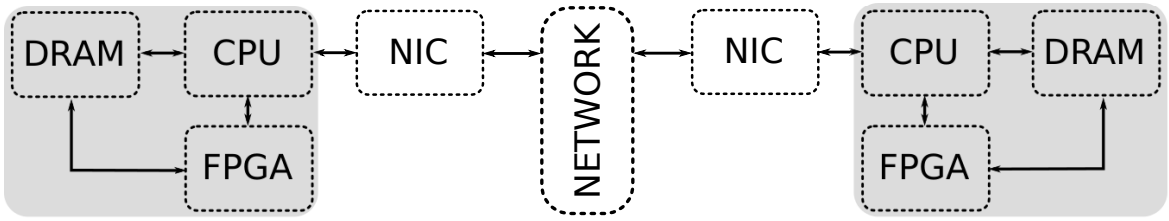
QP [110] is one such system which uses this model. They provide a node which contains two Opteron CPUs with GPUs and an FPGA attached as coprocessors on separate PCI buses. In this instance the FPGA is only connected to one of the two Opteron CPUs and must communicate through the CPU for any networking capability.

The vast majority of example systems which fit this coprocessor category use multiple FGPAs within a node/blade/rack, providing a dedicated separate network to interconnect the FPGAs. Higher level communication must be directed through the CPU. While these systems may allow for scaled applications over multiple FPGAs, they are typically limited internally by point-to-point topologies, limiting the scope of applications somewhat. In this scenario the distributed FPGAs are used in such a manner that they effectively create one large FPGA distributed over multiple chips, as they are still often attached to a single CPU.

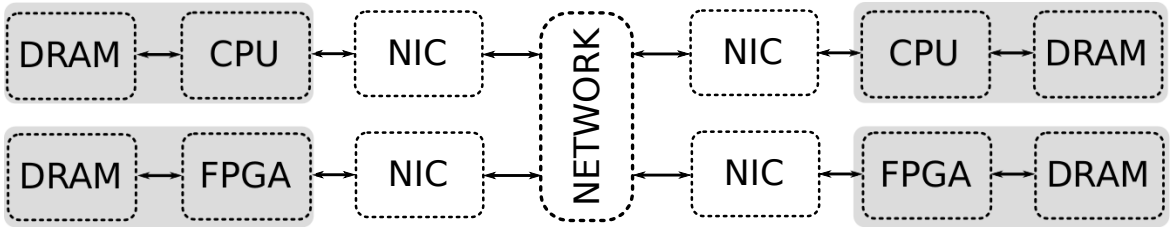
A prime example of this sort of architecture is Maxwell [111]. This is a proof of concept for a general purpose FPGA based supercomputer, comprising Intel Xeon CPUs and a total of 64 Xilinx FPGAs configured in a 2D torus, with CPUs and FPGA connected via PCI. This solution obviously suffers the many pitfalls discussed regarding a coprocessor architecture with regards to communication being directed through the CPU for network operations beyond a given scale. It allows only point



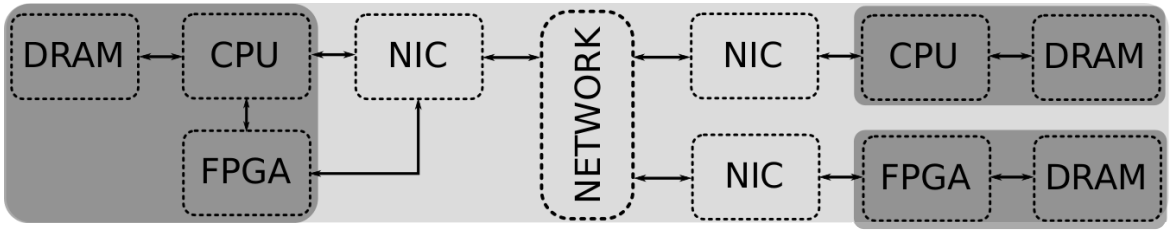
(a) FPGA attached as a bus-based coprocessor (e.g. PCIe).



(b) FPGA attached via system-bus, able to access local memory directly but not the network.



(c) Disaggregated FPGA, able to act as a full network peer but unable to address remote memory directly.



(d) Global address space in which the system bus protocol is extended over the network. The FPGA can act as a full network peer, access remote and local memory directly, and can also be scaled independently.

Figure 2.7: Possible system architectures and FPGA configurations. The shaded regions represent the limits of the addressability from a given node.

to point connections between the FPGA for parallel communications, severely limiting the topology.

Another example is Novo-G# [112]; a system based on multi-FPGA servers with a host CPU. While the FPGAs within a server can communicate directly with one another over a 3D torus network, inter-server communication is made through the CPU via PCIe, and then via Gigabit Ethernet or Infiniband. This limitation means that direct communication between FPGAs is again limited in scale. RCC [113] is another project which enables direct communication amongst FPGAs. Out of this project came the AIREN network [114], with a multi-gigabit RocketIO network for inter-FPGA communication (limited to point-to-point communications), and a separate network to connect a host PC. Like us they propose to use essentially a wrapped internal protocol for communication between FPGAs, allowing transparent use of the network from the user perspective (the only difference is latency for communication). However, their limited functionality in the network inhibits scalability.

2.4.3 System Bus Attached

Modern FPGA devices such as Xilinx Zynq Ultrascale+ and Intel Stratix 10 (complete with integrated hard-core processors including IO Memory Management Units) allow for the configuration seen in Figure 2.7b, where shared memory and cache-coherence is possible between the CPU and FPGA. While this tight coupling allows for lower latency transfers between local accelerator and memory, it does nothing inherently to alleviate the overhead of the cumbersome software networking stack. Typical methods such as TCP/IP are required to provide reliable transfer of data with costly memory copies between network layers.

Larger systems following this model have been designed to use the system bus of the processor in order to couple them far more tightly with the memory system of the CPU. Systems such as the Cray XD-1 [105], which uses AMD's HyperTransport fall into this category. The work of Ling et al. at Intel [115], which uses the Front-Side-Bus, also provides this sort of architectural configuration. They allow for direct access into the system memory of the CPU, allowing for much higher throughput between system memory and accelerator. However they still require the CPU to

initiate network transfers, meaning that the use of dataflow style processing over multiple FPGAs is inhibited by traditional networking techniques. Other works in this domain include LEAP [116], which creates coherent memory between multiple FPGAs, as opposed to between host and accelerator. Although their work is more concerned with the on-board caching mechanism than with the networking, and describes only simple chip-to-chip communications. The Convey HC2 [117] is another machine which allows coherent access between a single host and FPGA, paired as a tightly coupled coprocessor. Again though communication beyond the node must be performed through the CPU.

2.4.4 Disaggregated Network Peer

Like others have argued [8] we see that the remedy to the issues with the architectures discussed above is to promote the FPGA resources to the status of a full peer within the network, capable of issuing its own reliable transactions as well as being able to process inbound network traffic directly. In enabling the FPGA to perform RDMA operations directly and offloading traditional networking stacks into hardware (e.g. TCP offloading) this gives rise to the configuration seen in Figure 2.7c. Here we see that the FPGA is fully disaggregated from the CPU resources, meaning that FPGA resources can be scaled without increasing the corresponding number of CPUs. However, in this setup the FPGA is unable to exploit a lower latency, shared memory model with other distributed memory spaces; a property which is vital for many workloads and for providing the FPGA better control of the data flow. All this is without mentioning the significant scalability and complexity issues associated with TCP offloading [118] (discussed in detail in Section 2.5.2).

Recent work at IBM created a network attached FPGA system, which completely disaggregates the CPU from accelerator resources [119], [120]. This is done in order to allow CPU and FPGA resources to be scaled independently in data centres. They use a hardware offloaded transport layer in order to allow the FPGA to communicate directly with the network. However, this means that all communication must traverse the full networking stack. As the intention for this type of architecture is data centre applications there is no possibility to perform NUMA type accesses, and therefore tight coupling between the FPGAs and CPU memory hierarchies is

not supported.

In [121] they have created a customized transport layer for data-intensive applications on distributed FPGAs. The transport sits over the top of the Aurora PHY, and offers in-order lossless delivery of data. The protocol is very simple and uses static dedicated routing paths for a 2D mesh or torus topology, as such it is unclear whether this solution can be scaled beyond the fanout of the single router they propose.

The work in [11] presents an incredibly low latency communication protocol for direct inter-FPGA communication, achieving an impressive per-hop latency of 272ns. This comes at the expense of dramatically reduced scalability however, with no guaranteed reliability mechanism and only a simple ring topology possible.

2.4.5 Bump-in-the-wire

By far the largest and most sophisticated work so far in this domain is Catapult2, developed at Microsoft. Fitting primarily into category of a disaggregated solution, their system allows for FPGAs to communicate among themselves at the cloud scale [6]. The FPGAs are situated as a *bump in the wire*, placed between the CPU's NIC and a Top-of-Rack switch, and are used for in-network processing or local acceleration. They implement *Lightweight Transport Layer* (LTL) in the fabric of the FPGA which supports the free flow of TCP/IP traffic from the CPU into the network and uses simple UDP frames for inter-FPGA communication. While the UDP traffic uses lossless traffic classes it can never be *guaranteed* to be lossless in the switches. As such their LTL offers guaranteed packet delivery and ordering at the endpoints. They effectively turn UDP into a connection based protocol, keeping persistent connection tables at the sender and receiver, along with dedicated retransmission buffers to store unacknowledged packets.

A huge benefit of this type of architecture is that all network traffic is directed through the FPGA, allowing for in-network processing. This could be in-flight processing directed towards remote nodes (for example encrypted network traffic for deep packet inspection), processing for network attached or local hyper-converged storage solutions, or to create simple dataflow engines over distributed resources.

This Near Data Processing (NDP) [122], [123] is seen as a promising way to reduce the data movement within systems and tackle the growing prevalence of data-intensive workloads, thereby enhancing the performance and reducing power consumption. By enabling a reconfigurable fabric to take control of the network, the storage capability, local memory or caches, near-data processing which is tailored to specific applications can be achieved.

While this architecture has many desirable features such as placing the processing capabilities directly into the network and allowing reliable transfer for distributed FPGA computing at scale, there are several drawbacks which make it unsuitable for use in general purpose HPC. The main drawback is the fact that CPUs still use the traditional stack of TCP/IP for reliable data transfer. Their solution includes a dedicated NIC, which may offload functionality to the hardware, but it is still likely to suffer from the scalability and/or performance issues related to TCP in general. Another limitation is that there is no possibility to perform NUMA type shared-memory accesses, and therefore tight coupling between the FPGAs and CPU memory hierarchies is not supported. They state explicitly that this is not their aim however, as they note that tight coupling to reduce the latency of these transfers is unnecessary for them. This is because their main target application– ranking for the Bing search engine– requires only simple memory sharing [124].

2.4.6 Global System Addressing

Our work solves these issues by creating a Network Interface (NI) which sits in the fabric of the FPGA, leveraging a custom network protocol where the target node addresses are seen simply as the upper regions of a fully global memory space. Our desire to enable close system bus coupling with the FPGA along with a fully hardware offloaded transport layer enables us to reach the configuration shown in Figure 2.7d. In this instance the FPGA can act alone as a fully disaggregated peer on the network, but can also write directly into a shared memory space between the CPU and other resources (local or remote). This opens up the possibility for fine grained acceleration across distributed FPGAs, providing maximum flexibility in the architecture. By using standard memory mapped AXI transactions to issue reliable inter-FPGA communications with no complex API we enable the simple

development of distributed FPGA compute using readily available HLS tools.

2.5 Interconnection Networks

The interconnect is obviously one of the most important components within massively parallel architectures, and is fundamental to the overall performance of applications running on it. This is particularly true given the shift toward data-centric workloads and the current focus on power consumption (data movement has been shown to account for up to 40% of the power consumption in HPC workloads [125]). The network protocol and transport mechanism is an important component of the interconnect, as it dictates things such as buffering requirements, possible topologies and scalability. In this Section we review some of the most popular interconnect technologies used in HPC and data centres today, and discuss their feasibility in relation to FPGA based HPC systems.

2.5.1 Ethernet

Ethernet networks form the backbone of internet and have featured heavily in the TOP500 list since the rise of commodity clustering, simply owing to ubiquity of the technology. With 25G, 100G and now 400G link technology[12], Ethernet networks continue to be used for HPC despite serious issues; with solutions such as RoCE [95], iWARP [94] , TRILL [126] and TCP Offloading aiming to fix some of the problems associated with Ethernet based interconnect solutions.

2.5.2 TCP

The Transmission Control Protocol (TCP) [127] is the de facto standard for reliable communications over Ethernet networks. It is a connection based protocol which provides mechanisms for ordered and error-checked message reception, message segmentation, and retransmission of lost packets using an end-to-end sliding window protocol, which in addition establishes a congestion-control mechanism to help ensure reliable delivery. It provides an abstraction of the network for

[12] EETimes, 400G Ethernet is Here, as are Other Speeds- `https://www.eetimes.com/author.asp?section_id=36&doc_id=1333680#`, accessed April 2019

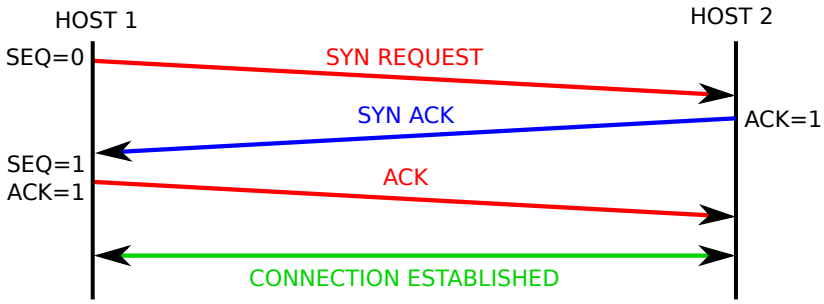


Figure 2.8: Setup for a bi-directional TCP connection for `send()` and `rec()` socket based reliable data transfer.

the application layer, obfuscating details of network congestion, packet loss and such from the programmer. Typically implemented in software, the design of TCP is focused on reliability and portability over performance. As such, traditional implementations are highly unsuited for the types of dedicated, "lossless" and low latency networks required for HPC applications.

2.5.2.1 Establishing a TCP Connection

A bidirectional TCP connection is set up using a three way handshake (shown in Figure 2.8). Firstly both sender and receiver must indicate that they wish to establish a connection (SYN). One side is the active participant in this, sending a message to the passive participant which includes the desired sequence number from which to start counting incoming data packets for ordering. The passive participant then sends a packet back to acknowledge (SYN-ACK) this synchronization stage, with its own starting sequence number for the reverse direction of traffic. One final message must then be sent to acknowledge (ACK) the sequence number for the other direction. Only then is a connection established. The sequence number (SEQ) at each side is incremented as bytes are sent, and the ACK number at each side is incremented as bytes are received and acknowledgements of these received bytes are posted to the sender.

2.5.2.2 Retransmission

Due to the ordering restrictions on the transfer buffers and the desire not to acknowledge every network packet, additional latency is seen on retransmissions. Upon the receipt of a negative acknowledgement or in the event of a timeout, the

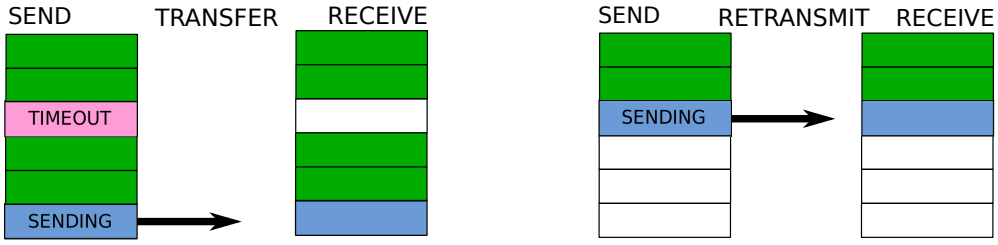


Figure 2.9: TCP retransmission strategy. In the event that a NACK or timeout is received, retransmission must begin from that point, regardless of whether following transactions have been completed successfully.

transfer state is discarded and transmission begins again from the first segment following on from the previous acknowledgement, this is described in Figure 2.9. Not only does this require resetting the transaction but it can cause duplicate packets. I.e. packets that were received properly from deeper within the transmission buffer cannot be used, and must be discarded by the receiver and resent after the lost packet is received. The retransmission mechanism we implement in this thesis requires only the corrupt packets to be resent. Although retransmissions are rare, the best-effort nature of Ethernet switches means that retransmission is a more common occurrence than in a lossless fabric, as packet drops occur on buffer overflow.

2.5.2.3 Latency and Memory

The latency of setup for connection establishment is a huge issue for short transfers. As such it is typical in HPC contexts to establish these connections at the initialization phase of the program and then leave them open for the lifetime of the application. However, in this instance the memory usage can become an issue. Connection state information is held for each connection, along with dedicated send and receive buffers for each connection pair. This means that many thousands of concurrent connections can create a heavy burden on the system. Our proposed solution only requires setup of the RDMA buffers at initialisation, after which (assuming proper locking and synchronization methods are used) any node is free to write into any of these buffers for the duration of the application lifetime.

Not only does the use of send and receive buffering create scalability issues with

respect to memory, but this also incurs a heavy latency penalty. Additional memory copies are required to the connection buffers. In typical implementations system calls are required to transfer data, with context switching and memory copies between user-space and kernel space adding additional latency. As TCP uses dedicated send and receive buffers and reassembly queues at each end, copying between these buffers and back into user-space adds significant latency to the software stack.

2.5.2.4 Offload Engines

Over time, Wide Area Networks (WANs) have grown in scale and complexity, and so the complexity of implementing TCP/IP stack has increased proportionally. The code for TCP often reaches thousands of lines[13], with many system calls needed over a number of software layers, and a large number of data copies being produced [128].

There are many solutions which attempt to offload the TCP protocol into dedicated hardware in order to eliminate many of the software overheads associated with excessive memory copying and context switching within the TCP software stack. These are known as TCP Offload Engines (TOEs), and either implement the full stack (full-offload) [118], [129], or only given functions to suit a more specific purpose (partial-offload) such as performing checksumming or TCP segmentation [130].

As the TCP protocol is considerably complex the area overhead of implementing a full TCP offload engine on an FPGA device is significant. Full offload into hardware is only an option for a very narrow class of applications. The complexity lies not only in offloading all the features of TCP, but in the heavy memory burden which is placed upon the system in maintaining dedicated per-connection buffers and state information. As such the only FPGA implementations for full TOEs that we are aware of are used for financial trading. In this latency-critical domain the number of possible simultaneous active connections is heavily sacrificed for the benefits of full offload. This dramatically reduced scalability is at odds with the

[13] An implementation of the TCP/IP protocol suite for the LINUX operating system- `https://github.com/torvalds/linux/blob/master/net/ipv4/tcp.c`, accessed April 2019.

needs of HPC applications.

Attempts have been made which seek to increase the scalability of TCP offloading. In [118] a solution is proposed to overcome the issue and their approach allows for over 10,000 simultaneous connections. However, this connection based approach still suffers massive memory utilization. They require external session buffers in DRAM amounting to 1.3GB for 10,000 sessions. Without a huge dedicated RAM for the offload engine this is extremely wasteful in terms of both memory usage and memory bandwidth. More recent work by Sidler et al. [131] reduces this footprint to 650MB by using optimizations which tailor the stack towards data processing applications for a data-centre environment. However this is obviously still huge, and is well beyond the on-chip memory capacity of state-of-the-art FPGAs. Due to the fact that we propose not to use these connection based techniques, scaling the number of communicators does not scale the memory overheads associated with our solution.

2.5.2.5 IP Routing

Ethernet networks typically route using the Internet Protocol (IP). Routing in IP requires tables to be established to associate a destination IP address with a given output port. A route between source and destination may have a number of different paths to choose from, and because the route may contain cycles the spanning tree protocol must be used in order to prevent broadcast storms and bridge loops. This creates a logical topology on top of the physical topology. Unfortunately this creates a limitation in that each source-destination pair can only follow one route through the network. While allowing for fault tolerance (if a link goes down another path can be set up), it does not allow for full exploitation of path diversity. Our arithmetically routed switch [32] used in this thesis allows for fully adaptive routing at the packet level, meaning that we can take full advantage of the path diversity of our chosen topology.

In practise what this limitation means for IP is that some topology types which promote high path diversity will perform suboptimally when using TCP/IP. There have been efforts to combat this issue and allow for multipath routing, but their adoption and success has been limited. Recently the IETF (Internet Engineering

Task Force) have developed guidelines for multipath TCP development [132], but actual implementations are scarce. At the link-level, multipath routing has been proposed by the use of TRILL technology [126]. TRILL (TRansparent Interconnection of Lots of Links) was created to avoid the pitfalls of the spanning-tree protocol, and allows for layer-3 routing techniques to be used over layer-2 links. Unfortunately adoption for this technology has also been *very* limited, and is supported by very few vendors.

2.5.3 UDP

UDP (User Datagram Protocol) is the connectionless transport layer used for sending unreliable data on top of IP. It provides only a very basic transport; the packet header contains only source/destination IP address and port numbers, data, length and a checksum. The header is a mere 8B, compared with the 20B TCP header. The protocol is unreliable, providing no guarantee of delivery for packets. It does not have the concept of acknowledgement, retransmissions or timeouts. As such any reliability mechanism not requiring direct user intervention would need to be implemented at the library level. There are a couple of examples of this sort of work in the literature. In [133] they introduce LA-MPI, which allows for retransmission of unacknowledged and timed out message fragments within MPI running over UDP/IP, Quadrics [134] or Myrinet [135] networks. However, as UDP is an unacknowledged protocol it is difficult to see how this solution can be employed for one-sided MPI communication.

There has also been work which adds reliability into MPI running over Infiniband's Unreliable Datagram (UD) transport [136], and they have actually shown that their software reliability can actually increase the performance of a molecular dynamics application by up to 25% using the UD transport instead of Reliable Connections (RC). This is due to the number of simultaneous communicators becoming a bottleneck, needing large concurrent numbers of connections. Unfortunately it is very difficult to envisage how implementing reliability at this high level might translate into FPGA hardware issuing its own network transactions.

Where the connectionless element of UDP is helpful is in the sheer number of

clients that can be serviced simultaneously from a single endpoint. As the protocol is completely stateless, holding no information about in flight transactions, source-destination flows etc. the protocol is highly scalable. This is why it is a good transport layer for applications such as media streaming, where servicing a large number of clients with a low latency is more important than delivery of every single packet.

UDP is not typically used for HPC applications, due to the necessity of reliable transfers at scale. It is conceivable to see the use of UDP for certain deep learning applications where acceptable results can still be garnered even with the loss of significant amounts of information. Although it is difficult to see how essential control information could be guaranteed without using the TCP protocol additionally. Recent works have shown that distributed machine learning algorithms can produce adequate results in the presence of a high degree of errors and noise [137]. This noise is typically introduced due to lossy compression/quantization, but in distributed algorithms can be introduced by relaxing communication/synchronization models. While there is very little work on running these algorithms directly over an unreliable transport, recent simulations injecting packet loss have demonstrated that it is feasible [138].

In terms of the infrastructure requirements to achieve this, there has been previous efforts to extend RDMA to an unreliable transport. Typically only supported over TCP, solutions such as RoCE [95] and iWARP [94] allow RDMA in Ethernet networks. In [139] a method is provided to allow for iWARP to be used over UDP. They modify the traditional one-sided sender based RDMA, where the sender must notify the receiver that new data has arrived as the receiver is oblivious to the data transfer. They introduce *Write-Record*, where the data which is sent to the receiver is placed in memory as normal, but then also posts to a completion queue to notify the receiver of a partial message transfer. The independent UDP packets and notifications can be coalesced if the message is small enough, and the receiver is

therefore notified of every 64KB that is transferred. This solution can provide scalability inherently, by removing the need for connection state information and source-destination pair buffering. Unfortunately however they do not afford reliable transfers. While this may be acceptable for some data centre workloads it is very difficult to envisage a scaled HPC application being able to tolerate packet loss in such a manner with reliability provided at the application layer.

2.5.4 Infiniband

Infiniband is a high performance, low latency interconnect for HPC systems which began life in 1999 as the merging of two separate projects; Future I/O and Next Generation I/O. The Infiniband consortium members are comprised of most of the major vendors within the HPC arena, and it is now the most popular interconnect solution for HPC systems after Ethernet. However, whilst making up 27% of the systems share on the TOP500 list it constitutes over a third of the total performance share (Figure 2.10). It is a well established and proven interconnect solution which overcomes many of the performance and latency issues associated with TCP/IP. Infiniband networks offer a lossless link layer, with link-level credit based flow control to provide a lossless fabric.

Infiniband provides two sets of semantics for communicating; channel, and memory. Channel semantics provide the standard, two-sided send/receive type operations used in sockets programming, where the receiver must be aware of the communication taking place. Memory semantics allow for one-sided communications using RDMA. The hosts must first register memory within the Infiniband hardware to provide the location of the DMA buffers which are pinned in memory. These buffers can be read or written by Infiniband, as functions are provided for remote read as well as write. However, these semantics do not aid in the provision of direct shared-memory operations. As Infiniband uses global and local *identifiers* to route through the network, rather than direct memory addresses, there is no way to issue direct load/store commands to Infiniband.

[14] As of November 2018. Image taken from TOP500 sublist generator.

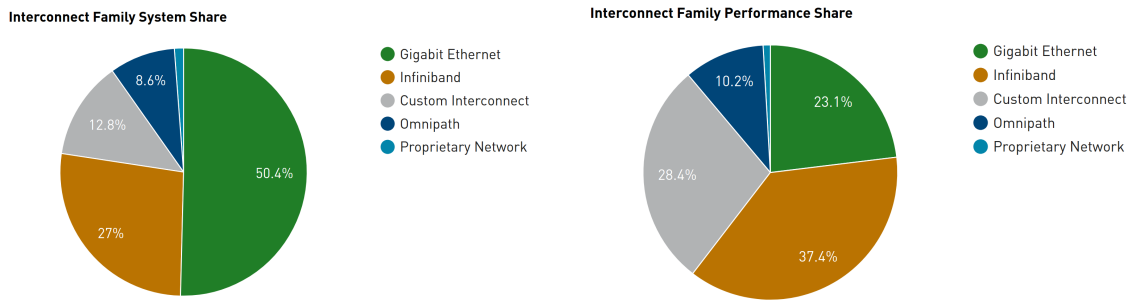


Figure 2.10: Interconnect Families in the TOP500 supercomputers[14].

2.5.4.1 Transport Types

The standard Infiniband specification allows for 5 transport types; Reliable Connection (RC), Unreliable Connection (UC), Reliable Datagram (RD), Unreliable Datagram (UD), and Raw Datagram (RAW). The RC and UD transport form the most typical methods of communication, and are analogous to reliable connection based TCP, and unreliable connectionless UDP within Ethernet networks. Reliable Connection sets up a connection between source and destination pairs prior to transmission, providing dedicated send and receive Queue Pairs (QPs) for these messages. This requires large amounts of memory for many active connections, creating a scalability issue. Attempts have been made to reduce this issue with software services layered on top of the transport layer [140], such as the SRQ (Shared Receive Queue), which uses a single queue at the receiver to be associated with multiple Queue Pairs.

The Reliable Datagram (RD) transport would appear to solve all of the issues associated with scalability, allowing a single QP to send and receive from any other RD QP. However, this has proven very difficult to implement and as far as we are aware no hardware implementation exists; certainly no manufacturer currently supports this transport. Efforts have been made to extend the functionality of the UD transport, providing reliability in software [136], [141], but these have not been adopted as replacements for connection based approaches.

The XRC (eXtended Reliable Connection) [142] is a transport service which extends the SRQ and gives improved scalability to multi-core machines, further reducing the memory footprint of the connections. Whereas the normal RC transport requires P^2N connections at each endpoint, XRC requires only PN, where P

is the number of processors per node and N is the number of nodes. This is because instead of requiring a process to have a connection to every other process, it only requires a connection to every node. However, the problem still remains that the memory footprint is bound by the number of connections, so scaling beyond a certain point becomes difficult. Decoupling the transport from the number of connections is possible via dynamic setup and teardown of the connections. However, this will introduce additional latency over static, dedicated connections which are set up during the initialisation phase of the application and remain for the duration of the run.

2.5.4.2 Complexity

The Infiniband specification is very complex, and whilst Infiniband capable Network Interface Cards (NICs) offload transport mechanisms to hardware, is not very well suited for FPGA implementations due to its complexity. The main problem with regards to FPGA implementation of Infiniband is the complex Verbs specification [143], which acts as an interface to the Host Channel Adapter (the Infiniband equivalent of an Ethernet Network Interface). This provides vendors with a set of functions that must be implemented as a software interface to the interconnect. If we wish for applications to be able to utilize the full Infiniband specification then the accelerator would need to be able to communicate using this software interface. Attempts have been made to implement the Verbs API on a GPU in [144], and they found that the large overheads involved in work request generation are not compensated by the savings in context switching; concluding that CPUs are better suited for this task. They go further than this and suggest that GPU architectures will need to be adapted for future systems; including an on-board processor to handle IB (Infiniband) interactions. The alternative being to seek a different RDMA capable networking hardware.

Another option would be to use a Target Channel Adapter (TCA), which has no required software interface, and allows for a far more limited feature set to be included (implementation defined). However (as the name suggests) the TCA is intended as a target for I/O, often bridging between other protocols and providing specialized hardware. They are not intended to replace the HCA as a more

lightweight communications interface for accelerators, and cannot be used to upgrade the FPGA's status beyond that of a network attached accelerator. Current implementations of Infiniband adapters on FPGAs only implement TCAs and have limited capability. For example the Polybus IP core[15] does not allow for reliable transfers and is limited to 1024 Queue Pairs (over the Unreliable Connection transport). Our interconnect solution avoids the issues with such a complex API as we use a simple network protocol designed specifically to facilitate lightweight implementation within the FPGA.

2.5.4.3 Routing

Infiniband, much like IP, uses deterministic table based routing. As such it suffers from many of the same pitfalls as described in Section 2.5.2.5, being unable to route adaptively and utilize path diversity in certain topologies. Each connection pair can only have a single path through the network. Attempts have been made to enable multipath routing within Infiniband [145], however these solutions are often limited by the existing specification and available hardware. They are not implemented as genuine extensions of the protocol, and we are unaware of their widespread adoption as genuine techniques for multipathing.

In [146] a method is proposed to enable multipath routing within Infiniband by changing the switch architecture. The Infiniband specification states that forwarding tables must contain only a single output port per destination, but by changing the internal architecture of the switch multiple ports can be used without changing the specification. However, this has never been implemented.

An alternative which has been implemented is to assign multiple Local Identifiers (LIDs) to the same physical node, and create multiple entries in the routing table. This is proposed in [147] and implemented in [148]. This is a very simple method to enable multipath routing but has some drawbacks. The number of table entries required will grow very large in a network with many connections and high path diversity, as multiple entries are required for each endpoint. The solution works by simply using multiple LIDs for a single location and then routing to one of

[15] Polybus Xilinx IB Link Layer IP Core- `http://www.polybus.com/ib_link_layer_website/ib_cores_brochure_xil.pdf`, accessed April 2019.

these LIDs which are actually destined for the same node. (Infiniband routes using a Global ID, intended to represent the subnet, and a Local ID intended to represent the endpoint within a subnet.)

2.5.5 Others

2.5.5.1 Cray Aries Interconnect

Cray's Aries NIC/Gemini interconnect technology [102] (later acquired by Intel[16]) was formed of an ASIC which allowed for four connected nodes within a blade via four independent Network Interfaces. As well as this they provide a router which enabled connections in a Dragonfly [149], at the time a completely novel topology.

They created a global addressing scheme, and the software stack enabled the use of a hybrid MPI and SHMEM programming model, providing direct hardware support for PGAS operations. The NIC is interfaced via PCIe, with the blade designed such that future incarnations could include CPU+accelerator configurations. Their system allows for caching at local nodes only, as remote operations are performed as PUT/GET, and are thus not cacheable by the initiator. They use two methods of communication within the NIC, offering *Fast Memory Access* and *Block Transfer*, with the fast path for single word transfers, facilitating lower-latency accesses for PGAS shared-memory operations. They also allow for out-of-order delivery on RDMA data, presumably to take advantage of the path diversity of the Dragonfly topology they have used for system implementations They require only a 128-bit write into the NIC and an x86 SFENCE instruction in order to initiate a fast transfer.

Unlike our solution, their NIC is interfaced using PCIe, and so would require acceleration hardware to implement elements of the PCIe's kernel driver if it were to be used as a master device. Not only this but the use of PCIe in this instance makes it difficult to envisage a simple solution in which the use of global shared-memory could be extended to include the FPGA as an initiator, due to the area complexity and difficult programmability that this number of protocol layers may involve. The

[16]Intel Makes a Deal for Cray's Interconnect Assets- `https://www.hpcwire.com/2012/04/25/intel_makes_a_deal_for_cray_s_interconnect_technology/`, accessed April 2019.

use of distributed FPGAs may also be difficult, although recent work has demonstrated the feasibility of direct GPU-FPGA communication over PCIe [150].

2.5.5.2 Intel Omni-Path

Intel's Omni-Path interconnect [151] is descended from Cray's Aries, and has been designed to enable the largest systems in both HPC and data centre contexts. It uses both link-level and end-to-end retransmission schemes, and uses the standard physical layer of Ethernet or Infiniband. It provides full support for the standard Infiniband Verbs API, and supports the use of the Shared Receive Queue connection type (Section 2.5.4.1). It allows for out-of-order delivery at the interface, with hardware reordering within the NIC. Omni-Path uses a connectionless approach, holding no connection state in the NIC, and like its predecessor it offers a reduced latency path into the NIC for short messages. Unfortunately only limited information is available on this proprietary technology, but given the support for Verbs API it is likely to suffer the same complexity issues for FPGA implementation we raised in Section 2.5.4.2. As such they utilize an *onloading* model, with lots of the work being done by the processor[17].

2.5.5.3 Blue Gene/Q

IBM's Blue Gene/Q is the third instance of their Blue Gene systems [152]. It offers a 5 dimensional torus topology, with 10 bidirectional links emanating from their compute nodes (each chip consists of compute nodes and I/O nodes for the file system). It supports hybrid programming models, with multi-threading enabled within the nodes and message passing for inter-node communication. In the earlier L/P instances a separate network was used for collective operations, whereas the Blue Gene Q integrates collectives operations into the router for the torus network.

The network accesses the memory subsystem via a *Messaging Unit* [96], which provides direct hardware support for RDMA read and write, as well as hardware network FIFOs which can be used for sending and receiving direct point-to-point messages. Unlike our proposed solution there is no special mechanism for handling

[17] Battle of the Infinibands, Part Two- `https://www.nextplatform.com/2018/04/13/the-battle-of-the-infinibands-part-two/`, accessed April 2019.

lower latency, small message transfers, and no method for sharing memory beyond the boundary of a node; with RDMA being the main vehicle for communication. However, the messaging unit features the ability for atomic operations during the course of a transfer, effectively performing in-network processing. This can enhance the performance of locking mechanisms and notification to work queues etc.

2.5.5.4 BXI

The Bull eXascale Interconnect (BXI) [153] has been developed by Atos for future exascale computing, providing direct hardware support for the Portals 4 interface[18]. All primitives (one-sided RDMA, as well as two-sided send/recv) are offloaded into the NIC, decoupling computation and communication fully. A dedicated fast path for PGAS messages is provided enabling lower latency transfers for these. As well as this atomic operations are offloaded into the hardware, further aiding implementation of PGAS models. The NIC contains a virtual-to-physical address translation unit, providing the ability to directly access shared virtual memory regions of remote nodes. Their router supports the implementation of many modern topologies including Dragonfly and Slimfly [155].

BXI provides an end-to-end reliability technique offloaded into the NIC hardware to bypass the need for soft retransmissions, with local flit-level link CRCs to detect transient errors and retransmit at the link-level. Our analysis in Section 5.3.2.3 and 5.3.3 shows that this sort of link-level retransmission is unnecessary in our system except at the most extreme error rates. In the end-to-end scheme they implement a go-back-N protocol to retransmit lost or corrupted packets and hold copies of the messages in retransmission buffers. We are able to solely retransmit lost packets as we do not require byte-ordering on reception. They acknowledge the expense of these buffers within the NIC and aim to mitigate against this by merging these buffers with stored connection state information. As the BXI NIC is interfaced using PCIe, it will suffer the same complexity issues when using distributed FPGAs as we discussed in 2.5.5.1, with any attached FPGAs likely to be viewed as a networked coprocessor.

[18]Portals provides a network programming interface for supporting the simple implementation of upper-layer communication protocols such as MPI, SHMEM or UPC [154].

2.5.5.5 Tofu

Fujitsu's K-computer uses their custom Tofu interconnect solution [156]. Tofu uses a 6D mesh/torus topology, providing similar characteristics and bisection bandwidth to the Blue Gene/Q [157]. They use link-level retransmissions in order to reduce the latency overhead when compared with the go back-N style of TCP/IP. Our solution differs in that our analysis shows that link-level techniques are unnecessary for our targeted network as we do not require go back-N for end-to-end retransmissions and we target a lower diameter network than Tofu (see Section 3.1.2). In Tofu collective operations such as broadcast, barrier, and reductions are all offloaded into hardware. The controller supplies a method for using RDMA, and short messages can be sent via a *piggyback* method; this reduces the latency of transmission by embedding the payload into the transfer descriptor as it is sent to the Network Interface [158].

2.5.5.6 FPGA Based Interconnects

There are a number of architectures in the literature which are either pure FPGA clusters using a custom interconnect, or in which the FPGA is used as the Network Interface implementing a custom network protocol, but not used as an accelerator. Bluehive [159] is one such pure FPGA cluster which uses a custom reliability and link layer; *Bluelink*. The system is designed for latency critical and small message transfers in neural network applications [12], rather than targeting general purpose HPC. It uses a short *replay buffer* at the sender to retransmit unacknowledged/timed out packets. This creates a short window for in-flight transactions and requires a point-to-point network as routing is performed hop-by-hop.

In terms of FPGA based NICs, EXTOLL aims to support both PGAS and MPI type communications in their custom network [160], connecting to host processors over HyperTransport. NetFPGA is an open source platform for implementing network interfaces on FPGAs, interfaced to hosts via PCIe [161]. APENet+ [162] is an FPGA based network interface designed for GPU based HPC communication in a 3D torus. They provide RDMA and enable direct GPU-GPU communications. In all these designs the FPGA is intended solely for use as a NIC, rather than as an

accelerator as well.

2.6 Our Interconnect Requirements

We have shown in Section 2.5 that there are many pitfalls to traditional networking methods, and other works have demonstrated similar requirements for fully customized solutions [12]. There are a great many solutions which provide TCP/IP capabilities to FPGAs, and given its continued presence in commodity HPC clusters it would seem an ideal candidate. Unfortunately as we mentioned in Section 2.5.2 TCP based reliability suffers severe performance degradation when implemented in software, and hardware offloaded techniques are non-scalable due to the complexity of the TCP protocol and the dedicated per-connection send/receive buffering requirements. While scalable, the use of more lightweight UDP transport cannot afford us the strong reliability we require for a scaled HPC system.

Alternative ASIC based solutions with the NIC on a separate die suffer from the fact that PCIe connection is (typically) required. Direct chip-to-chip communications would require a fully customized ASIC and PCB design. While modern FPGAs now typically contain hardened PCIe cores to reduce the complexity of implementing solutions within the FPGA fabric, there are many downsides to the use of PCIe. The first is that it decouples the network from the compute, creating a hard boundary between the two. This prevents shared memory operations from extending beyond the node level. The second is that for the FPGA fabric to communicate through PCIe to distributed resources will require an extension to the typical programming models for FPGA design. A more traditional solution would require CPU intervention for communications, which is something we deem unacceptable. This fact precludes the use of Infiniband technology for FPGA based HPC, because as we discussed in Section 2.5.4.2 it is too complex for Host Channel Adapter implementation within the fabric itself.

Other works [12] have also questioned the suitability of commodity interconnect solutions, arguing that the complexity of some standard solutions is unnecessary and the software reliability/unreliability of others is unacceptable for given

application requirements. They offer a lightweight solution with hardware reliability, and low latency transmission for small packets. This solution is not feasible for our needs however, as the focus on small packet sizes, while useful for their neural computation engine is not suited for more general use [159].

2.6.1 A Custom Interconnect Design

Having examined an array of possible interconnection technologies and existing FPGA-based HPC systems, we have determined that a custom solution is needed to fulfill our requirements for the interconnect. The primary goal of the work presented in this thesis is to support communication between distributed FPGA resources at rack-scale in a general heterogeneous HPC system with the FPGA as the main compute element. We have shown that no existing technology can support this without imposing severe limitations on the network; either in terms of the performance, scalability, complexity or reliability. As such we propose our own interconnect design which is implemented within the fabric of the FPGA.

This work has been done amid the context of a set of pan-European projects [92], [163]–[165] which aim to demonstrate a prototype heterogeneous computing system based upon FPGA compute resources. The central tenet of these projects is that the only viable way forward for future exascale class HPC systems is for the use of the FPGA as the main compute element, with dense packaging and close coupling of network/storage/memory to achieve the ambitious power envelopes to which future systems will be constrained ($\approx 10^{18}$FLOPS at 20MW). A detailed background into the motivation and activities of these projects is provided in Appendix A.

The system our interconnect targets is a generalized HPC system, but obviously there are some caveats to the potential use cases of FPGAs for generalized HPC workloads [9]. As we have discussed at length in Section 2.2, there are a number of workloads which are of course more suitable for execution in GPUs; those involving high arithmetic computation and regular memory accesses for example. However, our work builds upon the principle that there are a number of highly relevant (indeed some emerging) workloads which will benefit from tightly coupling the network, accelerator and host, using a global address space allowing for distributed and shared virtual memory between FPGA and host CPU [9]. Such workloads will

be heavily control-based with pointer-rich data structures, and will need to take advantage of the high volume of on-chip memory in the FPGA to reduce the level of communications.

The majority of applications which will be able to take advantage of such a system architecture are those which are heavily data intensive, such as those from the domain of data-analytics and deep learning. However in order to fully exploit the use of this system for these applications where traditional multi-cores excel it must be noted that advancements in many areas beyond the scope of this thesis will also be required. The software ecosystems used to program these systems will need advancing, as design reuse is a key concern in classes of problems such as branch-and-bound, map-reduce and neural networks. The development of standard HPC libraries for FPGA acceleration is required, along with simpler programmability and scalability across distributed resources. Our custom interconnect addresses some of these issues by making inter-FPGA communication far simpler, providing hardware primitives for shared-memory operations across the network.

Given that we have determined that a custom interconnect solution implemented within the fabric of the FPGA is the only way that we can satisfy all of our requirements. The Network Interface design we describe in this thesis advances beyond the current state of the art by supporting the following features. Note that existing solutions may address some of these issues individually, but we are the first to address every one in combination:

1. We offload the reliability layer into hardware wherever this is possible. (Fallback on software mechanisms is still required to provide application level reliability and tolerance for terminal network errors). This eliminates the requirement for CPU intervention for network communications, vastly improving the performance over a similar software based transport layer. We offer a relatively lightweight networking infrastructure which permits FPGA networking capability to easily accommodate for accelerator resources in the same FPGA fabric.

2. We provide hardware primitives to support an MPI+PGAS programming model,

which we identify as being the most appropriate solution for future heterogeneous FPGA based HPC systems, in terms of the user programmability and performance (by breaking the reliance on the host/accelerator programming model). We also offer a number of performance enhancements to better facilitate true one-sided MPI communications.

3. We allow for scalability of the transport layer to a comparatively high number of concurrent communicators. Our solution requires no static information to be held in the Network Interface, as would be typical in a traditional offloaded, connection based transport layer. Instead we only require transient information to be held regarding outstanding network transactions.

4. Inter-FPGA communication is extended beyond the limit of a single blade/chassis. Many existing solutions require software initiated communication to extend beyond a given scale. Our Network Interface is capable of handling out-of-order packet delivery. In doing this we enable the use of a lightweight, custom FPGA based switch design[19] which means that we requires no additional networking capability for communication outside of a restricted point-to-point topology.

2.6.1.1 Hardware Offload

As we discussed in Section 2.4, traditionally HPC or data centre architectures have used accelerators as PCIe attached coprocessors, either only capable of communication between themselves and their host CPU [110], [111], or to other accelerator resources via limited, point-to-point networks [11], [114]. Others have decoupled the CPU and FPGA completely, allowing the FPGA to stand alone within the system issuing and processing its own network transactions. The recent work at Microsoft on their Catapult 2 system [6] provides the most advanced example of this. This sort of architecture lends itself incredibly well to in-network processing, and distributing compute over multiple FPGAs in the instance where a problem is too large for a single device.

We view this sort of architecture as the prime candidate for reconfigurable HPC,

[19] The implementation of this switch is beyond the scope of this thesis, but is detailed in [32].

as the FPGA should be regarded as the main compute element within the system, and the network should be as tightly coupled to compute and storage as possible to minimize data movement. Transforming the communication element into a programmable compute element which can utilize in-network processing and dataflow style computation is necessary to reduce the power consumption and push compute closer to the memory and storage resources. Such an architecture opens up the possibility of Near Data Processing (NDP) [122], [123].

In order to provide the FPGA with the capabilities it needs for tighter coupling with the network a full hardware-offload for reliable transmission is required, bypassing the CPU in order to maintain performance with direct FPGA-FPGA communication. Offloading the transport layer into hardware has long been known to solve the performance issues associated with software based transports, but in this instance it is also necessary to disaggregate the FPGA from the CPU. Going through the CPU has been shown to be detrimental to the achievable throughput and latency within the system. As well as this the use of the CPU is antithetical to the idea that the FPGA should be viewed as a stand-alone resource, decoupling the CPU from the FPGA completely and allowing scaling of the FPGA resources independently of the CPU.

2.6.1.2 A Novel Transport Layer

The primary issue with traditional connection based transport layers is that large numbers of short lived communication channels (typical of many HPC workloads) are not well supported by this model. Establishment of the connection state information at sender and receiver are required prior to data transfer, and teardown is required to free this state. This means that additional latency is seen due to the additional handshaking required to continually establish and teardown these connections.

A related problem to this is the fact that if a great number of these connections are simultaneously created but are left open in order to reduce the latency of future communications, then the burden on the memory utilization of the host can become very expensive. Typical one-to-all (scatter) type collective communications seen in many HPC workloads can create enormous numbers of these connections

simultaneously, making connection based transport layers unsuitable at scale.

Connectionless transport mechanisms forgo this stored state information and explicit handshaking which ensures strict parameters for guaranteed delivery and ordering. Instead they force the application or software library to deal with any erroneous or missing data at the receiver. This approach allows for much lower overhead on communications but is obviously incompatible for applications which require a lossless fabric. Typically it is streaming applications which can handle missing information or reduced precision via interpolation or error detection/correction are able to take advantage of connectionless transport protocols.

Our interconnect is able to provide the high performance, hardware based reliability of a connection based transport layer, without any of the explicit communication overheads and static state information overheads associated with them. We can offer this because within our target system the destination of each network transaction is simply a memory address located within a virtual, global address space. Since every network transaction is simply an extension of the system-bus protocol, we need no information regarding source-destination pairs, target buffers etc.

The only information required which needs to be stored in the Network Interface is transient information regarding in-flight transactions which have been issued to the network[20], which is used to rebuild failed transfers. In doing this the scalability of the transport layer is no longer determined by the number of possible concurrent communicators, but on the number of concurrent in-flight network packets from a given source node. Many of the traditional issues regarding ordering, packet sequencing and such are also eliminated simply because we write to target memory locations using RDMA, rather than writing to target connection buffers. In this manner the traditional "connection" between source and destination is made implicit within the transfer, as RDMA buffers are set up at initialisation time and determination of the state of these buffers is done at the application layer, completely independent of the transport and reliability mechanisms.

[20]Our performance enhancements also store information regarding the progress of in-flight RDMA transfers being received.

This solution offers reliability without the scalability issues of traditional hardware-offloaded reliability layers, and this is perhaps the most important overarching contribution of this work. This is because the requirements for efficient distributed FPGA based communication demand such a solution. One which offers the ability to have the rich feature-set of a traditional HPC interconnect, with the low hardware overheads of a connectionless transport layer and complete independence from the CPU. In separating the reliability aspect of the transport layer from the connection aspect which is made implicit in the RDMA or shared-memory transfer, we vastly simplify the operation of the Network Interface. In this manner, in the points at which we refer to our transport layer as being *connectionless* within this thesis, we mean to say that there is no requirement for the hardware to store and maintain static connection information.

2.7 Concluding Remarks

We have seen in this Chapter that burgeoning application domains and ever tighter power constraints are turning systems architects toward FPGAs. While GPUs are capable of providing very high floating-point performance, the FPGA has been shown to outperform GPUs in terms of their performance-per-watt, as well as showing promising performance for more data-intensive workloads. We have shown that if the FPGA is to be effectively utilized then it must be regarded as the main compute unit within the system, with the CPU being used for simple synchronization and management tasks.

We argue that a custom interconnect solution providing hardware primitives for NUMA accesses and RDMA is the most efficient and effective option, given the problems with existing FPGA based interconnects and the communication patterns expected for general purpose HPC on distributed FPGAs. The programming model for modern FPGAs must be easily able to drive the Network Interface in order to promote programmability and portability between platforms.

Therefore our custom interconnect solution must offload the capability for reliable network transfer onto the hardware of the FPGA (or at least out of the software of the CPU). However, in order to enable this on the FPGA and maintain scalability

of the solution we must ensure that no connection state is held or source-destination pair transmission buffers are required in order to maintain reliability.

We have demonstrated the benefits which arise from tight CPU-FPGA memory coupling, as well as from the full decoupling of the CPU and FPGA with regards to network communication beyond the node. In this respect we see that there are many prior works which enable either one of these requirements, but very little work in synthesising these two solutions. Our Network Interface must therefore enable the FPGA fabric to communicate simply with remote memory and accelerators, using the same communication models as it communicates with the local CPU.

This solution fits the model described in Section 2.4.6, and is something that no available solution currently offers. Other shared memory systems containing FPGA accelerators such as [105] are either fixed in their scalability and/or topology among the FPGAs [116], inhibiting the use of distributed FPGA resources in a dataflow architecture. Typically they also do not provide for disaggregation of the FPGA from the CPU resources [117]. Other network-attached FPGA systems such as [119] which allow this decoupling typically have limited scalability or they do not allow for low latency access into shared memory since they rely on standard protocols such as TCP/IP or UDP. Those solutions that offer custom and dedicated hardware offloading for reliability are limited in their scope, typically providing only lightweight point-to-point networking capability [11].

Chapter 3

Network Interface for HPC Communications

In Sections 2.4 and 2.5 we discussed the limitations of current interconnect technologies with regards to wider uptake and effective exploitation of FPGA based HPC systems. We argue that current solutions and standardized network protocols are inherently unworkable. This is mainly due to architectural designs causing performance or latency issues (such as the use of a software based transport layer) or scalability and reliability problems due to the nature of the protocols themselves (using connection states for reliable transfers).

Given this viewpoint, the design of a new custom Network Interface (NI) is proposed and described within this chapter. This NI uses a custom lightweight network protocol (which solves some of the issues discussed previously), and aims to address the requirements to support two distinct hardware primitives for HPC communications; remote shared memory operations and RDMA transfers. These two primitives are supported to enable two very disparate communication types to use the same physical interconnect (small low latency synchronization messages, and larger high bandwidth data transfers). One of the driving factors behind the design is the need for a high performance FPGA based interconnect which enables tight coupling between the FPGA fabric and the network, but maintains low enough logic overheads to enable effective accelerators to be placed within the fabric as well. These are key features for enabling the uptake of FPGA devices for HPC.

This Chapter provides an overview of the proposed system architecture and topology, and the most basic properties of the Network Interface and switch. We discuss the interfacing required between the system bus, NI and network, and show our segregated traffic paths for shared memory and RDMA operations with an analysis at the end of the Chapter justifying our design decisions.

3.1 System Architecture

It is obvious that the Interconnection Network is one of the major limiting factors in system scalability, performance and power consumption. As overheads on communication and synchronization become heavier as the system scales, maintaining a high bandwidth, low latency solution is key. Currently the largest HPC systems are supporting in the order of a hundred thousand nodes, and this is expected to rise to the order of a million endpoints [166]. Scaling many of today's fixed topologies to this level is simply not feasible, especially given that network costs and power consumption are among the main concerns in practical deployments of large scale systems [167], [168]. Supporting a tiered interconnect with hybrid topologies is viewed as an essential way to keep power consumption and costs under control without sacrificing performance [169].

3.1.1 Hierarchical Interconnect

The targeted system architecture is based around the one which is used in the ExaNeSt project [163], which uses a tiered interconnect solution and offers topology hybridization, with a fixed lower level topology at the node and within the blade and flexibility in the upper tiers. Figure 3.1 shows this system architecture. Each node (QFDB, Quad FPGA Daughter Board) is comprised of four Xilinx Zynq Ultrascale+ FPGAs (detailed in Section 3.2.1), connected directly in an all-to-all fashion with high-speed serial links. Within the mezzanine the nodes are connected in a fixed, direct torus topology which allows for lower latency communication between neighbouring nodes. The upper tiers of the network will be formed from indirect, high radix switches which are able to interconnect large numbers of these lower tier networks while keeping the overall diameter of the system low. This has

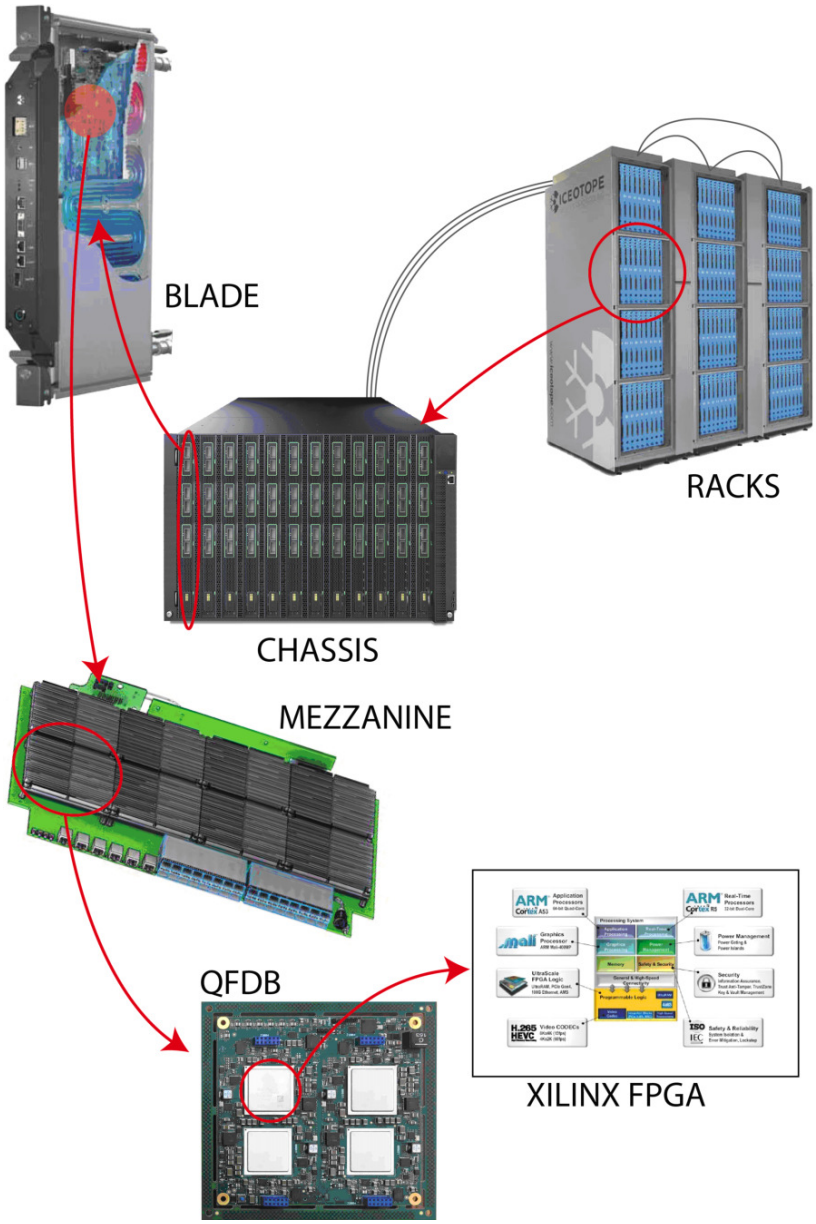


Figure 3.1: Tiered system architecture of ExaNeSt.

been shown to be an effective means of scaling the system when compared with extension of the torus beyond the blade. We demonstrate this enhanced scalability and reduced cost in [170], although the work is beyond the scope of this thesis.

3.1.2 Network Topologies

As mentioned above, the chosen topology is very important for the performance of the system, as it directly affects the latency of communication and the available

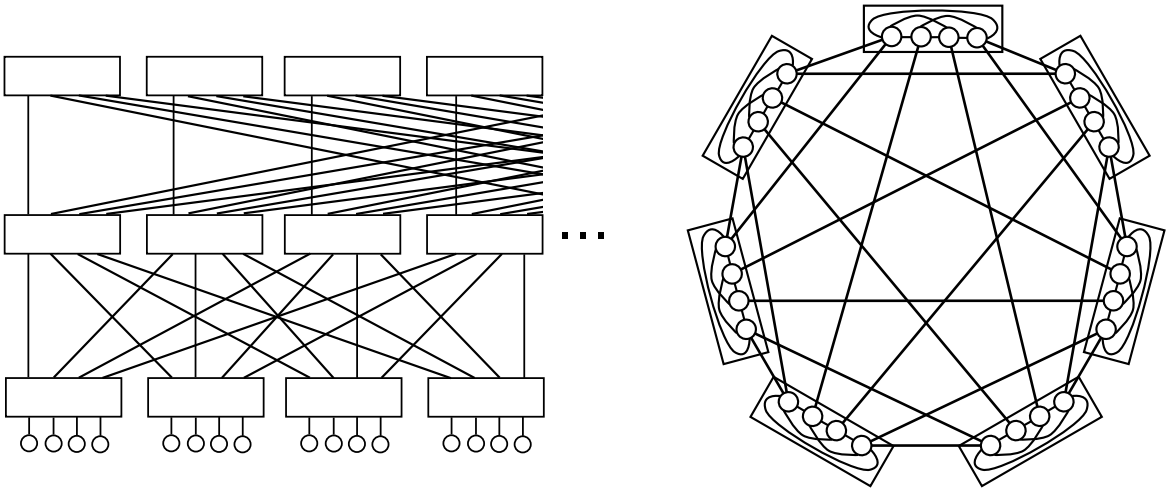


Figure 3.2: Possible topologies for the system. One quarter of a 4-Ary 3-Tree Fat-Tree (left) and Dragonfly (right).

bandwidth. In the upper tiers of the network our system targets modern, low diameter topologies such as the Fat-Tree and Dragonfly. Examples of these topologies are shown in Figure 3.2. On the left is a portion of a fully connected Fat-Tree network with 3 tiers and 4up/down links per switch (4-ary 3-tree). Table 3.1 demonstrates the average number of hops required to reach any destination node from a given source node in varying scales of network. As we can see the high radix, indirect solutions– Dragonfly and Fat-Tree– have superior scaling properties.

Table 3.1: Average distance to reach any node in different network topologies at scale, taken from [171].

Nodes	**3D torus**	**BGQ (5D)**	**Tofu (6D)**	**Fat-tree (R32)**	**Fat-tree (R64)**	**Dragon fly**
1,024	7.50	6.00	6.53	4.46 3L	2.94 2L	3.50
2,048	9.46	7.00	7.48	4.83 3L	2.97 2L	3.62
4,096	12.00	8.00	9.21	4.87 3L	4.48 3L	3.69
8,192	15.00	9.00	11.21	4.93 3L	4.74 3L	3.72
16,384	19.00	11.00	10.43	6.26 4L	4.87 3L	3.73
32,768	24.00	13.00	16.21	6.73 4L	4.94 3L	3.74
65,536	30.23	15.00	20.54	6.86 4L	4.97 3L	3.75
131,072	37.99	17.00	24.37	6.97 4L	6.48 4L	3.75
262,144	48.00	21.00	30.24	8.47 5L	6.74 4L	3.75

3.1.3 Network Switch and Addressing Scheme

Our system employs an FPGA based switch design, presented in [32]. This has been developed to enable flexibility in the topology, scalability when implemented within the fabric of the FPGA, and support our custom network protocol. Our switch utilizes a geographic addressing scheme to route packets to their destination arithmetically, using the upper 22 bits of the address (allowing for over 4 million possible endpoints). Two routing algorithms have been developed which enable Dragonfly or Fat-Tree topologies to be easily constructed.

The switch is able to eliminate the need for costly routing tables due to the arithmetic routing scheme, making it highly suitable for FPGA based implementations. It also allows for multiple endpoints to be reached directly between interconnected FPGAs, in a torus topology for the lower tiers for example.

One of the main implications of the design of the switch (specifically the lack of routing tables) on the Network Interface is that it allows for fully adaptive multipath routing, taking full advantage of the path diversity of the network [172]. What is meant by this is that any packet can take any path through the network to reach its destination. In standard interconnect technologies such as Ethernet or Infiniband this is not a simple task, rendering topologies with high path diversity ineffective [171].

Typically a table based routing mechanism will perform a lookup on a destination and output the packet to a particular port depending on the lookup. In our system if the arithmetic routing scheme sees that several ports can be utilised to get to the same destination, one will be chosen at random[1]. The ability to utilize the path diversity of the network is desirable for balancing load over the links. While multiple paths can be taken in Ethernet or Infiniband solutions these are typically not able to balance load optimally. This is because HPC and data centre traffic tends to be non-uniform and because any single flow/connection between source and destination is routed deterministically, using a single path for all packets. Only different flows can be sent along different paths. This load balancing issue caused by non-uniformity of traffic is particularly exacerbated by recent developments in

[1]Minimal routing is known not to balance traffic very well in Dragonfly topologies for example, so randomized Valiant routing is typically used [173].

unifying the storage and network traffic on the same physical interconnect [174].

However, the ability to utilize fully adaptive routing within the network has a significant impact on the design of the Network Interface, due to the ability for Out-of-Order (OOO) packet delivery. Since forwarding happens in the switches on a packet-by-packet basic arithmetically, ordering cannot be guaranteed. Techniques such as employing reorder buffers [175] can be used to alleviate this problem, but are expensive in hardware. In Chapter 4 it is shown that our system needs no reordering for OOO packet delivery in all but the most exotic circumstances, and that the novel, hardware-offloaded transport layer we present is sufficient for maintaining memory consistency despite the properties of the network.

3.1.4 Unified Interconnect

The interconnect within HPC systems has been shown to consume as much as 35% of the total system power even when idle [125]. It is also the case that traditional HPC systems employ a number of separate networks; a network for communications and data, a separate network for storage, and maybe one for management traffic also. This was typically tolerated due to the fact that storage has traditionally had very high latency access. This meant that storage can be kept in separate physical blades/racks from the compute nodes, with the increased access latency being much less of an issue. However, due to the fact that modern persistent storage devices[2] can now provide accesses in the order of 10μs, there is increased pressure to place the compute closer to storage, giving us hyper-converged architectures.

Given the fact that the interconnect consumes so much power and we can utilise hyper-converged solutions, the system will therefore use a single unified interconnect to carry all network traffic; storage, data, control and inter-processor communication. This will create a more energy efficient interconnect, purely due to the fact that there will be less physical cabling and transceivers to consume static power. Also given that we provide an FPGA based solution this opens up the possibility of placing processing elements within the path to the network and storage elements, further reducing data movement costs. Recent work has shown the feasibility of

[2]Intel Optane SSD DC P4800X Review- `https://www.storagereview.com/intel_optane_ssd_dc_p4800x_review`, accessed April 2019.

this solution [176], showing a solution which allows access to storage from the FPGA fabric without the involvement of the processor. What this unified, single interconnect means for our work on the Network Interface is that MPI traffic (RDMA) and PGAS (shared memory) traffic will both be issued over the same fabric, using the same packet formats and protocols.

3.2 Hardware Platform

As mentioned above in Section 3.1.1, each of the lowest tier nodes is comprised of four interconnected FPGA based Multi-Processor System-on-Chips (MPSoCs), with these nodes providing a number of uplinks into the higher tiers of the interconnect. In this Section we will provide an overview of the Zynq Ultrascale+, and discuss the merits of using this modern MPSoC device.

3.2.1 Xilinx Zynq Ultrascale+

While the design of the NI facilitates the use of distributed FPGAs completely decoupled from any CPU, the work presented across the entirety of this thesis is implemented within Xilinx ZCU102 development boards[3]. The ZCU102 board contains a Xilinx Zynq Ultrascale+ FPGA as well as a number of IO interfacing technologies, although for our work we only make use of the four SFP+ cages available for interfacing between the boards using a number of the 16.3Gb/s transceivers on the FPGA device.

The Xilinx Zynq Ultrascale+[177] (shown in Figure 3.3) is a cutting edge MPSoC (Multi-Processor System-on-Chip), containing a hardened processing system (PS), based around a quad-core ARM-v8 A53 with shared L2 cache, dual-core ARM-v7 Cortex-R5 processor, as well as a wealth of reconfigurable logic (PL), all on the same die (full system details in Table 3.2). The FPGA fabric interfaces with the PS via various AXI master and slave ports (the AXI interface is discussed in Section 3.3.1 and detailed in Appendix C). A number of these are internally connected to the Cache-Coherent Interconnect, enabling the user to create caches within the programmable

[3]Xilinx Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit- `https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html`, accessed April 2019.

Table 3.2: Specification of the Zynq Ultrascale+ device.

Part Number	ZU9EG
Processor cache	4x ARMv8 A53 (<=1.5GHz) L1- 128KB Private Instruction & Data L2- 1MB Shared
Real-Time Core cache	2x ARMv7 Cortex-R5 (<=600MHz) L1- 32KB Private Instruction/Data 128KB Private Tightly Coupled Memory
Memory	4GB DDR4
FPGA CLB LUTs CLB Flip-Flops Block RAM DSP Slices	ZU9EG 274,080 548,160 32.1Mb (912x36Kb) 2520
Tranceivers	24x 16.3Gb/s

logic of the device and enable coherent access to distributed shared memory.

There are three main reasons we target this device. The first one is that we wish to allow for the hybrid decoupled network/tightly-coupled system memory architectural model described in Section 2.4, so we target a modern SoC device which can easily provide system-bus access to the FPGA resources. The choice between Altera and Xilinx SoCs which both provide this functionality lies with the inclusion of the hardened ARM processor cores. There has been much excitement recently surrounding the use of smaller, lower power cores for more energy efficient High Performance Computing [178], and as the supporting tools mature for 64-bit ARM processors, their processors are now being adopted at even the largest scale[4].

The third reason we target such a device is due to the fact that these modern FPGA based SoCs such as the Xilinx Zynq Ultrascale+[5], and the Intel Stratix 10[6] have made a crucial step forward over previous technology toward adoption in HPC and data centre contexts; that is the inclusion of an *IO Memory Management Unit* (IOMMU). Traditionally used for system protection against malicious or faulty peripheral devices [179], providing a memory management unit to peripheral devices can be used to extend virtualization to the fabric of the FPGA, enabling a

[4]Fujitsu Begins Production of Post-K- `https://www.fujitsu.com/global/about/resources/news/press-releases/2019/0415-01.html`, accessed April 2019.

[5]`https://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html`, accessed April 2019.

[6]`https://www.intel.com/content/www/us/en/products/programmable/soc/stratix-10.html`, accessed April 2019.

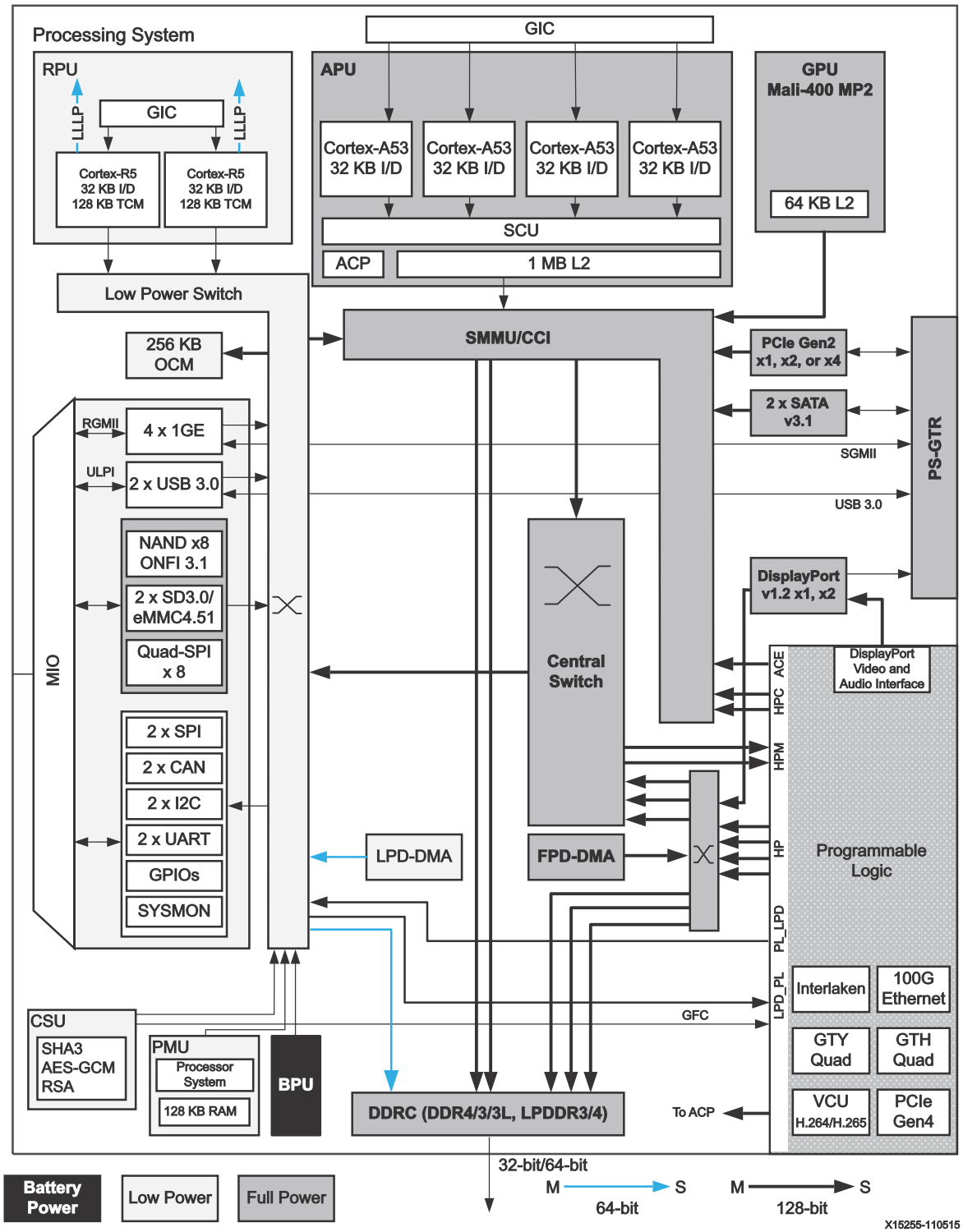


Figure 3.3: Block diagram of the Zynq Ultrascale+, taken from [177] (pp. 18).

unified, global address space [180]. Until recently FPGA vendors gave this feature little attention. However, the paradigm for accelerator based computing has begun to shift away from the traditionally decoupled PCIe attached coprocessor toward a fully integrated solution. One in which the FPGA fabric and CPU are tightly coupled to the same shared memory space, thus enabling lower latency and higher memory bandwidth to the accelerator, as discussed in Section 2.4.

In a complete system the function of the IOMMU is imperative to the functioning of a large-scale globally shared address space. However, the work to enable this functionality is outside of the scope of this thesis, which focuses on the interconnect and the ability of the FPGA to operate independently in an efficient and reliable manner. As such, we provision for the use of a distributed, shared-memory system by use of a small region of physical memory, mapped into a user-space application's virtual memory. Appendix B details how this workaround is set up using the `mmap()` system call, and also discusses the addressing limitations which are faced in using the Zynq Ultrascale+ device.

3.2.2 ARM Cortex-A53

The ARM A53 processor is a dual issue, in-order processor [181]. The Zynq Ultrascale+ contains four of these cores, which share interfacing and resources within the FPGA fabric. While the development of highly complex cores has been instrumental in attaining high single threaded performance, they are notoriously power hungry [182]. In order to achieve greater energy efficiency there have been suggestions about using lower power cores with less aggressive techniques for ILP (Instruction Level Parallelism), sacrificing performance for efficiency [125], [183], [184]. In the context of a HPC system in which the FPGA is viewed as the main compute element, this sacrifice seems appropriate.

Since each core can only issue a maximum of two concurrent instructions, and the pipeline must remain in-order, this causes an enormous latency penalty for remote memory accesses, as the pipeline must stall until the response is returned. We implement a technique for alleviating this issue on remote write operations, providing an *early acknowledgement* before the actual completion of the operation. This mechanism is discussed in detail in Section 6.1.

This issue may raise concerns about the use of such a simple processor (which is simply a product of the SoC we currently use), and while it certainly makes the issue of remote operations more problematic the issue could still occur on other processors. Modern superscalars are designed to mask a limited amount of local memory latency, but will readily stall if latency is more than a few tens of cycles. As the latency for remote memory accesses is far in excess of the latency for instruction issuing, the maximum depth for out-of-order transfers could still easily be reached.

In order to enable this mechanism of offering early acknowledgements for shared memory operations some modifications are required to the NI. Certain operations must be blocked in order to maintain consistency. This is implemented within the transport layer and is discussed in Section 5.2.2. A discussion of the implications of the relaxed memory model on the network is given in Chapter 4.

3.2.3 Multi-Gigabit Transceivers

The Zynq Ultrascale+ MPSoC has several different transceiver types (GTH-/GTY/GTR), with different properties and performance characteristics. The GTH transceivers used for the high speed serial links in our system are set up to run with a line rate of 10.3125Gb/s, with a full duplex link- parallel lanes for TX (transmit) and RX (receive). The transceivers are grouped into banks of four; called a *Quad*, each containing TX and RX lines, along with shared reference clock IO pins. One such Quad is connected to a 2x2 SFP cage which provides access to switches and other endpoints in our system.

Note- *For testing in many parts of this thesis we use a direct connection between two nodes or have the transceivers in loop-back, in order to more accurately take timing measurements from a user-space application. This is because of the current lack of a distributed run-time for the system. However, experiments have been conducted in which the NI and switches are used in conjunction, in order to demonstrate that these two IPs are completely functionally compatible (see Section 3.4.2, 3.4.3 and 6.1).*

The reference clock is typically set up in *Input Mode*, in which the reference clock input is used to clock the transceiver. There is also available an *Output Mode*, in which the transceiver is clocked from another transceiver using a recovered clock (RXRECCLK-OUT) [185]. The resulting output clock can then be routed to use in

other locations.

A MAC (Media Access Control) layer is required prior to serialization in order to add preamble, end of packet sequences and idle cycles, required for ensuring data integrity and synchronization of the link. Following this the information is serialized and transferred across the link. For this work we use the Xilinx Aurora 64/66b MAC and PHY IP [17]. Aurora is a link layer communications protocol developed by Xilinx. It can also be used with 8b/10b line encoding, but in order to achieve lower packet overheads we use 64b/66b encoding, reducing the overhead from 24% to ≈3%.

3.3 Overview of Network Interface Design

The most basic function of the Network Interface is to act as a bridging mechanism between the on-chip memory-mapped AXI (the processor's system bus protocol), and the custom protocol we use for the wider network. In this section we discuss these protocols and the design of this basic functionality, and show how the traffic into the NI is segregated depending on the type of operation being performed (an RDMA operation or a shared-memory type operation).

Figure 3.4 shows the block level design for the complete networking IP stack on the Zynq Ultrascale+. All of the hardware used to support the 10G networking capability is implemented within the reconfigurable logic (PL) of the device. A single AXI interface port (M_HPM0_FPD) is used to access the NI from the processing system (PS); providing the functionality for issuing RDMA transfers, as well as directly performing remote shared-memory operations.

3.3.1 AXI Interfacing

The static regions of memory mapped into user-space which give access to the NI and other hardware within the FPGA fabric are interfaced with AXI ports. ARM's AMBA (Advanced Microcontroller Bus Architecture) AXI (Advanced eXtensible Interface) protocol [186] and its coherent extensions (ACE) are interface standards

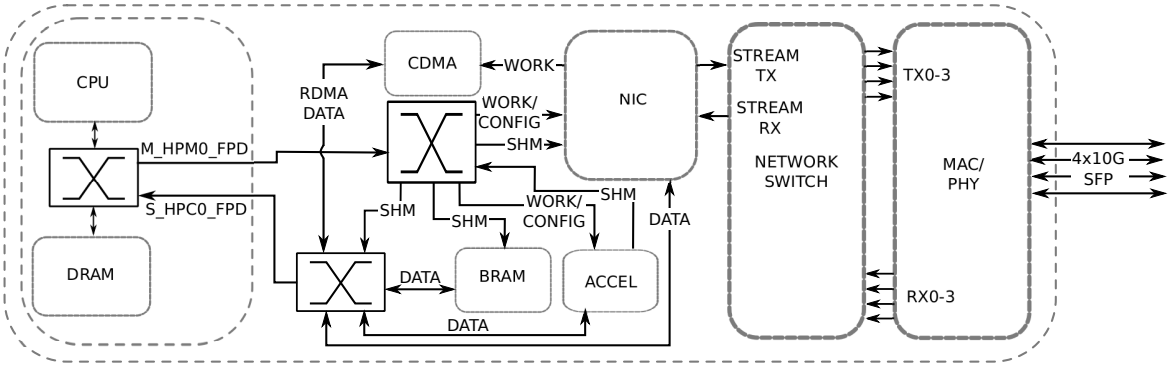


Figure 3.4: Block level system design for the networking IP stack.

which are suitable for use in high performance on-chip interconnects. As SoC designs increase in complexity, standard interfaces are adopted to increase productivity and design reuse. As such AXI has been adopted by Xilinx and forms the backbone of communication between the Processing System and Programmable Logic fabric in their "Zynq" product line, as well as in much of their IP catalogue. Given that certain characteristics of the AXI protocol affect the design of the Network Interface, a brief overview of the protocol is given in Appendix C.

Unfortunately the AXI protocol, while being highly suitable for on-chip communications is inappropriate for communication over a much wider distance. This is primarily due to the lack of support for retransmission or bit error checking in the standard. As well as this AXI has a very wide and awkwardly sized bus width, not simply amenable to transmission over serial links using typical physical layer encodings such as 8/10b or 64/66b [187]. It is for this reason that one of the main functions of the Network Interface is to serve as a bridge between AXI and the custom network protocol which has been defined for use across the inter-chip interconnects within the system.

3.3.2 Network Protocol and Bridge

The custom protocol for the off-chip network consists of a 128-bit header, followed by the payload (if any), and then a 128-bit footer. Each flit of the packet is 64-bits, as this is the maximum that can be clocked into the 10G ports per cycle. Widening the datapath to 128-bits, while allowing for a lower clock rate inside the NI, would increase the area footprint. The requirement for a smaller overhead in

Table 3.3: Packet types and their function.

Packet Type	**Operation**
AR	Shared memory operation, read request.
R	Shared memory operation, read response.
B	Shared memory operation, write response.
AW	Shared memory operation, write request
RDMA_WRITE	RDMA write operation, request/data packet.
RDMA_WRITE_REGISTER	RDMA write operation, first packet of a transfer, to be registered for quick notifications (see Section 6.2).
RDMA_ACK	RDMA write operation, response packet.
COMPLETION_NOTIFICATION	Completion notification packet used between local and remote NI to indicate a completed RDMA transfer for unregistered transfers.

the FPGA is regarded as more important.

The maximum payload size for a single packet has been set to 512B. This means that the largest payload constitutes 64-flits of data ($64 \times 64-$bits$= 512$ Bytes). This incurs additional packet overhead in the form of packet header and footer ($(4/68) \times 100 \approx 6\%$ overhead) over larger packet sizes. For example, given a maximum payload size of 4KB (a full page on ARM-v8), the packet overhead would be a minuscule 0.78% ($(4/516) \times 100 = 0.78\%$). However, having such large packets traverse the network not only increases the buffering required within the switching elements and the NI, but could cause higher jitter for small, latency critical packets. Even using priority based flow-control this may occur if a large packet has been allocated the switching resources before a small packet arrives. For this reason we use a smaller maximum packet length. This will also enable better load balancing within the network, as switching resources will not be held as long. This is particularly true given that the system allows for packet level multipath routing.

In the header is contained information regarding the target node and address, the packet type (see Table 3.3), payload size and error correcting code, as well as other information. The footer contains the source node ID, the ID for the transaction, a packet level CRC code, and other fields relating to the reconstruction of the responding packet. Note that the full source address is not required in the footer, and is not provided by the system bus protocol which initiates the transaction. Once

a response is routed back to the initiating node a transaction ID is then used to route back through the on-chip interconnect to the actual source of the transaction.

3.3.2.1 Bridging

To translate between the on-chip AXI protocol and the custom, off-chip network protocol used over the 10G serial links of the nodes and switches, four distinct controllers are implemented within the NI:

- Sender
  - Memory-mapped AXI request to network packet.
  - Memory-mapped AXI response to network packet.
- Receiver
  - Network packet to memory-mapped AXI request.
  - Network packet to memory-mapped AXI response.

These are formed of a set of independent state machines which parse incoming data, direct it to the correct location based upon the type of operation encountered, and initiate/clean up any tasks associated with the transaction. These tasks include jobs such as keeping track of incoming requests in order to build the corresponding response packet, providing notifications for completed transactions, or checking the validity of the packet. The action of the controllers is described in detail in Appendix D.

3.3.3 Inbound Messages and Response Packets

Given that the AXI response channels are routed back to the source using ID signals rather than a source address, information must be stored within the NI to enable the response packet to be built. When an inbound request packet arrives from the MAC a high-speed CAM (Content Addressable Memory) is used to issue a new ID for the transaction, as well as storing the original ID and the coordinates of the geographic address for the source node. In this way the response can be routed back to the source node, and then from the source NI it can be reformed

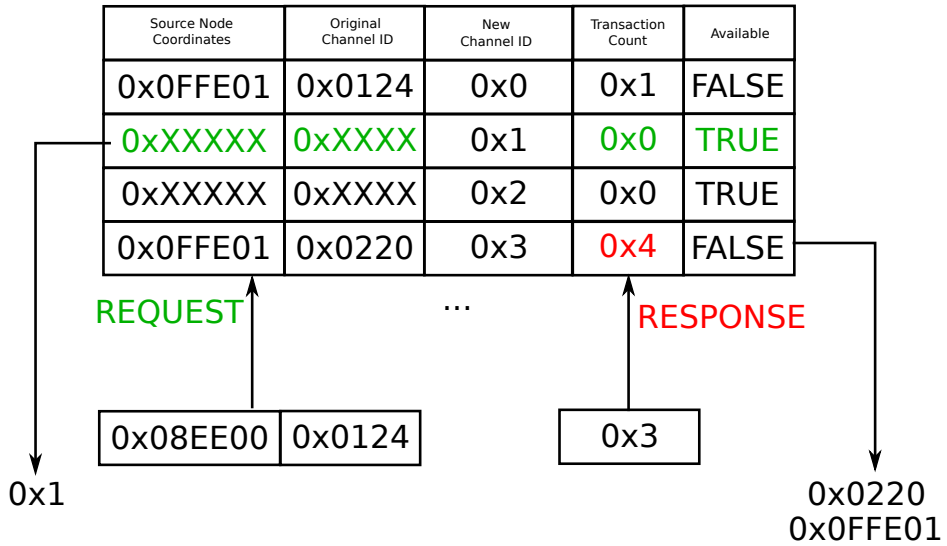


Figure 3.5: CAM table entries for building response packets.

as an appropriate AXI response. A CAM structure is used so as not to incur any latency penalty on forming the request packet; issuing the new ID within a single cycle.

Figure 3.5 shows the structure of the CAM. Upon the arrival of a remote request packet, the incoming source node and ID are checked against the contents of the CAM. If a matching entry already exists in the table then the same ID is issued to the new transaction, and the count is incremented in the CAM on the number of outstanding transactions. This is required, as it is possible to see multiple outstanding transactions with the same ID from the same source in quick succession. For example if the DMA issues many packets, or some data structure is sent using several shared memory operations.

If there is no corresponding entry already in the table, then a new entry is added to the CAM. If the CAM is full then the transaction must be held until an entry becomes available. The CAM is very small, with only 8 entries. This value was chosen because large CAM structures are not feasible on an FPGA device due to their area overheads. As well as this the expected number of concurrent transactions with separate IDs is expected to be very small, as the round trip latency between the NI and DRAM is only ≈30 cycles. This means that only under extreme circumstances such as many-to-one collective operations with a large number of very short messages arriving in bursts could this become an issue. While the work presented here

does not treat collective operations in any special manner in the hardware, a simple solution could be sought segregating collective operations from this mechanism using a new entry in the *packet type* field in the header.

3.3.4 Shared Memory Communications and RDMA Transfers

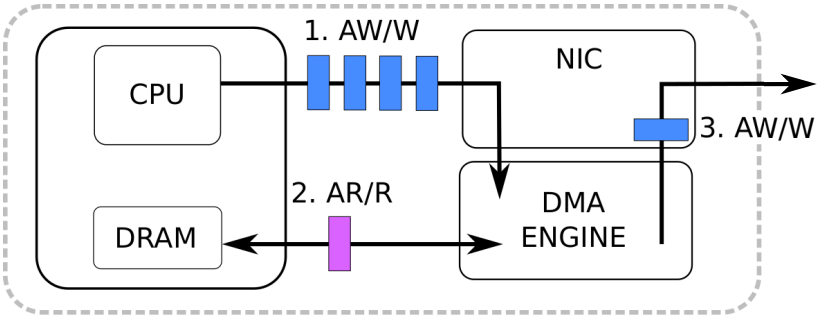
The Network Interface has completely separate AXI interfacing for issuing shared-memory, PGAS operations into the network than it does from standard RDMA transfers. The decision to have a segregated datapath through into the network comes from the disparate requirements of these communication models.

3.3.4.1 Shared Memory Operations

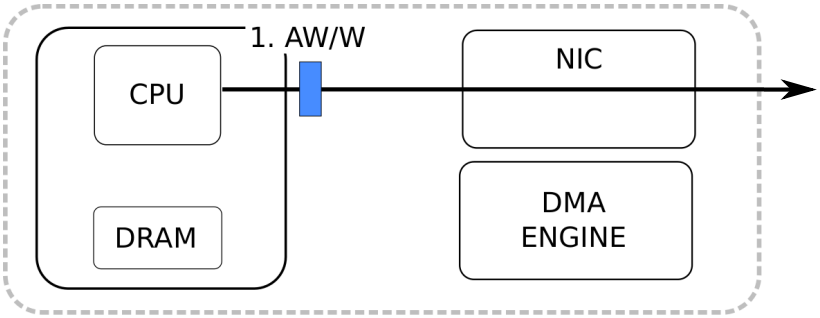
Reducing the latency of shared memory operations is critical as these often form synchronization tasks such as barrier and fence operations, and typically with small message sizes. Having the overhead (regardless of how low) of setting up an RDMA transfer for small operations such as these is unnecessary as the system will use a global virtual address space, with the uppermost bits identifying the target remote node's physical location geographically within the system. In doing this the processor can issue direct load/store instructions into the NI. These are presented as standard memory mapped AXI transactions at the interface between the PS and PL. The concept of this utilization of standard load and store operations into a global address space is presented in [15]. Figure 3.6a and 3.6b show how the setup for an RDMA transfer differs in terms of complexity from a simple read/write into the global address space. We evaluate the performance seen in our system when issuing these two communication types in Section 3.4.1.

3.3.4.2 RDMA Transfers

In order to perform RDMA transfers the user must write into a work buffer inside the NI. The DMA engine is the *AXI Central Direct Memory Access* IP core from Xilinx [16], which pulls and pushes memory mapped data. It is set up during initialization to continually pull work items from the NI whenever there is work to be performed. In doing this we are able to catch and manipulate the transfers



(a) Setup for an RDMA transfer, requiring multiple AXI writes and a read from main memory.



(b) Performing a shared memory operation (`STR` instruction) directly into the Network Interface.

Figure 3.6: RDMA setup and shared memory operation within the Network Interface.

within the NI, which allows for tracking of the RDMA operations. This is used as the basis for forming the hardware-offloaded transport layer, described in detail in Chapter 5. Once the transfer has successfully completed a notification is placed on a queue, which the local process can then poll to confirm that a transfer has completed.

Figure 3.7 shows the data which must be written into the work queue inside the NI to initiate a DMA operation. A 64-bit source address must be written, followed by a 64-bit destination address, and finally a control register must be written. This control register contains the number of bytes to be transferred, as well as setup information regarding the transfer addressing mode, interrupt thresholds etc.

There are currently two distinct buffers; one for work originating from the processing system, and one for work originating from the FPGA. This is done to allow for notifications to be built and provided to the correct initialiser. While this suffices for the purposes of this thesis within our prototype, it means that currently there is no method of assigning the same physical work buffer to multiple logical buffers.

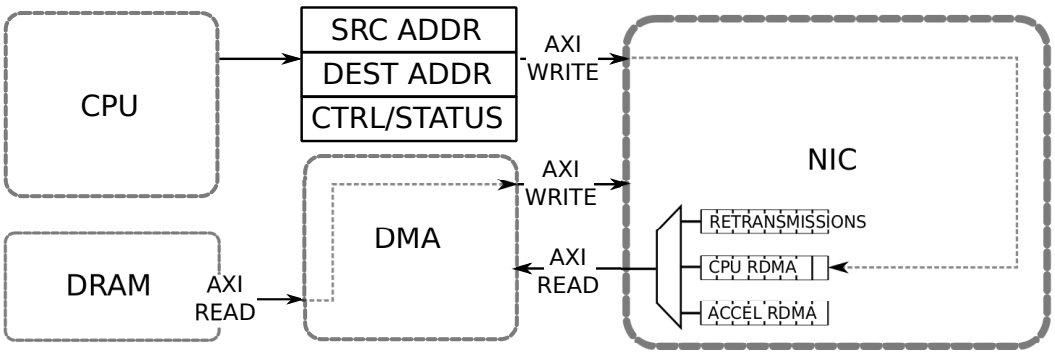


Figure 3.7: Mechanism for adding transfer descriptor to work queues within the Network Interface.

Only one process may currently take ownership of each work buffer. This can be easily remedied by returning an operation number which is assigned to the work item to the issuing process. This can be placed with the notification to associate a process to the completion of a specific work item.

3.3.4.3 API

The NI is configured and accessed using several user-space functions which allow an application to set up data transfers and confirm their success. A simple notification queue is currently used within the NI, forcing the user to poll the queue until a transfer has completed. In the event of a catastrophic failure in the system such as multiple failed delivery attempts, an interrupt must be raised to the system. This should be used to force the system into a checkpoint-restart procedure or data migration [188]. Tables 3.4 and 3.5 give an overview of the functions available to the programmer, and their action within the NI. All instances of `map*` refer to the base address at which the local FPGA resources are `mmap()`'ed into the application's memory space. The accelerator related functions are specific to the dummy accelerator block we use to test our network for direct inter-FPGA communications in Chapter 7.

Table 3.4: Data structures used in the API for the networking stack and accelerator blocks.

Structure	Members
`RecvStruct`	**uint64_t n**: number of bytes received, **void* data**: location in memory of data that has been received.
`AccelConfig Struct`	**uint64_t intermediate_bram_addr**: location to put results from work that has been performed, **uint64_t latency_computation**: artificial latency added to dummmy accelerator block (see Chapter 7), **uint64_t dma_queue_addr**: physical location of the RDMA work queue within the Network Interface, **uint64_t remote_bram_addr**: **uint64_t remote_work_dest_addr**: physical location of a remote accelerator work queue, **uint64_t remote_accel_bytes**: number of bytes which are to be snet to remote accelerator via RDMA. **uint64_t local_notification_addr**: location of local queue to place notification of competed work. **uint64_t local_or_remote**: Used in dummy accelerator only, determines whether this is local or remote accelerator. **uint64_t local_dma_notif**: **uint64_t remote_block_work_source**: **uint64_t remote_block_work_notif**:
`AccelWorkStruct`	**uint64_t src_addr**: source address from which to take data, **uint64_t bytes_to_dma**: number of bytes to pull to work on, **uint64_t dest_addr**: where to place the work once completed, **uint64_t dest_bytes**: number of bytes to push once work is complete, **uint64_t notification_addr**: address of queue to send notification of completed work.

Table 3.5: User-space functions for programming the Network Interface.

Function Name	**Action**
`int send(uint64_t * map, uint64_t src, uint64_t dest, uint64_t bytes);`	Puts an RDMA work item onto the Network Interface, returns 0 when RDMA operation is successfully initiated and data pushed into the network.
`int confirm_notif(uint64_t * map)`	Awaits notification of a completed RDMA operation form a previous `send()` operation, returns 0 if successful, returns 1 if timeout.
`int send_blocking(uint64_t * map, uint64_t src, uint64_t dest, uint64_t bytes)`	Puts an RDMA work item onto the Network Interface, and awaits acknowledgement of completion of the operation from the receiver. Returns 0 if successful, returns 1 if timeout.
`int recv(uint64_t* map, RecvStruct* data)`	Awaits notification of received data from a remote source, places data memory location and number of bytes received into the passed RecvStruct argument.
`quick_recv(uint64_t* map, RecvStruct* data)`	Awaits notification of received data from a registered receive operation from the Network Interface, places data memory location and number of bytes received into the passed RecvStruct argument. Returns 0 if successful, else 1.
`void config_accelerator(uint64_t* map, AccelConfigStruct* data)`	Configures an accelerator block to enable it to work independently of the CPU, arranging data movement following operations etc.
`int work_accelerator(uint64_t* map, AccelWorkStruct* data)`	Send a work item to an accelerator block. Returns 0 if work successfully placed on work queue, else returns 1.
`shm_write(void* addr, void* data, uint64_t length)`	A simple wrapper for a store instruction into remote memory.
`shm_read(void* addr, uint64_t length)`	A simple wrapper for a load instruction from remote memory.

3.4 Segregation of Traffic Types

As discussed above in Section 3.3.4, the system offers two methods of communication over the network; small low latency transfers in the form of direct shared-memory operations, and traditional bulk data transfer over RDMA. In this Section we perform some experiments to show the advantages to having a segregated datapath through the NI for these two different traffic types.

3.4.1 Small Transfer Latency

Typically smaller and more latency critical transfers for synchronization or control messages will be performed using the shared-memory data/control path through the NI, directly translating load/store instructions from the processor into network packets via local AXI transactions. Obviously this has reduced latency over the setup and transfer of similar small data via the RDMA datapath, due to the additional latency of writing to the DMA work buffer and pulling the data from local memory to be sent to the NI.

A simple experiment is set up to show the advantage of using the shared-memory datapath through the NI for small transfers as opposed to the setup and processing of an RDMA transfer. The setup on a single FPGA in loop-back mode is shown in Figure 3.8. This uses two completely independent Network Interfaces and data paths from the processing system in order to emulate exactly the action of sending a message over two distributed boards connected via a 10G SFP link.

In the experiment 16 bytes of data are transferred from an application in user-space to a BRAM located within the programmable logic of the "remote-side" of the FPGA. For the shared memory operation the time measurement taken is from the cycle in which the initial AXI transaction is seen in the PL to the cycle when the response packet is transferred back to the PS. For the RDMA transfer the measurement is taken from the cycle in which the first write transaction into the DMA work buffer is seen in the PL, until the return notification is seen once the transfer has been acknowledged.

The benefits of performing small transfers using the dedicated datapath for

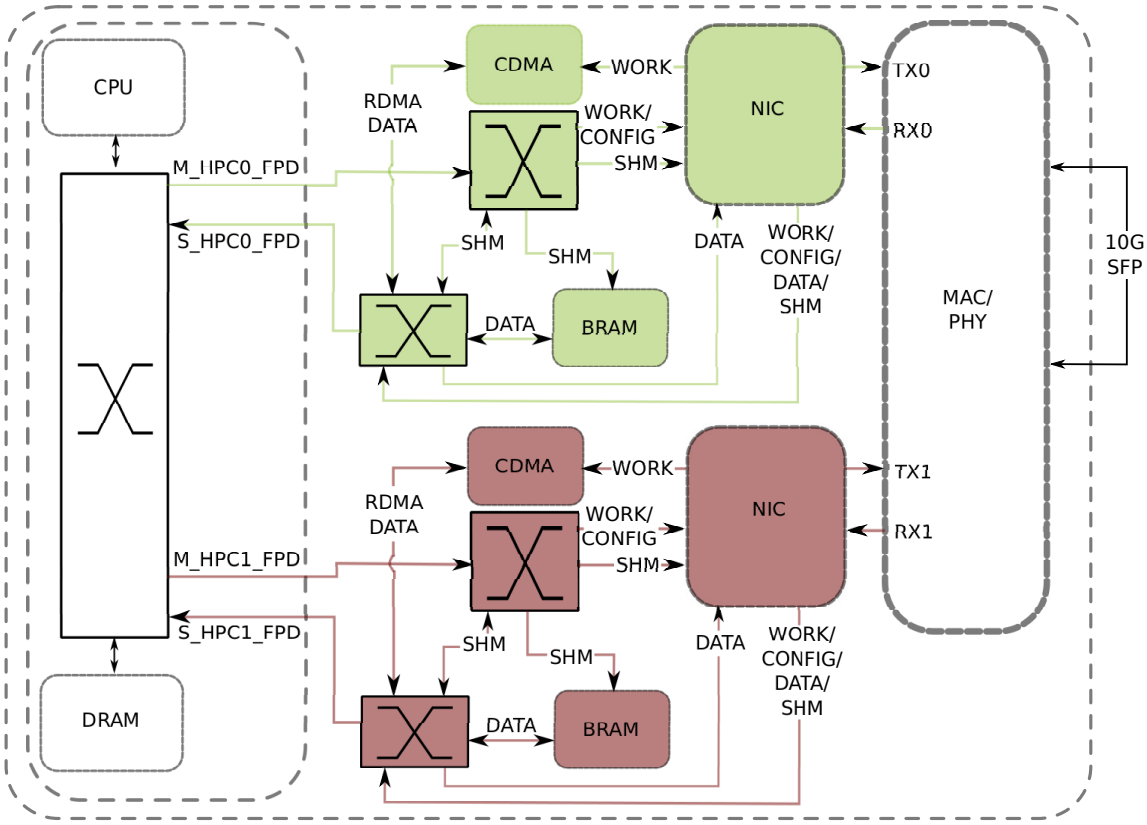


Figure 3.8: Setup emulating distributed system on a single FPGA, with logic for two implementations containing completely isolated address maps on a single FPGA.

Table 3.6: Latency for a small 16 byte transfer using the shared-memory datapath and the RDMA datapath through the NI.

latency component	**Shared memory (ACK'd)**		**RDMA (w/ notif.)**	
	cycles	**ns**	**cycles**	**ns**
Total	**172**	**1101**	**232**	**1485**
Initial write- last flit at NI output	24	154	69	442
Read from RAM	-	-	30	192
TX MAC in- RX MAC out x2	59	378	59	378
RX MAC out- Resp/Notif at TX MAC in	21	134	23	147
RX MAC out- Completion	9	58	22	371

shared-memory operations is clear, reducing the round trip latency for full acknowledgement of the write operation by over 25%. This value would be enhanced given the opportunity to optimize the retransmission table of the shared-memory operations. Currently only a single 64-bit value can be associated with a single table entry in the NI, meaning that to send a 16 byte transfer requires the NI to produce two packets (this is discussed in detail in Section 5.2.2). This increases the packet overhead on these operations substantially and adds over 10 cycles latency (≈5.5%) when compared with sending a single 8 byte packet. As the main area consumption in this module is not associated with the data storage but the CAM (Content Addressable Memory) which identifies the addresses of prior entries, increasing the amount of data storage capacity for individual entries is negligible. This would enable greater performance enhancement over that already seen.

As well as this there is a lot of additional overhead from the use of an off the shelf MAC/PHY layer. The Aurora IP constitutes 68.6% and 50.9% of the round trip time for the shared-memory and RDMA operations respectively. This is a shocking latency penalty, and merits the implementation of a custom physical and data-link layer. The ability of a custom protocol solution to dramatically reduce the latency of transfers over both standard Aurora and Ethernet solutions has been demonstrated in [11], although performing this work would be far beyond the scope of this thesis.

3.4.2 Shared Memory Throughput Limitations

While we have shown that shared-memory operations are favourable for short messages in which lower-latency is required, they are very unsuitable for heavy loads and are not intended for bulk data transfer. We have set up an experiment to demonstrate the shared-memory capability of the system, and show the performance implications of using shared-memory operations for data transfer. In this experiment two interconnected ZCU102 evaluation boards are used, connected using a 10G SFP link and routed using our custom switch design (Section 3.1.3). Each board is loaded with the same bitstream, and set up as shown in Figure 3.9. In this instance we use a small IP module, the *Address Remapper*, in order to adjust the target address to a correct value which will enable proper routing through the network switch. In a full scale system a TLB (Translation Lookaside Buffer) would

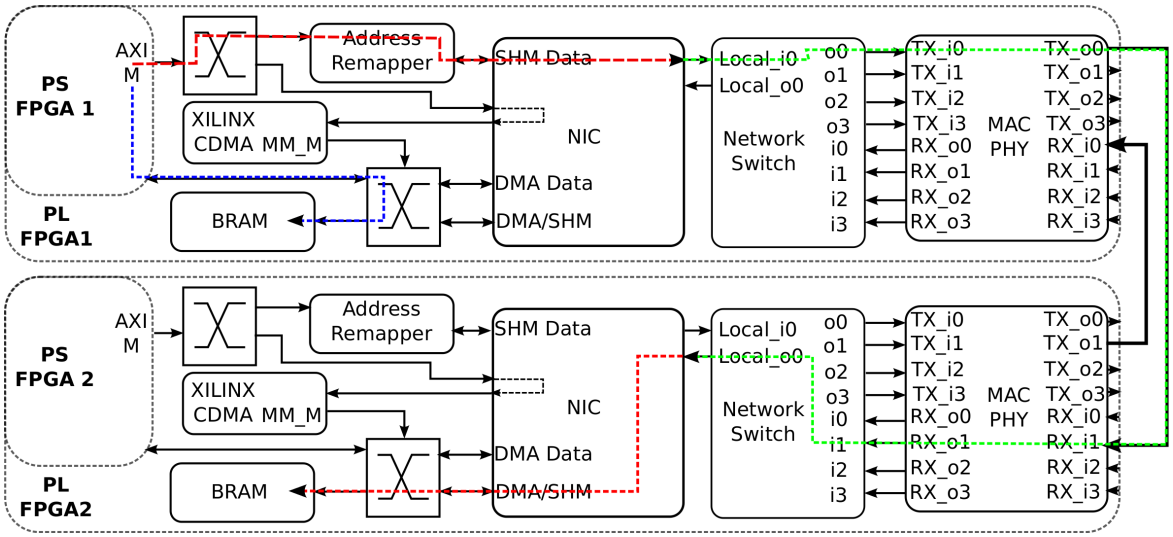


Figure 3.9: Distributed setup with two FPGAs communicating through the network switch, used to run the STREAM benchmark. Red and green shows the path through to remote memory from the local CPU, with the blue path indicating communication with local fabric memory.

be used here in order to translate from a physical address sent out to the PL to a virtual address in the global address space. In our prototype we simply map the upper bits to be routed to the correct port of the switch.

We configure the address mapper to send to address 0x0520140011110000 (upper 22-bits 0x00014805). Configuring the *local port* value of the two routers thusly will emulate the routing of the packet to a node in another chassis of the network:

	Address	Cabinet	Chassis	Daughter Card
Switch 1	0x00014403	00000101	0001	0000000011
Switch 2	0x00014805	00000101	0010	0000000101

This means that the packet will be routed out of the network switch over the first output (port o0/TX_o0), towards the node in chassis 2. As the receiving node matches the node ID in the destination address we route out of the second switch through the local port (Local_o0). If the packet was destined for a different cabinet the packet would be directed out of the uplink port (port 3 in this implementation).

The experiment we run shows the discrepancy in bandwidth that can be expected when accessing remote memory as a simple shared memory operation. The STREAM benchmark [189] was run on one of the nodes. This benchmark runs a series of memory operations using different access patterns in order to test the whole memory subsystem. The code was compiled from C source using GCC with the

Table 3.7: Results for running STREAM benchmark.

PS DRAM				
Function	**Best Rate (MB/s)**	**Avg Time (s)**	**Min Time (s)**	**Max Time (s)**
Copy:	3345	0.0489	0.04784	0.0497
Scale:	1826	0.0887	0.0876	0.0905
Add:	2033	0.118	0.118	0.119
Triad:	1683	0.144	0.142587	0.151
Local BRAM				
Copy:	45.4	3.52	3.52	3.53
Scale:	44.0	3.64	3.64	3.64
Add:	44.2	5.43	5.43	5.43
Triad:	44.6	5.38	5.38	5.38
Remote BRAM				
Copy:	4.8	33.6	33.6	33.6
Scale:	4.7	33.8	33.8	33.8
Add:	4.7	50.8	50.8	50.8
Triad:	4.7	50.8	50.8	50.9

-O2 optimization flag, with OpenMP multithreading switched *off*. The results are shown in Table 3.7 and show the achievable memory throughput for the benchmark in three configurations:

1. Memory is allocated in local DRAM accessed directly through the interconnect of the Processing System and the cache hierarchy.

2. Memory is allocated in local Block RAM (BRAMs) inside the Programmable Logic, having to be accessed through the Full Power Domain AXI master port (see HPM in Figure 3.3).

3. Memory is allocated in a remote BRAM on the second board, accessed through the network.

We see that the degradation in performance is significant (≈2 orders of magnitude) when we have to use the PL to access memory. There are several factors which cause these poor results. As the user accesses this memory transparently as a simple STR/LDR instruction, the limitations of the processor come heavily into play. As we discussed previously, the ARM Cortex-A53 is an in-order dual issue CPU, meaning a maximum of only two read/write transactions can be in flight at any point in time. The processor's pipeline will block until it receives the acknowledgement of successful write or the data from a read operation. If we were to use

an Out-Of-Order core then this limitation would be dependent on the issuing depth and the number of concurrent shared-memory transactions which can be handled by our NI. This is dependent on the size of a CAM used within our implementation which is discussed in detail in Section 5.2.2.

Another cause of the performance degradation is the fact that the entire memory hierarchy can be used when accessing the DRAM in the PS. By accessing the PL through the AXI master ports we are unable to cache the resulting data. Obviously this is required as there is no way of keeping track of stale data in the PS side. Since it cannot be known what a given hardware block or remote processor is doing at any point in time without exclusive access it cannot be known that the data remains valid in the cache and so must be marked uncacheable.

It is shown that the throughput to access remote memory is about an order of magnitude slower than using local PL BRAM. This makes sense from the observations on the basic latency of a remote shared-memory access and given the CPU blocking. We have previously observed that it takes around 150ns round-trip time for a simple AXI write/read issued from the CPU into a *local* BRAM in the programmable logic. This is around an order of magnitude less latency than we see in the round-trip time for a remote operation (see Table 3.6). Given the blocking nature of these operations the latency in acknowledgements causes the degradation of the performance we observe here.

It is obvious that this model of communication is not suitable for large data transfer between remote memory regions and that the RDMA operation is required to provide a far more appropriate method to transfer larger chunks of data. The shared-memory path to the network here is only suitable for simple low-latency communication between processors, as typically seen in synchronization messages or control packets. The advantage in having this kind of communication present in the system is that the latency of giving access to the user of the whole memory space allows us to reduce the latency of transfer for these small packets. It is also worth noting that the NI prioritizes the forwarding of the shared memory requests over RDMA transfers. It is able to do this very simply due to the separated data paths through the network, so no additional software overhead is incurred by marking

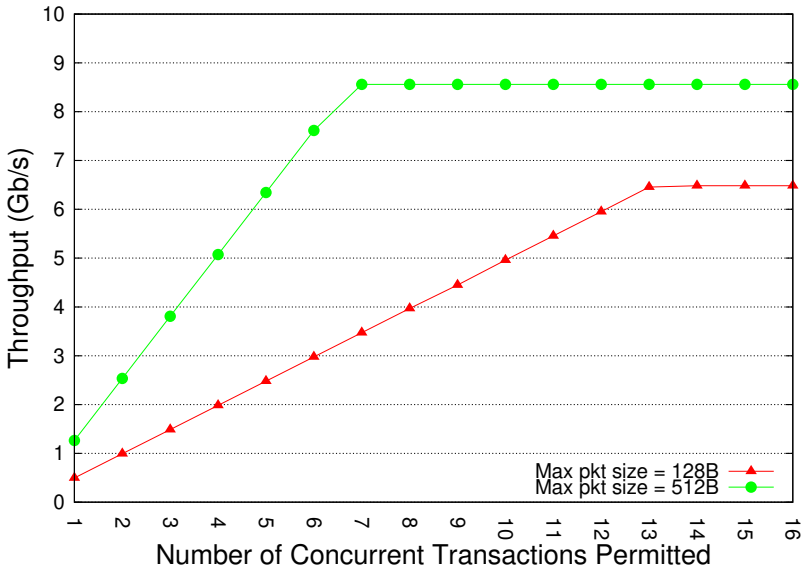


Figure 3.10: Throughput over number of concurrent packets permitted from single source in network, shows saturation.

these packets manually as high priority. Due to the limitations of the shared memory operations for larger transfers our system will rely on RDMA for bulk data transfer, which we test in the next Section.

3.4.3 Testing Link Throughput

In order to test the performance of larger RDMA transfers through the network we use a traffic generator to vary the packet size and number of possible simultaneous in-flight transactions to show the maximum attainable throughput of the system over a single link. Figure 3.10 shows the increase in the achievable throughput when transmitted over our NI and custom switch by varying the number of possible outstanding transactions the DMA can issue simultaneously, and by varying the burst length (and thus network packet size) of the DMA transfers. By increasing the maximum payload size from 128B (16×64-bit) to 512B 64×64-bit) we are able to increase the saturation point for the throughput by ≈32%. Using a packet size greater than this may have adverse effects on the network. Keeping packet length relatively short allows us to perform better load balancing (and will thus aid in lowering the congestion in the network). The increase in throughput is due to the packet header and footer, which contribute 4 flits of overhead on the link per packet, given our 64-bit datapath. This gives a reduction in the packet overhead

from 20% (20/16) to 6.3% (68/64). If we were to increase the data width to 128-bits (reducing header and footer to a single flit each) the packetization overhead will drop further still to 3.1%.

The saturation occurs at 8.56Gb/s, which is seemingly lower than the 9.4Gb/s promised by the GTH transceivers (given our packet overhead and the link being run at 10Gb/s). However, this additional drop can easily be accounted for. The MAC layer adds a preamble at the beginning of transmission and an end-of-packet flit at the end. An inter-frame gap is also required between packet transmission to aid in keeping the transceiver clocks synchronized. This results in a raw throughput drop of about 9% in our specific case (though this is dependent on frame size). It is unclear as to whether a customised MAC/PHY layer would make any significant difference to this value. These idle cycles are required to account for the difference in clock rates which may be seen between the transmitter and receiver, synchronizing the two. Although this throughput is around 15% lower than the raw link rate, we deem it acceptable given that this value is actually the *goodput* of the NI, switch and link, having taken into account the packet overheads and leaving strictly the payload data.

3.5 Concluding Remarks

In this Chapter we have presented a novel hardware networking stack which enables modern FPGA devices to communicate with distributed shared memory resources. Hardware primitives are provided to support two separate models of communication; NUMA-style read and write operations into remote regions of memory, and RDMA for data transfers between remote nodes. The author's specific contributions to the work presented in this Chapter are as follows:

1. Design and implementation of a novel NI microarchitecture which enables (i) Transparent read/write operations (in user-space) to regions of remote memory/accelerators, as well as (ii) User-initiated RDMA operations to allow high throughput data transfer to remote memory. This is the first such network infrastructure of which we are aware that provides direct hardware support

for these two communication primitives together in a reconfigurable environment.

2. A low-latency protocol translation mechanism between the on-chip memory-mapped AXI and a custom interconnect solution intended for networking in High Performance Reconfigurable Computing. This allows for transparent use of *remote* memory-mapped AXI transactions.

3. Implementation of a hardware prototype able to perform full end-to-end remote memory operations between two MPSoC devices, connected via 10Gbps serial links.

4. Experimentation which shows basic performance metrics of the system. We show the clear benefits in small transaction latency from our dedicated shared-memory datapath, as well as showing the level of performance degradation seen in shared-memory operations when repeatedly accessing remote memory using the in-order A53 processor.

Chapter 4

Error Recovery and Memory Consistency

In this Chapter we evaluate the different types of error which can occur within our network, and evaluate the ways in which they can be mitigated. Error in this context refers to bit-level errors which can occur on the high-speed serial links during network traversal. As a result of these errors packets can be misrouted or lost, but we do not allow for regular packet dropping due to buffer overflow within the network (as happens in Ethernet networks). Although not currently implemented (and beyond the scope of this thesis) a link-level, credit-based flow control mechanism is envisaged in order to prevent packet dropping. Buffer overflow is by far the main source of lost packets. We do not concern ourselves with the flow control technique employed, as this work is pushed beyond the Network Interface into the switch. Owing to our use of a connectionless transport it may be unnecessary to handle the flow control mechanism at an end-to-end level. Rather, it makes sense to utilize link-level credit techniques to prevent buffer overflow, given that we can utilize the full path diversity of adaptive routing at the packet level [32].

There are numerous types of error which are made problematic due to the ability of the system to deliver Out-Of-Order (OOO) packets. This is a consequence of the connectionless transport layer (presented in Chapter 5) and the switch architecture which permits full adaptive routing [32]. We show that this is not an issue in all but the most exotic cases, due to the fact that our system writes directly into target memory locations rather than copying from send/recv buffers at the Network

Interface.

An important note regarding this analysis is that the fallback of checkpoint-restart mechanisms discussed are to be a *very* rare occurrence indeed, being used when multiple retransmission attempts have already been made. Given the statistical improbability of this happening repeatedly to the same packet given modern link error rates of $\approx 10^{-12}$, it is safe to assume that severe network failure or node failure is the likely cause of such an event. The checkpoint-restart mechanism we make reference to in these cases is an application level fault tolerance technique; the implementation of which is far beyond the scope of this thesis since it falls outside the realm of hardware techniques and into much higher level reliability mechanisms.

4.1 Shared-Memory Operations

Shared-memory operations are typically formed of very small messages and should not be used with sufficient frequency to overload the NI with requests (typically these would be unicast or many-to-one messages). As such the data to reproduce these transactions is stored within the NI. In the event of a negative acknowledgement (NACK) being returned the entire message can then be rebuilt without involving the CPU. Multiple failures/timeouts of these shared-memory operations requires the assistance of the CPU.

4.1.1 Remote Read

If a remote read transaction posted through the shared-memory interface of the NI repeatedly fails then the result for the system is catastrophic. If the initiator of the transaction is the CPU, in this instance there is no way to recover as the processor will stall until data is retrieved, due to the in-order pipeline of the CPU in our system (ARM Cortex-A53). Even in an out-of-order processor progress can only be made until the point where we hit data dependencies on the missing values or the depth of out-of-order issuing has been reached.

If a NACK is encountered after multiple retry attempts (ruling out the possibility of misrouted packets), then the error must originate from a failure on the

receiving node, from the network fabric, or from a programming error. The NI must be used in order to handle this situation. The CPU is incapable of being returned an incorrect value (NACK'd) from a load instruction; the processor simply hangs in this instance. In order to prevent lockup of the system we must provide a positive acknowledgement with fake data from the sending NI to the CPU. After this has taken place we must interrupt the processor and initialise a software-based checkpoint-restart mechanism in order to restart the program safely.

Unfortunately this method means that debugging the error could be more difficult given the fact that the processor will have moved on from the offending instruction, making it harder to locate the error. The NI can be instrumented to retain information regarding the failed transaction and mitigate against this. In this instance we envisage that the interrupt routine for handling this error can read this information back from the NI and log the error.

4.1.2 Remote Write

Repeated failures of a shared-memory remote write transaction forms an equally catastrophic failure condition, requiring the initialisation of a checkpoint-restart mechanism which must be present in any HPC system beyond a certain scale [188]. In order to enhance the performance of the system and prevent stalling of the in-order pipeline of the ARM Cortex-A53 the NI is instrumented to perform an early ACK of remote store instructions (see Section 6.1 for more details). The NI provides an acknowledgement and we must store the data to rebuild the transaction in case of retransmission. In this instance the processor can carry on, but software mechanisms must be used to confirm with the NI that the operation has successfully completed or whether it is still in flight.

If the early acknowledgements feature is disabled then an error here does not prove as much of an issue for debugging as for remote read operations. This is because returning a negative acknowledgement to the CPU for a store instruction results in a segmentation fault, crashing the program and automatically invoking a checkpoint-restart mechanism. If the early ACK system is in use then the NI must interrupt the CPU after multiple retry attempts, in the same manner as an error in remote read operations.

4.1.3 Acceleration Resources Performing Shared-Memory Ops

If a (repeated) failure is seen in a remote read/write transaction issued by an acceleration resource, the course of action is much the same as described above. Some of the issues with pipeline stalling can be made less problematic as the accelerator can be permitted to issue a higher number of transactions than the dual-issue Cortex-A53. The significant problem to be overcome in this instance is when the FPGA resources are completely disaggregated from the CPU. In the instance where a network attached FPGA is working without any CPU resources[1] provision must be made within the FPGA to transmit a failure message to associated CPU resources, allowing it to take control of the situation. This could be done within the NI in order to prevent an increase in complexity in acceleration IP designed by the end-user. The NI is aware of the origin of the transaction and so can change behaviour based upon the initiator of the transaction. If the origin is an acceleration resource it would be simple to change the error-raising mechanism.

4.1.4 Exclusive Accesses and Consistency

For remote write operations the early acknowledgement system (detailed in Section 6.1) can obviously cause some consistency issues if extra steps are not taken. There are two ways in which consistency can be enforced for shared memory communications:

4.1.4.1 Software Based Consistency

The first is to use the early ACK system for performance to free the CPU to perform other work. In this instance the programmer must communicate with the NI to ensure that any subsequent work which is dependent on the successful completion of the operation is able to do so. For example, this may occur when exclusive access on a shared resource is required and the CPU cannot write data to this resource until it has locked access. These software locks can be very common in distributed applications, and this is a typical way to gain access.

[1] We can easily envisage this situation in a configuration where multiple large FPGAs are used for acceleration with a single CPU being used for management, rather than having a network of CPUs with FPGA logic on the same die.

The use of this technique is a common way to see performance gains by hiding messaging latency in the CPU. It is required of the programmer when writing one-sided communications in MPI for example [67], where MPI models completion, consistency and synchronization entirely separately. The result is that the programmer must reason about the possible data dependencies in their program and explicitly handle cases where data races can occur with barrier and synchronization methods. This is typically required of HPC programming models, because implementing strict sequential consistency at the hardware level is far too expensive.

4.1.4.2 Hardware Exclusive Accesses

The second way we can maintain consistency is via the use of a hardware exclusive access operation (LDREX/STREX in ARM's Thumb assembly) which when used via an AXI interface use the AxLOCK signals (see Appendix C). In the case where an exclusive access is seen on a write transaction (i.e. the AWLOCK signal is high) we bypass the early ACK system within the NI, forcing the CPU or other resource to wait for the real ACK packet from the network before returning it to the initiator. This gives the programmer the option of providing a hardware oriented exclusive access where delivery is guaranteed on receipt, or a more relaxed software version in which other non-dependent work can be performed while awaiting a full response.

When an LDREX instruction is issued an *exclusive monitor* in the CPU is written with the associated process and transaction ID, in order to prevent access from others[2]. The monitor does not allow other accesses until a corresponding STREX is seen from the same master device. The issue with using these locked accesses is that only a single AXI bus is supported; the function is not intended for use over a networked system. The processor prevents unauthorized access based upon the IDs, but is unaware of the initiating node. This means that in order to allow this functionality the Network Interface must track all inbound exclusive accesses at the receiver, providing its own monitor which tracks not only IDs but also the source node of a transaction. This is because the two different nodes may use the same

[2] ARM Software Development Manual- `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dht0008a/ch01s02s01.html`, accessed April 2019

outbound ID, but obviously originate from separate AXI buses.

With respect to the failure of these transactions, it is simple to differentiate between the transaction failing correctly (due to a prior locking of the resource from another node) and incorrectly (due to bit-errors etc.). This is because the error code returned will be different. In the case of an exclusive access failing due to prior locking, retransmission can be attempted until the resource is free and normal steps can be taken if the failure is a repeated CRC failure or lost packet for example.

4.1.4.3 Performance of Exclusive Load

As described above, the remote read must register the exclusive access in the remote NI, and return an error if exclusive access cannot be gained (another process/node has access locked). Following successful registration a corresponding exclusive store will eliminate the entry in the NI. Unfortunately there is little opportunity to enhance the performance of these operations if two nodes contend for the same resources, as the result of the read operation is obviously required before returning the response. The system and programmer must therefore aim to minimize the number of remote memory accesses, which is a well established practise for NUMA-style systems anyway [190].

It may be of some use to cache failed LDREX instructions at the receiver and stall on sending a NACK back to the sender. This could be done in order to allow for the operation to be performed once the node with exclusive access has unlocked the resource. Doing this could decrease the incidence of multiple retransmissions on these accesses. However, it could introduce certain issues such as duplicate packets in the network. An example of this is if the timeout for retransmission is much shorter than the amount of time a node may require exclusive access to the resource.

It could also be useful to store the data that has been read from remote memory in the sending NI, thus enabling multiple read/writes to this "remote" location to actually comprise local accesses to the NI hardware, reducing latency for the operation. This will prevent use of the network for multiple processes within a single node attempting to access the same resource. However, this can obviously create consistency issues if the data is able to be cached at another remote node

(i.e. a normal non-exclusive access). Also this does not prevent a remote node's access travelling over the network before failure/success of the exclusive access transaction, unless the access were from another process local to the node which has exclusive access rights.

4.1.5 Summary

The following Table 4.1 shows a summary of the errors which can present themselves in shared-memory operations, and the ways in which we can mitigate against them:

Table 4.1: Shared-memory operation error types, the complexity for handling them in the NI, and their mitigation.

Error Type	**Complexity in NI (1-3)**	**Mitigation (following multiple retransmission attempts)**
Remote Write	1	Return error and let program segmentation fault occur.
Remote Read	2	Return incorrect value with non-error code, store debugging information in NI, raise interrupt to local CPU.
Remote Write Early ACK'd	2	Store negative ACK in notifications and await CPU check, or store debugging information in NI, raise interrupt to local CPU.
Accelerator Write	3	NI returns error, send error information to designated CPU in group, or to centralised accelerator error handling. CPU must then interrupt and begin checkpoint-restart.
Accelerator Read	3	NI returns error, send error information to designated CPU in group, or to centralised accelerator error handling. CPU must then interrupt and begin checkpoint-restart.
Load Exclusive	3	NI must differentiate. If access error, retry. If network error return incorrect value with non-error code, store debugging information in NI, raise interrupt to local CPU.
Store Exclusive	2	NI must differentiate. If access error, retry. If network error return error and let program segmentation fault occur.

4.2 RDMA Data Errors

Due to the fact that it is possible for large amounts of data to be processed through the RDMA mechanism, the shared-memory reliability model cannot be followed in this instance. The required buffering for outstanding transactions would be too great a burden for the NI to bear. Instead we track RDMA operations and outstanding transactions, enabling us to rebuild any negatively acknowledged packets.

In the event that an RDMA packet fails after multiple attempts to reach its destination the CPU must again be invoked into handling the issue. This is because a repeated failing RDMA operation can only come from either a severe network failure, node/DRAM failure (assuming program correctness) or some other similar hardware issue. However, once the system is interrupted other methods can be brought into play to solve the issue as opposed to checkpoint-restart. This is because an RDMA operation in and of itself is not fundamental to the operation of the program, it simply prevents progress with computation.

Fault detection mechanisms can be used to identify the problem [191] and possible migration of the process and data to another node can be performed [188], at which point the operation can be retried. This causes significantly more issues if the originator of the transaction is an acceleration resource. As the accelerator will need to pass state information to a CPU in order for the CPU to get the accelerator and its associated NI to a state where it can attempt the failed RDMA operation once again.

4.3 Header or Footer Errors

Errors within the header and footer of a packet can cause numerous problems with the system. If the upper portion of the source or destination address become corrupted then the packet or its response can become misrouted. Due to the geographic routing scheme, if an illegal address is seen within the system then this could theoretically cause livelock; where the packet is continually redirected in a loop round the system.

There are a number of mechanisms to avoid this situation. The first includes

packet ageing, where the packet is only permitted to travel a given number of hops before it is dropped. Inclusion of this mechanism in the switch requires modification of one of the packet fields and incrementing an *age* value, although this would require use of a number of reserved bits in the header. One issue here is that this must be placed in the header in order to prevent the requirement of store and forward switching within the network. Another more important issue is that this method takes no account of the buffering time required in the switches. A packet may be held in a single switch for a substantial amount of time but not incur any ageing.

A second method to combat this issue would be to use the concept of an *epoch* within the system (similar to that used in the arbitration mechanism presented in [192]). Each packet is sent within a given epoch and labelled as such, and if the system then rolls over to an epoch too far in the future and the packet remains in the system it will be dropped. This method means that packets will age with buffering, but may be difficult to implement within a large-scale system due to the need for synchronization. Every switch and endpoint must be aware of the current epoch and roll over simultaneously.

In a situation where the packet header or footer is corrupted, if only a single packet level CRC is used then a negative acknowledgement cannot be routed back to the source because the source address may be corrupt. Due to this fact, and because maintaining correct header information is so important to the functioning of the network, an Error Correcting Code (ECC) is used to protect the header. This can be used end-to-end in order to fix erroneous packets. If the packet has been previously misrouted then it can then be directed to the correct node. The additional overhead of protecting the header and footer is not great, given the importance of maintaining correct information in this portion of the packet. Alternatively error correction on the header could be performed at the link-level as the check and correction can be performed while the remainder of the packet is being received, thereby not incurring a store and forward penalty. In this instance the switch will see additional complexity, which would be highly dependent on the given implementation [193].

4.4 Out-Of-Order Packet Delivery

To enable a lightweight implementation we require a connectionless transport for minimizing state information within the NI. As well as this the switch architecture targeted for the system (see Section 3.1.3) allows for fully adaptive routing at packet-level granularity. What this means is that out-of-order packet delivery for normal and retransmitted packets is a very real possibility. Ethernet and Infiniband networks do not have to deal with this issue as they do not allow for adaptive routing at this level. Given that they use table based routing, the route over the network between any connection (TCP) or Queue Pair (Infiniband) is fixed, so necessarily the packets arrive at the destination in-order. This is at the expense of a reduced ability for dynamic load balancing within the network.

While it may be natural to think that the burden of reordering packets at the destination limits scalability at the receiver, we show that this is not true for our system. While there may be exotic cases in which maintaining packet ordering is required, given our communication model, network protocol, and framed in the context of a fully global address space; in the majority of cases ordering is not an issue.

We identify four possible types of destination for shared-memory and RDMA transfers, with only one of these requiring proper sequencing. Every packet travelling over the network originates from a memory-mapped transaction and is translated back into a simple memory-mapped transaction at the receiver, destined for some location within a global address space. The network packet includes the source and destination node IDs corresponding to their geographic location, and the destination memory address. This means that we can acknowledge the individual portions of a much larger DMA transfer for example, as each segment of the transfer forming these packets generates individual target addresses (see Figure 4.1). Effectively if the sender receives a NACK on a packet it will be able to calculate the original source address and be able to resend just that portion of the transfer. The receiver is notified of new data when each of the segments of the DMA transfer is in place. In this manner reordering occurs naturally within memory, as the "sequence number" for the transfer is just a virtual memory address, offset from

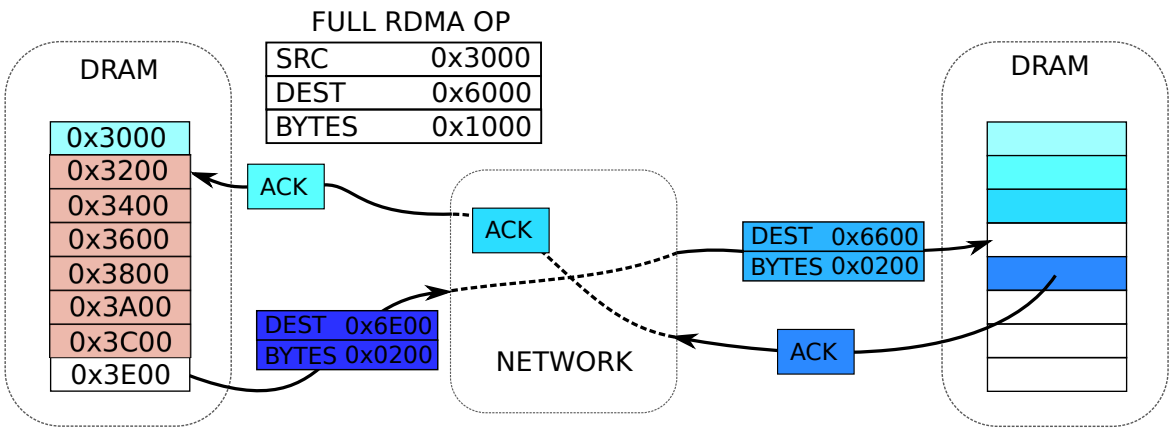


Figure 4.1: Ordering does not matter for RDMA operations directly into memory, as each packet has complete destination memory address associated with it.

the base of the transfer.

The first identified destination type is the situation described above, where a simple RDMA is placing data into the main memory of the system. We see that packets are arriving out of order, but because the destination is simply a memory location this does not matter as the target memory address is present in the packet header. This is the same manner in which the Cray Aries Interconnect is able to handle out-of-order delivery in block transfers [102]. Likewise, ACK packets contain special bit fields within the ID they carry, which enable the original transaction to be identified and reconstructed or retrieved (depending on the traffic type) for retransmission.

In Figure 4.2 and Figure 4.3 we see an example of map and reduce type operations. These operations are used heavily within data-centric HPC applications and may be suitable for FPGA acceleration in a tightly coupled SoC environment with CPUs [9], [194] (as we demonstrate in this thesis). In both these instances the operations are commutative, so permutations on the arrival of data will not affect the final result.

In 4.3 we see some function performed on the input data, perhaps for a database operation. The input to this function may be a large data structure, spanning multiple packets. In this instance ordering of the data is important, but not within our system. Given that access to the network is given via a memory-mapped interface, it is possible to use the address offsets for a data structure being sent to determine the boundaries of each key for the map operation. All that is required in

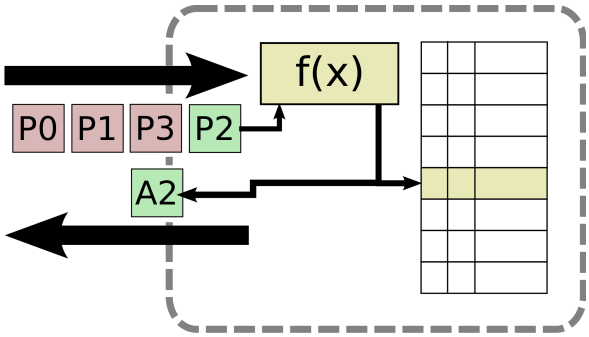


Figure 4.2: Map type operation. Transaction ordering is not required in this instance given the commutative nature or the operation.

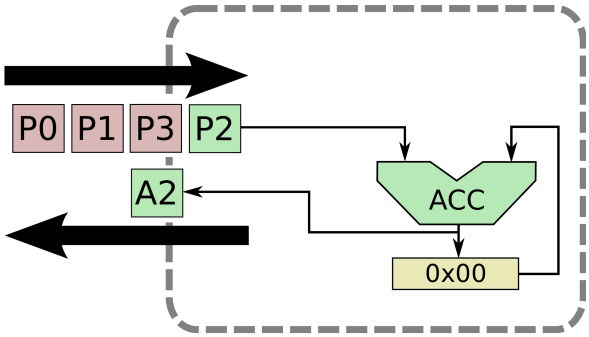


Figure 4.3: A reduction or other typical many-to-one type operation, where ordering is irrelevant.

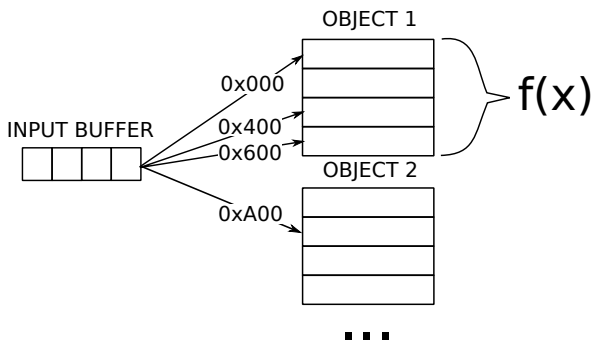


Figure 4.4: Mapping with large keys is still possible without guaranteed ordering within the network, as addressing offsets can be used to reorder easily.

this instance is a small amount of additional logic to create buffers which naturally rebuild the key, as opposed to a simple register map being used. This is shown in Figure 4.4.

Finally, in Figure 4.5 we see data being sent to a non-commutative operation; in this example an FIR-filter. In this sole instance is data ordering of real significance, as the output depends on the sequence of the input values. Reordering at the receiver is required in this instance. However, we only expect to see instances such

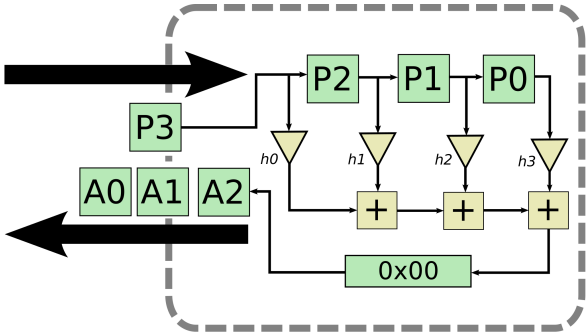


Figure 4.5: An FIR filter, non-commutative operation where the ordering of individual values is important. This sort of operation will require a dedicated reorder buffer, but is atypical of many-to-one operations, so concurrent multiple user access this would be unusual.

as this in very specialized hardware blocks in the accelerator and it would not be typical of most memory-memory RDMA type operations for HPC communications. Much more typically this sort of computation would be performed locally on data being streamed from/into memory where ordering would normally be maintained.

These sorts of operations would also typically not be involved in many-to-one type communications, and we are unaware of any such example in the literature. As such it is reasonable to assume that any specialized block in this instance can contain re-order buffers to handle OOO packet delivery. Only one such buffer will be required. This buffer's ownership will be acquired and locked when the block is used. Using a single reorder buffer here is a reasonable way to handle OOO packets, as multiple masters will not be expected to communicate with this sort of block simultaneously.

From the send side, the DMA engine has a mode of operation known as "keyhole" read and write. This allows full DMA operations to target a single memory address as if streaming to a FIFO. In the instance where a node wishes to communicate with this sort of hardware block the *keyhole* write mode can be used. Thus allowing the NI to identify this type of operation and act accordingly, perhaps placing sequencing numbers as additional flits in the header, allowing the reorder buffer at the receiver to operate.

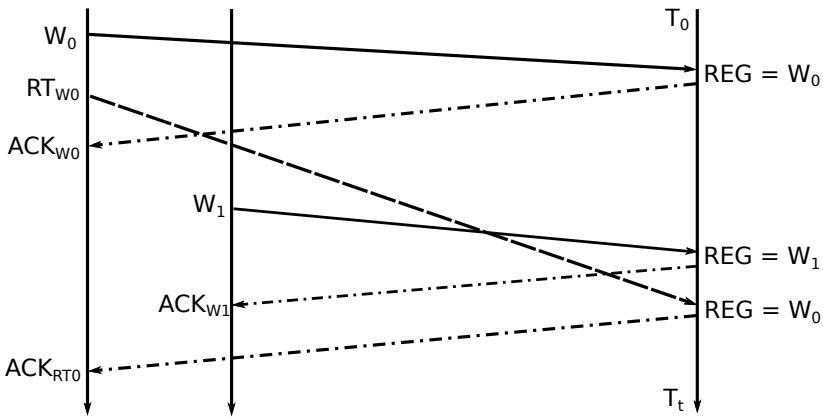


Figure 4.6: Duplicate packet occurring due to retransmission.

4.5 Duplicates in Retransmission

The overwriting of previously acknowledged data can occur and cause consistency issues in the situation described in Figure 4.6. Here, a packet W_0 is sent, and the ACK is returned as normal, but takes sufficiently long to cause a timeout at the sender. A retransmission RT_{W0} occurs, and another write operation W_1 (possibly from a different source) is issued to the same location, arriving before the packet RT_{W0}, which then overwrites the new value upon arrival.

There are several ways in which duplicate packets could be protected against in end-to-end retransmission, for example those described in [172] or [195]. However, these methods either suffer from scalability issues or severely impact the useful bandwidth of the links. In a typical RDMA transfer this problem is mitigated naturally because the sender sends an ACK packet to the receiver once it has received a positive ACK from all the individual packets within the DMA transfer. Only at this point will the receiver begin working on the data which it has been sent.

While eliminating this issue, this method of RDMA acknowledgement requires an additional round-trip latency for the ACK packet between sender and receiver. We have devised an improvement to the architecture which enhances the performance of the RDMA by allowing the receiver to track RDMA operations targeting it. It provides receipt of a successful RDMA transfer to itself locally, eliminating the need for the additional round-trip latency for the ACK packet. However, this means that duplicate packets must be handled differently in this instance. In Section 6.2 we present the architecture which allows for this optimization and discuss

how the duplicate packets are mitigated against.

4.6 Concluding Remarks

This Chapter provided a review of the possible errors that are introduced to our network as a consequence of some of the properties of our system. We have shown how to mitigate against these using a relaxed consistency model, pushing responsibility for synchronization onto the programmer in some instances. We identify areas where the performance can be enhanced and show the effects of this for consistency and correctness. The main takeaway points from this chapter are:

1. We allow for out-of-order delivery of packets within the network. This does not cause issues in all but the most exotic circumstances, which are atypical of many HPC workloads. This is mitigated against in most cases due to the direct addressing of memory.

2. We describe the need for strong protection of the header in order to prevent misrouted packets and livelock. What we mean by this is that an error correcting code (ECC) will be placed within the header, and used to detect and recover from errors in destination address and other integral header information. Forward error correction can either be performed end-to-end with slight modifications to the packet, or at link-level at the expense of additional latency of error checking and correction as well as additional switch complexity. Due to the fact that the foot of the packet does not require this protection to prevent these issues, store and forward would not be required in the network to perform link-level error checks, meaning that we can retain virtual cut-through switching.

3. We introduce several performance enhancements later in this thesis (the technical contributions for which are presented in Section 6.1 and 6.2). In this Chapter we have provided contributions in our discussion of the issues that these enhancements can cause with regards to consistency, and how the issues are mitigated against.

Chapter 5

Transport Layer for FPGA based HPC

Previously (Chapter 3) we have discussed the basic Network Interface developed for HPC communications within networks of reconfigurable devices. In this Chapter we expand on this work, introducing a novel transport layer which allows reliable, connectionless communications directly between CPU or FPGA fabric and remote resources.

Our solution is fully offloaded into the hardware of the NI, requiring no software intervention in order for the accelerator to issue reliable transactions into the network, and no OS intervention in order for the user to issue reliable transactions to the network. In doing this the FPGA accelerator resources are disaggregated from the CPU and the user application avoids costly OS system calls. Thus enabling much better capabilities for dataflow type applications over distributed FPGAs and for lower latency communications between distributed resources. As the RDMA data cannot be stored within the NI due to memory limitations on the device, and the shared memory data cannot be sent off-chip due to latency overheads, the action of these two types of transport is completely segregated, just as the data/control paths are segregated within the NI. We present the design and implementation of these two mechanisms, and of a fault injection mechanism used to test different fault tolerance mechanisms.

Our analysis at the end of this Chapter in Section 5.3 suggests that under varying bit-error-rate conditions an end-to-end reliability mechanism is preferable in terms of performance over link-level error checking/correction for all but the most adverse conditions (bit error rates $> 10^{-7}$). This is particularly true when accounting

for the fact that costly retransmission buffers are required for link-level error checking and retransmission, or additional bandwidth overheads are required to include packet level error correcting codes or flit level error checking. As well as this, these mechanisms will both require a more latent store and forward switch architecture. Our findings run contrary to existing solutions such as [153] which provide link-level retransmission, but given a topology with a short diameter we argue that this level of error checking is unnecessary.

5.1 Reliability Requirements for Reconfigurable HPC

Hardware Offloading

In order to properly utilize distributed FPGA resources for acceleration, the memory copies traditionally required between accelerator and host CPU in order to issue reliable communications to the network must be eliminated. In order to achieve this we propose a fully hardware-offloaded transport layer, bypassing the CPU completely from FPGA issued network transfers. This has long been known to solve the performance issues associated with software based transports, but in our instance it is also necessary to disaggregate the FPGA from the CPU.

Connectionless Transport

Maintaining scalability with regards to the memory footprint on the FPGA is very important. Given that the entire networking stack is implemented within the fabric of the FPGA, in order to decouple the CPU from the accelerator the transport layer must not hold connection states within the NI. Doing so would present a significant challenge to scalability. One option is to hold connection states in off-chip DRAM, although this presents problems of its own [118]; reducing the available memory bandwidth for useful work and increasing the latency of transmission. The second option is to maintain a connectionless (i.e. datagram based) approach, storing only the information required to reinitialise current outstanding transactions within the network in the event of failed delivery or packet corruption.

Our solution is able to offer reliability with no connection state owing to the fact

that network packets are simply an extension of the system bus protocol. We write directly into memory, and RDMA is set up directly from user-space with little overhead. The NI stores information regarding outstanding RDMA transfers only, as opposed to open source-destination connections. Therefore the memory required at the sender is bound only by the number of possible outstanding RDMA operations concurrently in flight, rather than the possible number of source-destination pairs.

5.2 Implementation of Transport Mechanism

Given that our NI provides hardware primitives to support shared-memory, NUMA-like operations, as well as RDMA operations for larger data transfer, we provide two separate methods for reliability in our transport. The need for low latency and the smaller burden that shared-memory operations place upon the system in terms of memory use mean that we can directly support data storage and retransmission in the NI. When shared-memory operations are issued the data to reconstruct the full operation is held in the NI until the transaction has been acknowledged.

For larger RDMA transfers we cannot afford to store this data in the NI. The required retransmission buffers would be too large to provision a sufficient number of packets given the scale of the network we target. This means the data consistency must be handled by the sender (middleware or user). Data must remain unaltered in the sender's memory until the whole RDMA operation has been acknowledged. The upside of this is that we do not require additional buffering unless absolutely necessary. The sender initiates an RDMA transfer and the data may not be overwritten until notification of the completion of the RDMA transfer is seen. In instances where the user wishes to overwrite and work on this data in parallel with communication then they must copy the data and provide double buffering. In this way the NI remains lightweight with only very small retransmission buffers, but also the data can be worked on *if* it is required. The downside of this is that at some level software must be given charge of maintaining consistency of the RDMA data. This duty can be given to the user, in which case performance would be maximised

at the cost of programming complexity. However, a simple user-level library could also be created which would wrap and obfuscate calls to the NI, checking completion status and performing additional allocation and data-copying if required.

In this section we present the microarchitecture of the two transport mechanisms developed for use in our system. One created for shared-memory operations, providing storage within the Network Interface for retransmission of small (in frequency and packet size), latency sensitive packets. A second is created to handle RDMA operations, which will form bulk data transfers, thus being unsuited for in-NI storage for retransmission.

5.2.1 Overview

Figure 5.1 shows the full transport layer within the Network Interface. Clearly marked at the top are the memory-mapped AXI interfaces which provide access to the shared-memory datapath, RDMA datapath and RDMA work buffers. At the bottom are marked the stream wrapped interfaces which carry our custom network packets. As can be seen the datapath for shared-memory operations and RDMA operations are completely segregated within the memory-mapped side of the NI. Separate modules provide the transport layer reliability for each of the two communication types on the send side, with all data destined for the network being passed through one of these two mechanisms.

The mechanism to provide reliable transfer relies solely on modules within the sender. All that is required of the receiver is to track the inbound requests in order to rebuild response packets appropriately. Although in order to enhance the performance of the system we introduce a receive side module to track outstanding RDMA operations. This is described in Section 6.2. As packets are sent they are assigned a special "Type" within the header which enables the receiver to identify how to handle the received packet, this is discussed in Section 3.3.2. Table 5.1 shows a breakdown of the individual modules and sub-modules which comprise the transport layer of the Network Interface.

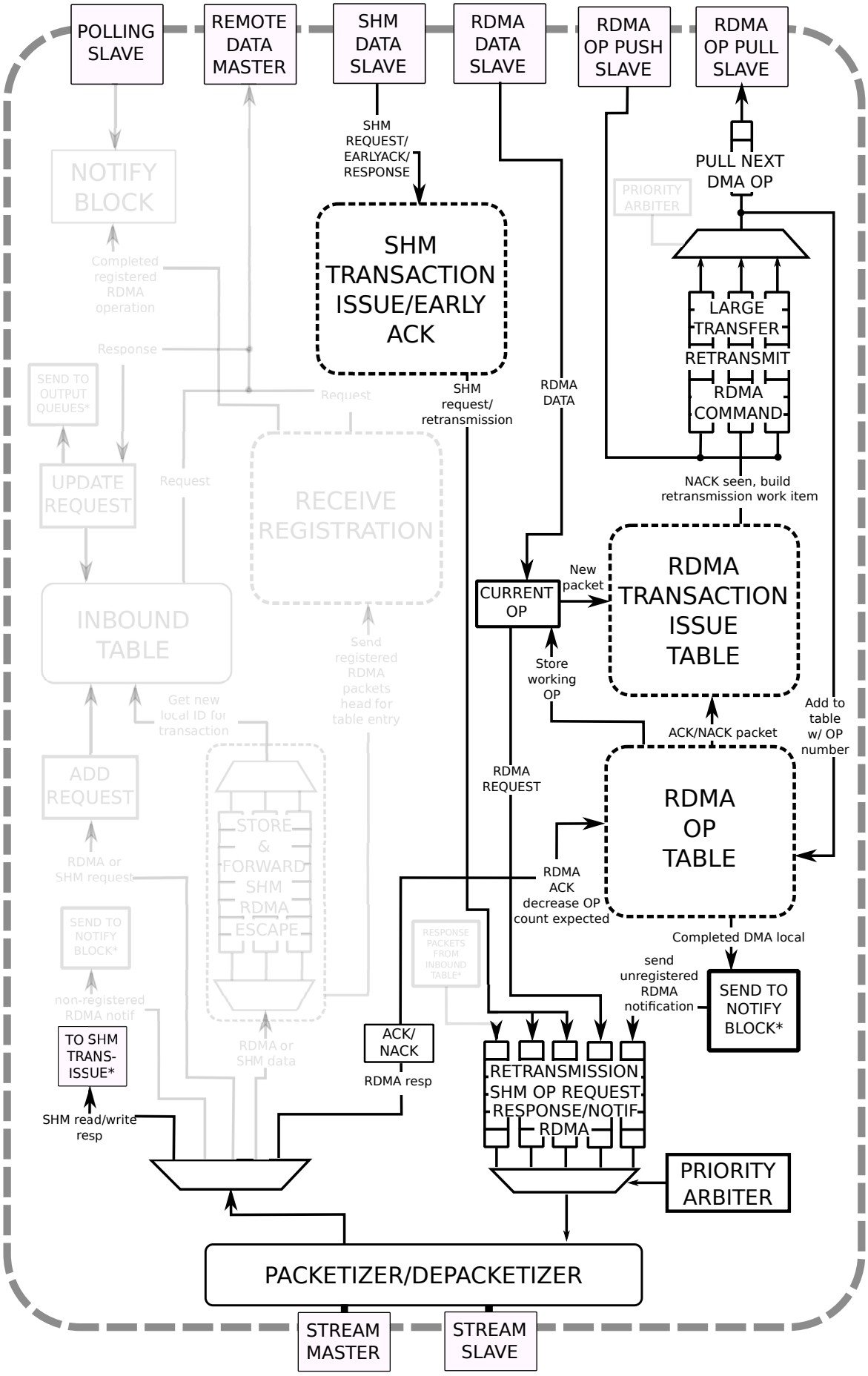


Figure 5.1: Highlighted here is the send-side transport layer within the NI, showing the separated methods for shared-memory and RDMA transfers.

Table 5.1: Transport layer modules (and sub-modules) within the Network Interface and their function.

Module (Sub-module)	**Function**
SHM Transaction Issue/Early Ack	This module provides the retransmission mechanism for shared-memory operations. Read and Write transactions passing through this module are wholly stored, and directly rebuilt in the event of a negative acknowledgement.
Transaction CAM	The addressing information is stored in a CAM in order to rebuild transmissions in the event of failure. The associated data is also stored.
Transaction Timers	A bank of timers is stored here and associated with the CAM table entries for timed-out retransmission.
Early ACK Module	The notification portion of the module sends early acknowledgements back to the sending process to enhance performance, this is discussed in Section 6.1.
RDMA OP Table	New entries are added as the DMA engine pulls new work items from the queues. The OP Table updates the new entry in the *Current OP* structure, used to build individual network packets. The table can currently hold 16 outstanding operations in flight at any one time.
RDMA Transaction Issue Table	Every time a new DMA packet is sent, it is recorded in the transaction table with the operation number and its virtual offset number. Acknowledgements (positive or negative) are sent back here to clear the table entries. In the event of a NACK the metadata stored here is used to build a partial operation for retransmission.
Transaction Timers	A bank of timers is stored here and associated with the RDMA transaction table entries for timed-out retransmission.

5.2.2 Shared Memory Retransmission IP

When a shared-memory operation is presented to the NI via the *SHM Data Slave* port it is sent to the *SHM Transaction* module. This module is composed of a table which is used to store all the information required to rebuild the transaction if a retransmission is required. The data (for write operations) is stored along with the transaction ID, destination address, burst length etc.

A list of available table entries is kept and an empty entry is popped from the list when a new transaction is presented. A timer which is linked to this entry is then loaded with a timeout value for retransmission and initialised. Currently the timeout value for retransmission is fixed. An appropriate baseline value must be determined. This value will be dependant on the latency of transmission which is affected by many things; the topology, system size, workload characteristics, other network load etc. Once a table entry has been found an ID equating to the table location is issued to the request. This enables the response packet to identify the corresponding table entry and to clear the associated retransmission timer.

A CAM (Content Addressable Memory) is used to store the outstanding transaction's destination address, as the full list of outstanding transactions must be checked every time a new transaction is presented. This is done to ensure memory consistency. In the event that a transaction is issued to the same address as an existing in-flight transaction it must be stalled in order to avoid the possibility of write-after-write or read-after-write data hazards. In normal programming practice write-after-write hazards should not really occur, however they must be protected against. The inclusion of the *early acknowledgement* mechanism (described in Section 6.1) to increase the performance of remote write instructions can increase the probability of this occurring, although repeated NUMA writes to the same address in quick succession would indicate poor programming practice.

Figure 5.2 shows the mechanism for keeping consistency among the shared-memory operations. At the arrival of a new transaction the table entries are checked for their destination address. Any used entry (in flight transaction) which matches the address will be placed in a list for pending transactions so as not to block the pipeline in the NI. Once a response is seen and the entry has been cleared from the table the list is checked. If any transactions match the destination address then they

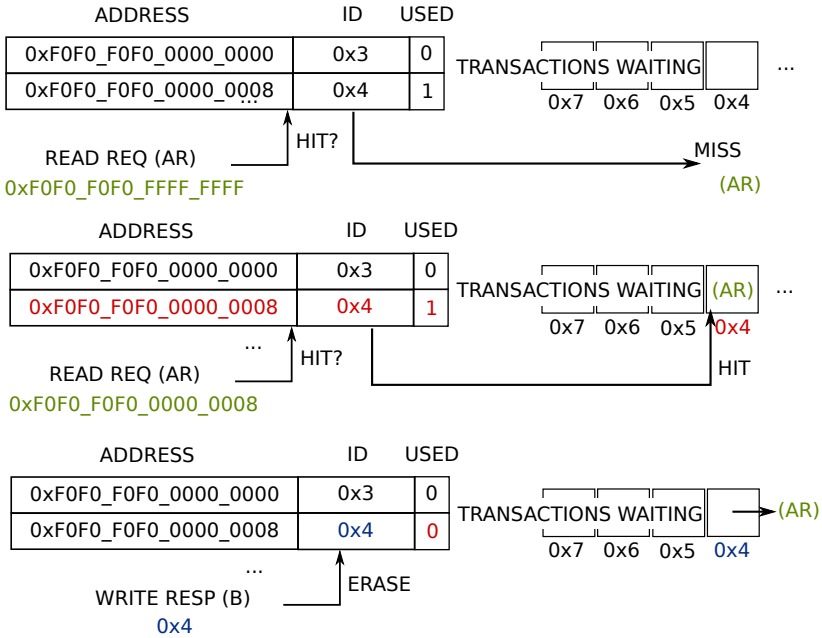


Figure 5.2: Mechanism to ensure consistency in shared-memory operations. Reads/writes to the same address will stall in the NI.

are processed again as normal.

5.2.2.1 Current Limitations

A suitable value for the timeout for retransmissions cannot really be found until a larger-scale prototype system is available to use, so for now it is set to an arbitrary value of 64μs. This value was chosen as it is simply 10000 cycles at 6.4ns within the logic fabric, and other works [6] provide similar timeout values of 50μs for example. This length is easily sufficient so as not to cause premature timeouts within our testing. Although the timeout value is fixed, a simple enhancement to the mechanism may have a dramatic improvement on the efficiency of timeout usage. Since the system utilizes a geographic addressing scheme (see Section 3.1.3) we can determine the distance of any destination node using very simple arithmetic. The (average) hop distance can then be used to set a higher or lower timeout value accordingly. This distance could change depending on the routing mechanism used. If the routing scheme is minimal [196] then the hop distance will not change. However, if a non-minimal scheme is used [24] then under high network load or when a given flow heavily saturates a link then the path may be longer than normal, as varying paths/distances are possible by misrouting. This may make estimation more

difficult.

Currently a single table entry in the data table corresponds to a single 8-byte transaction only. If a burst transaction is presented to the shared-memory retransmission IP the transaction is split into multiple single *beat* transactions (burst length 0), and multiple entries in the table. This corresponds to a much larger packet overhead for shared-memory operations as a packet is generated for every one of these entries. The additional memory required to store larger transactions is marginal however, and can easily be accommodated. For example, if we plan to store a maximum of 64-bytes per transaction (cache line size for the A53 processor[1]) we could accommodate for a 72 entry table using only a single 36Kb BRAM element on the Zynq Ultrascale+ device (out of a total 912). Note that this is to store the data associated with the transaction only, not including transaction metadata.

It may seem to limit the scalability of the system to use a CAM for storing the in-flight addresses for lookup, as the area overheads for a large CAM are not feasible for FPGA implementation. Currently we only support 16 concurrent entries. This value was chosen because all four cores of the Processing System on the Zynq device can issue two concurrent transactions (dual-issue, in-order cores), and as the interface path is 128-bits wide when we downsize to our 64-bit internal datapath this can produce up to 16 simultaneous 8-byte transfers. However, it is not expected that a given node will produce a large number of concurrent point-to-point shared-memory operations. It is much more likely that these are to occur in multicast synchronization messages; in this instance sending replicated data to multiple endpoints. As we discuss in Section 8.2.5, providing dedicated hardware to support collective operations is something which would be highly beneficial to the performance of the system, but unfortunately is beyond the scope of the work within this thesis.

Despite this fact, there are many possibilities to extend the number of entries in order to support software implementations of these collective operations. We can do this without incurring heavy performance penalties or increasing the area complexity of the mechanism dramatically. By simply reducing associativity levels

[1] ARM Cortex-A53 MPCore Processor TRM- `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0500e/BABCFDAH.html`, accessed April 2019.

and checking portions of the table in batches we can avoid increasing the size of the N^2 scaling of the matching logic of the CAM, while increasing the number of possible entries in the table. While this will increase the latency of lookup if the table is heavily populated, doing so in order to dramatically increase the size of the table should be viewed as a very reasonable trade-off.

There is also the possibility to increase the efficiency of the CAM in general, as our implementation is generally unoptimized. Recent work has attempted to address the scalability of CAMs within reconfigurable logic using the on-chip RAMs to implement the matching logic for the CAMs [197]. They stack bits of the words in columns of separate RAMs rather than having the whole word within a single row, thus enabling parallel lookups.

5.2.3 RDMA Retransmission IP

As mentioned previously (Section 3.3.4.2), the RDMA engine is currently a Xilinx AXI CDMA [16]. This sits outside of the NI within the FPGA fabric, but all operations directed to the RDMA engine must pass through the NI (see *RDMA OP PUSH SLAVE* and *RDMA OP PULL SLAVE* ports in Figure 5.1). This is because information is gathered inside the NI which enables it to track the operations which are currently being performed. In the event of a retransmission being required the NI is therefore able to rebuild partial transfers from the information about the initial operation which was submitted.

We are unable to keep all the data from outstanding operations within the Network Interface because the on-chip memory requirements for retransmission buffers in the FPGA would be too large. It is possible to envisage the use of link-level retransmission buffers to enable the burden to be distributed over many switches in the network, thus reducing the buffer requirements at the send-side NI. However this would be insufficient; copies of the data must still be kept for end-to-end retransmission in case there is any major issue with the network, for example faulty links or endpoints. It is impossible to simply push the data into the network and *guarantee* delivery of the packets. There also has to be some sort of end-to-end retry and acknowledgement mechanism. The only mechanisms which allow this (such as UDP) can only ever be best-effort transport mechanisms. Another reason this

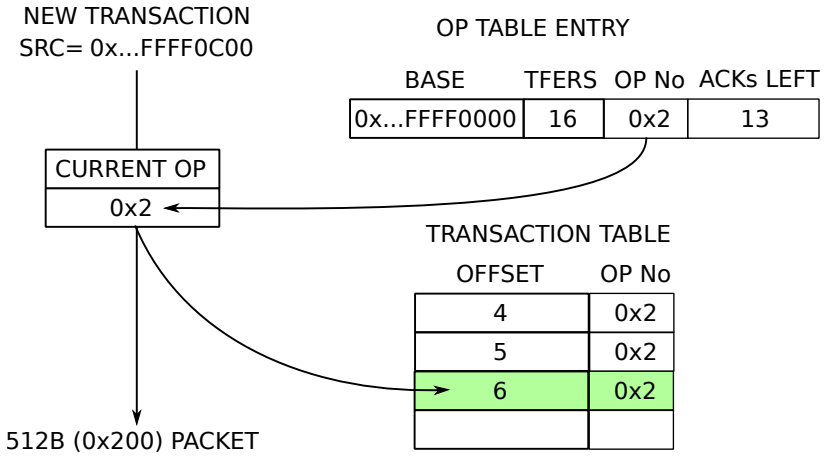


Figure 5.3: Shows how current DMA operation is marked for the individual transactions that comprise the full transfer, using offsets from the base virtual address, as opposed to storing direct addresses.

is not feasible is that ensuring consistency would become far more difficult if the network rather than the endpoints were in control of acknowledgements. A full, end-to-end acknowledgement would still be required.

The RDMA transport mechanism shown in Figure 5.3 is formed of a hierarchy of two tables. There is an *Operation* table which tracks the list of RDMA transfers/-operations which have been issued. There is also a *Transaction* table which monitors the individual in-flight packets which are associated with one or more full RDMA transfers.

To issue an RDMA transfer the CPU or accelerator logic writes a work descriptor into the *RDMA OP PUSH SLAVE* port of the NI (Figure 5.1). Table 5.2 shows the transfer descriptor, which must be 64-bit aligned. Source and Destination address are written, as well as a Configuration register which gives the transfer size and options relating to interrupts and transfer modes. These work descriptors are sent to queues within the NI which are pulled by the DMA engine every time a new transfer can be processed.

As the DMA engine pulls the transfer descriptor in to begin processing it is logged within the *DMA OP Table*. The DMA operation table entry consists of a base address, the number of expected transfers, an operation number, as well as a field to track errors. Every new transaction that enters is assigned an OP number for the current working operation. The transaction is then logged in the Transaction table. The individual transactions are given an offset number in the table entry, calculated

Table 5.2: Transfer descriptor for an RDMA operation.

Struct Entry	**Details**
`next_desc_ptr`	Address of the following work item, 64-bits.
`src_addr`	Base virtual address of the data to DMA, 64-bits.
`dest_addr`	Base virtual address of the target for the DMA operation, 64-bits.
`control`	Control register for work item 32-bits, 25:0 = bytes to transfer (max 64MB), 31:26 = reserved.
`status`	Status of transfer 32-bits, 31 = transfer complete status, 30:28 = decode error (incorrect address), slave error (slave could not accept data), internal error (internal DMA engine error), 27:0 reserved.

from the base address, as the DMA transfers within a single operation must be contiguous in virtual memory. The DMA OP number is sent with the packet in order to mark the responses as they return. The offset value can be placed in the transaction table along with the OP number to reduce the size of the table entry, rather than storing the source and destination address for every outstanding transaction.

Figure 5.4 shows how individual packets are sent through the transport layer and what happens on return of an acknowledgement packet. As the individual packets are entered into the Transaction Table an associated timer is started, with retransmission performed upon timeout or receipt of a negative acknowledgement. When a response packet arrives back it is sent to both the OP Table and the Transaction Table. If the response is positive then the OP table decrements a counter on the number of expected responses remaining, if not then the operation is marked as failed but remains in the table to handle other acknowledgement packets.

Figure 5.5 shows how a retransmission works in the case of a negative acknowledgement. The transaction table builds a transfer descriptor from the table element in the event of a failed transaction or timeout and places it into a retransmission queue for the DMA engine to process. When the DMA engine processes the failed transaction and retries, the address is checked against the OP table. The OP marked failed whose range contains this address is then refreshed. This is done in order to prevent the system from sending a notification packet to the receive side until every transaction has completed successfully, including retransmissions. Only once

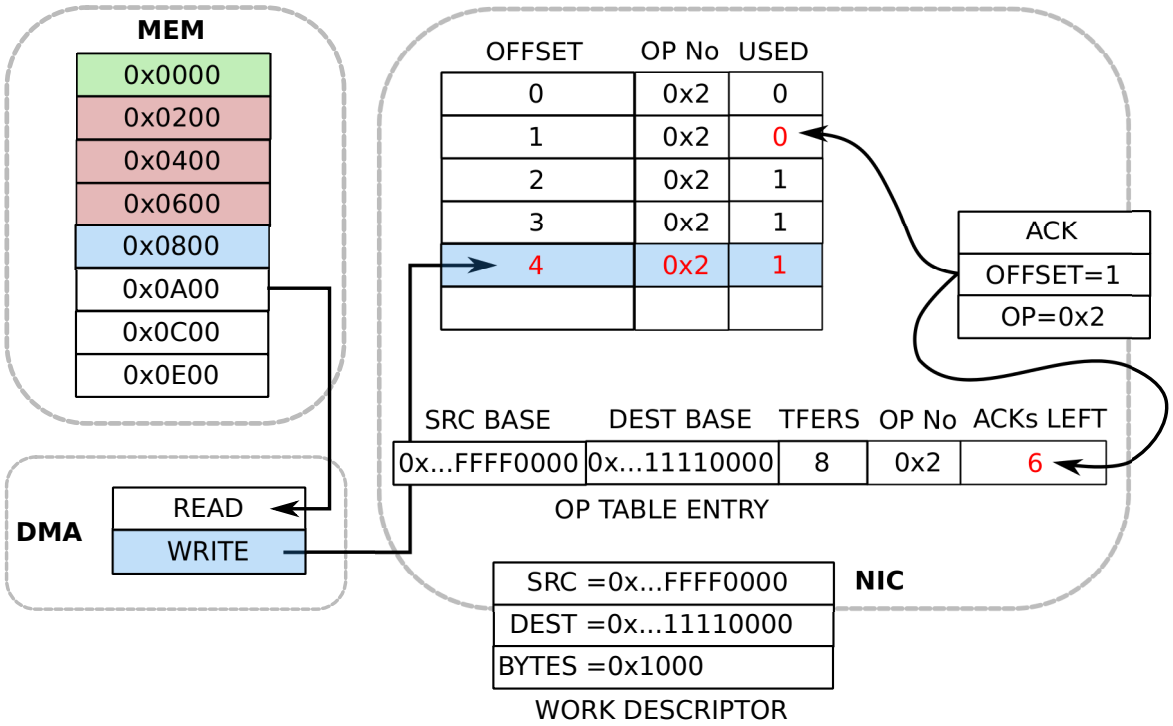


Figure 5.4: Return of an acknowledgement packet, and its effect on the OP Table and Transmission Table within the NI.

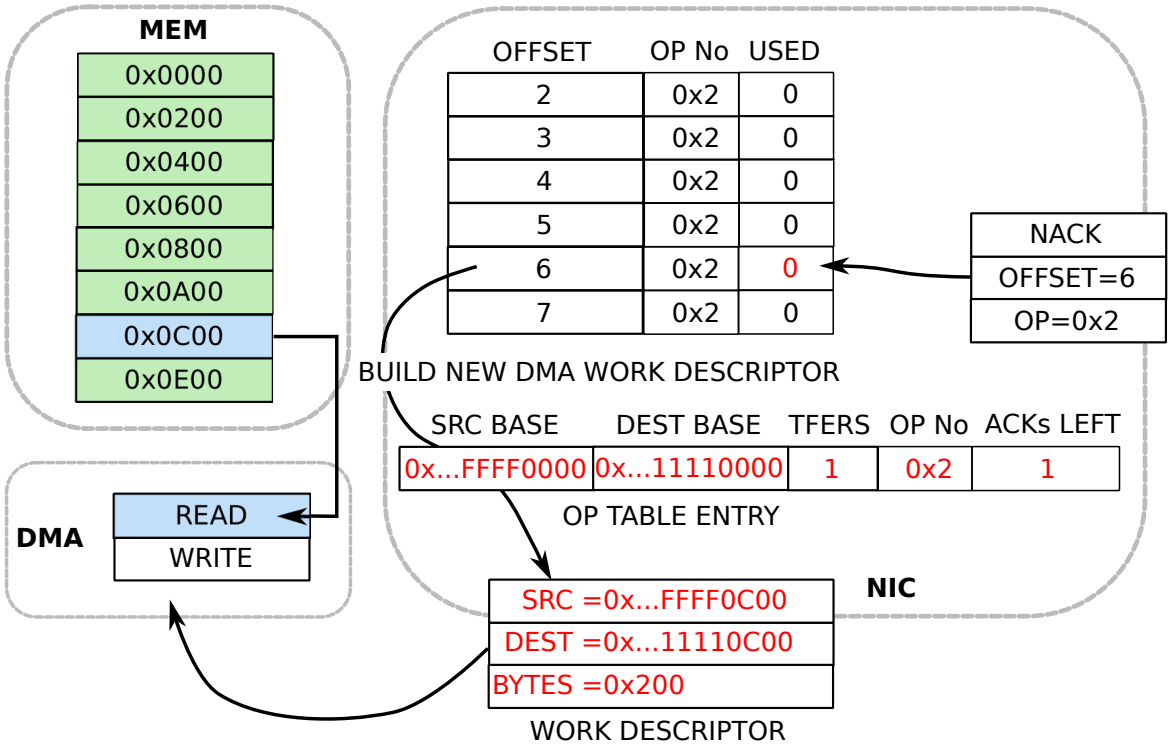


Figure 5.5: Rebuilding a new DMA transfer descriptor from a negative acknowledgement, using the original base from the OP table entry and the offset of the negatively acknowledged packet.

the count of expected transactions in the DMA OP Table entry is zero will a notification packet be constructed and sent to the receiver. On the receive side the notification packet is processed very simply; being added to queues which must be checked by the CPU or accelerator resources.

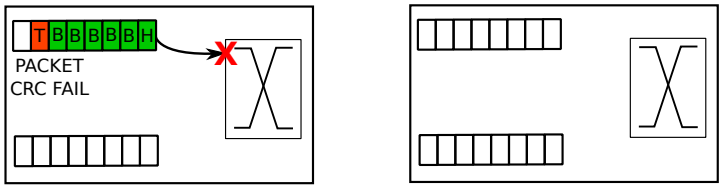
5.2.3.1 Current Limitations

The current implementation has static, dedicated queues for notifications and for RDMA operations. This means that we have two separate physical ports on the NI to handle accelerator based transactions and CPU based transactions. Having a single physical interface with the ability to handle multiple logical interfaces is necessary to allow more than one process to issue RDMA operations simultaneously. This can easily be done by associating logical queue identifiers to RDMA operations, with the CPU or accelerator passing additional information to the NI. Processes can set up these queue identifiers during initialization of the program.

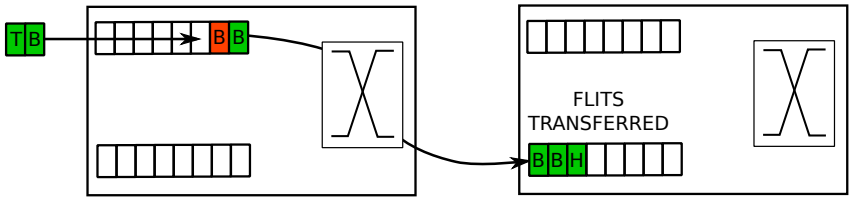
Providing these logical queues does not limit the scalability of the system in any way, as the notifications given to the user show the information about which operation has completed. Only one queue ID is needed per process, per node in this instance. This is as opposed to requiring a queue per source-destination pair, or a queue per source-subnet (as in more scalable implementations of the Infiniband RC transport [142]).

Given that all of the RDMA operations pass through the NI we can begin to provide priority scheduling for the RDMA transfers. The DMA engine operates by issuing read transactions to a single address repeatedly (the physical address of the work queue in the NI). We can therefore easily modify the transfer descriptor (which has redundant bits already) to include a priority level field, and send the descriptor to separate queues within the NI. As well as this, another feature which can simply be implemented is the ability to segment very large transfers into smaller ones. This prevents smaller transfers from being starved of resources and also helps enable increased overlap of computation and communication. This is discussed in further detail in Section 6.3.

Unfortunately these features are insufficient for providing complete, priority-based QoS within the full system, as the network itself must be made aware of



(a) Packet-level CRC checking, which requires a store and forward architecture in the switch. The whole packet is required prior to error checking.



(b) Virtual cut-through switching, where switch allocation occurs and flits are able to progress to the next switch before the entire packet has been buffered in the previous stage.

Figure 5.6: Store and forward and virtual cut-through switching.

the priority placed upon packets and act accordingly. All we are able to provide at the Network Interface is the priority level information, and to arbitrate between different priority RDMA transfers in the event that multiple operations are simultaneously issued.

5.3 Retransmission and Fault Tolerance Strategies

Many interconnection fabrics today provide link layer error checking with CRCs (Cyclic Redundancy Checking), or forward error correction (FEC). While techniques such as these can provide added reliability and reduce the latency or frequency of retransmissions, they come at a price. Including a packet level CRC at every switch increases the average latency of transmission for all packets. The switch necessarily incurs additional latency as its architecture must be store and forward in this case. Figure 5.6a shows the store and forward nature of a switch, where the whole packet is buffered before switch allocation. Using a Virtual-Cut-Through [198] technique would allow packets to be processed and begin leaving the switch before the CRC at the end of the packet can be checked. Including CRCs at the flit level is not feasible either, as a flit can be erroneous but prior flits may have been processed and left the switch already (see Figure 5.6b), meaning a technique for dropping these forwarded flits would be very complex indeed.

Using Forward Error Correction in the form of an Error-Correcting Code (ECC), would enable highly reliable transmission with low jitter even on erroneous packets, but also requires a store and forward switch architecture. Obviously this comes from the fact that errors which are introduced on the fabric are to be fixed before transmission. The price that is paid in this case is reduced bandwidth of the links. The packet overhead from adding an ECC varies dramatically depending on the type of code used and the number of bit-errors in a flit which can be identified and fixed. It is important to note here that the chances of bit-errors occurring raises temporally with respect to a previous bit-error being seen, as described by Gilbert in his Markov model for burst error probability [199]. Therefore multiple errors can become much more likely if a single error is seen. This means that forward-error-correction is not an absolute guarantee to prevent retransmissions. However, typical high speed serial links using traditional copper cabling operate at an expected maximum bit-error-rate (BER) of $\approx 10^{-12}$, which is very low.

In our opinion having such low error rates on the links means that the cost of ECC on the useful bandwidth available is not worth the price. Additional average latency on the links by adding CRC is also not worthwhile. In this section we evaluate two possible techniques for link level error checking and forward error correction against a purely end-to-end technique and find that even under very aggressive error conditions there seems to be little advantage in using link-level techniques. Partly this is due to the fact that latency penalties for end-to-end retransmission are not as high as the requirements for store and forward switching given the fact that we are able to take advantage of a low diameter topology such as Dragonfly [149] (see Figure 5.8), which is permitted by the system switch described in Section 3.1.3.

5.3.1 Latency and Fault Injection Mechanism

The work presented in this thesis cannot be tested easily beyond a small system. There is currently no distributed runtime system available for our hardware, making testing of real-world distributed applications impossible. There is also no port of programming interfaces such as MPI onto our simple API. Another issue is

the fact that we do not have access to the virtual-physical and physical-global mapping hardware which is required for a larger system to be created. The IOMMU configuration is complex and would only serve as half of the required translation scheme (because only physical addresses can be mapped to the CPU-FPGA interface, a translation stage inside the FPGA fabric is also required). Due to this fact we only have access to small windows of statically mapped physical regions of memory, suitable for testing the raw capabilities of our design. Testing the hardware in a real-world system with real world applications would require many person-months (possibly years) of effort, which are clearly beyond the scope of this thesis.

Due to this fact, we have created a fault injection IP which mimics the flow of data and response packets between nodes over multiple hops within a larger system. The IP (seen in Figure 5.7) adds an artificial latency to any incoming traffic to simulate large scale network traversal, and creates positive (ACK) or negative (NACK) acknowledgements based upon the error injection rate and the distance travelled.

The *Rand Addr* IP block shows how we manipulate the target address to represent a weighted, randomly assigned distance to which the packets must travel. We desire (and are able given the proposed switch design for the system, see Section 3.1.3) to use modern low diameter topologies, such as Dragonfly or Jellyfish. In our case we assume a Dragonfly and so we model path lengths between 2 and 5 hops; distance 1 is not possible here as we assume an indirect network topology. For example, see how Figure 5.8 illustrates the number of hops it takes to traverse a Dragonfly network between any two given points. The address of the transaction is changed to simply reflect the number of hops we wish the packet to travel. This value is then used to multiplex the packet into the correct path through the fault-injection circuit.

We test with two different traffic distribution models; a *local* distribution **(-L)** that gives higher probability to communications that are closer with ratios of 8:4:2:1 for distances 2 to 5 hops, and a *remote* distribution **(-R)** which uses ratios of 1:2:4:8. In the *Injector* block we see how the manipulated address is used to send the packet information through an array of delays and comparators to emulate link latency and error injection. The width of this array is attributed to the distance the packets

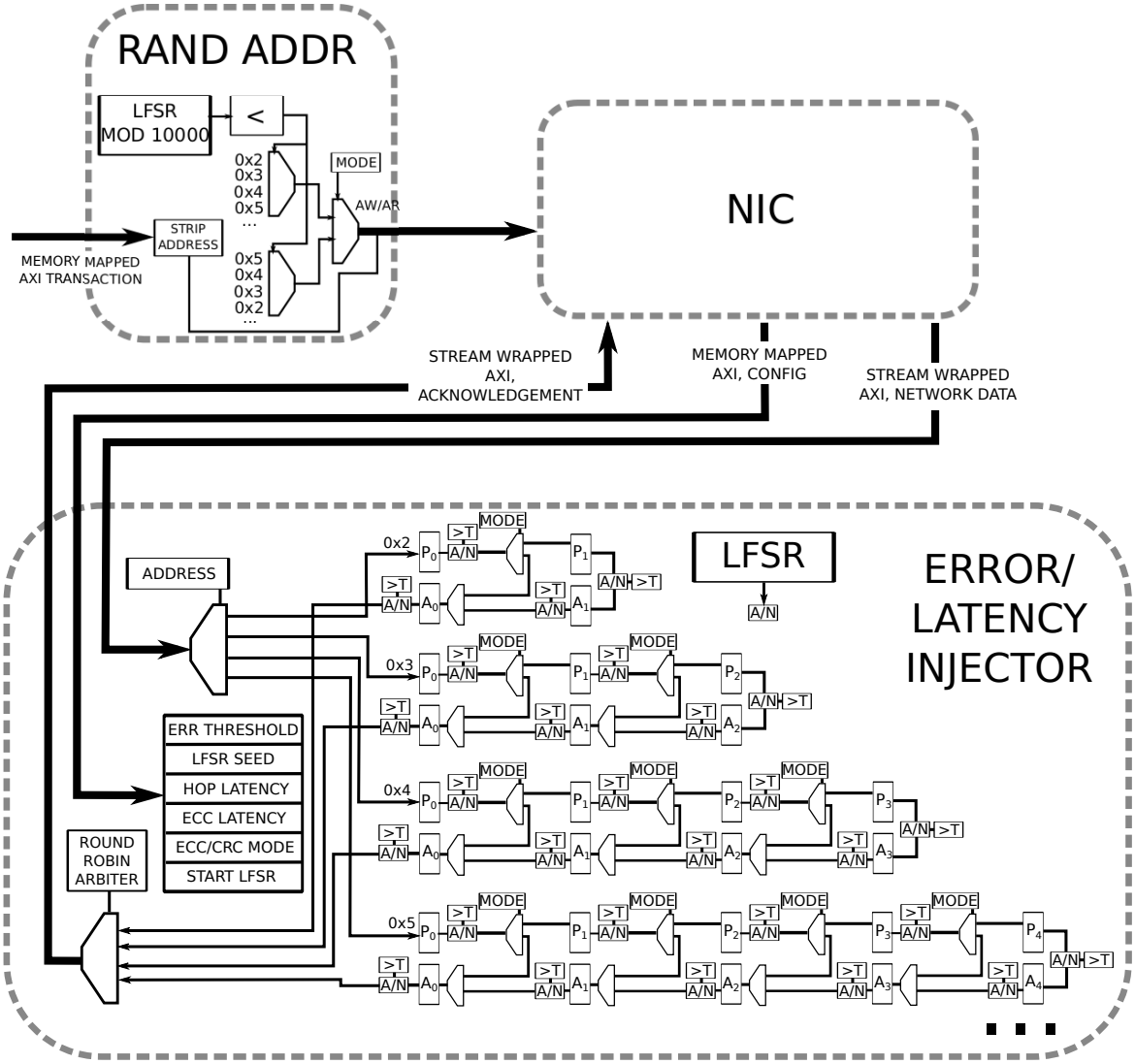


Figure 5.7: Microarchitecture of the fault injection hardware used to test the transport layer.

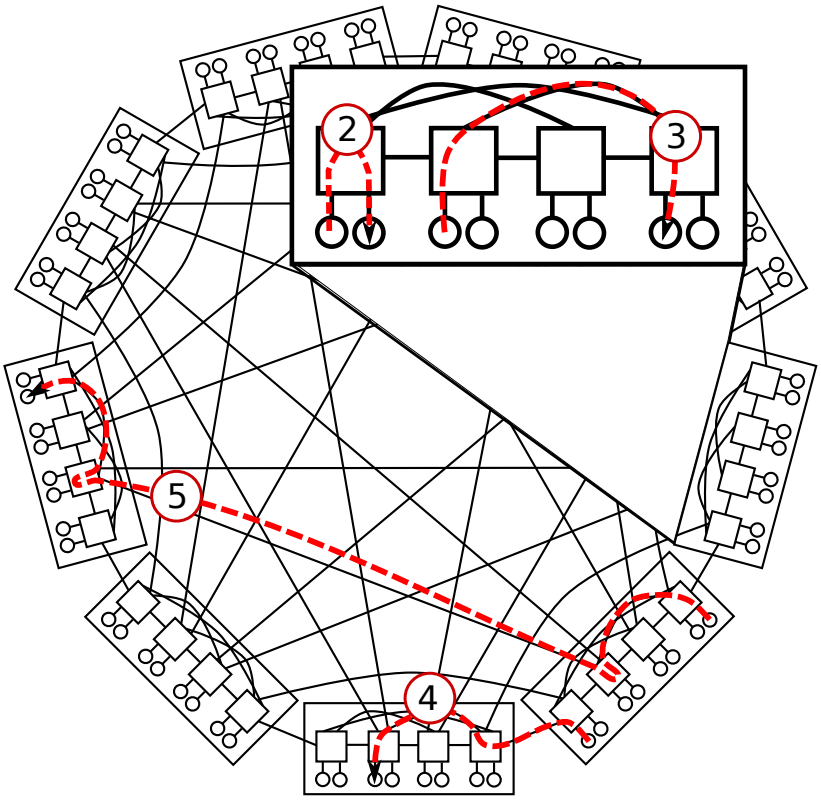


Figure 5.8: The number of hops to cross a fully connected Dragonfly network.

must "travel". Errors can be introduced and NACKs generated at any number of these stages (hops).

The relevant data from the packet header/footer required to form the response packets are held as a structure until the entire packet has entered the block. The redundant data is discarded. Upon the arrival of the last flit of the packet, the held data is placed into the network at P_0, with a timestamp T which denotes the time it is able to progress. The timestamp is calculated given the current time T_0, the per-hop latency H_L, and the ECC latency (if there is any) E_L as $T = T_0 + H_L + E_L$ in cycles. At every cycle the timestamp is checked against the current time t, and is allowed to progress once it has been reached.

The P_x (packet) and A_x (acknowledgement) structures used to hold the response packet structure represent short buffers. This is required to account for the fact that collisions may occur in the delay network, as can be seen in the fact that arbitration between the source of the A_x structures is required between A_{x+1} and P_x. Once a structure is allowed to progress one hop its timestamp is recalculated, but also it goes through a process of fault injection.

The A/N fault injection blocks take the error threshold from the configuration register, and a random number generated by a 32-bit LFSR (Linear Feedback Shift Register) to determine whether a fault should be injected. If the result is positive the response entry will be adjusted to provide a negative acknowledgement for that given transaction. If the mode of operation dictates that link level error checking is in place, then the resulting packet will be inserted directly into A_x, where x is the number of hops the packet travelled before having the error injected.

5.3.2 Measuring Latency and Jitter

The fault injection IP is used to test three possible scenarios for error detection and correction within our prototype in order to determine which would be most suitable for a full-scale system. We use measurements and parameters from other experiments (Section 6.1) to model the properties of the network such as hop latency and CRC store and forward latency. By doing this we hope to gain a better understanding of which method may be most suitable when incorporated into a larger system. The three scenarios are:

- End-to-end CRC checking **(E2E)**. In this instance there is no link layer error checking performed, allowing for minimized latency for normal transfers, but requiring far higher latency for retransmission.
- Link-level CRC checks **(LLV)**. In this instance the switch performs error checking and drops the packet if an error is observed. This solution offers a higher average latency for regular transfers, but reduces the latency of retransmission and does not waste network bandwidth propagating erroneous packets.
- Link-level ECC **(ECC)**. In this instance there is an additional bandwidth overhead which is modelled by simply sending more packets in order to reach the required amount of data to be sent, reducing the goodput of the network. This solution offers a higher average latency for normal packets but very low jitter, as unexpected errors are mitigated against.

5.3.2.1 Basic Parameters

In the end-to-end CRC check mode (E2E), the packet must travel the full round trip through the network and cannot use the faster path out of the array of fault injection blocks (see the shorter path through the multiplexers and P_x/A_x buffers in Figure 5.7). Link level CRC check and ECC mode add a fixed latency E_L to each hop. This corresponds to the requirement for a store and forward architecture in any switch using this error checking method. The delay equates to the number of flits in the packet, as the whole packet must be present and verified correct before progressing to the next delay block (switch). However, since ECC is intended to protect the packet from corruption we assume a fault can never be injected in this mode. This assumption is not entirely realistic as ECC can only *recover* from a small number of bit errors (although the probability of *undetected errors* over a copper link in common ECC implementations is effectively nil). If burst errors are seen on the link and there are too many errors to correct then retransmission would be required. Note that this simplification favours the ECC model in terms of performance, packets simply take on the latency of the full round trip for the data and acknowledgement, ignoring the fault injection blocks. As we will see, even with this favourable conditions for ECC, and end-to-end mechanism will still outperform it in terms of latency and throughput.

For the link-level CRC check (LLV) instead of waiting for a timeout for retransmission (introducing additional latency) we choose a more performant solution. We generate negative acknowledgements in the switch where the error is introduced and send them back to the NI. In doing this we minimize the delay for retransmission, and the fact that we do not have a suitable concrete value for retransmission timeouts becomes irrelevant.

From other experiments (Section 6.1) we see that the round-trip time for a single RDMA packet is around 1.9μs. For simplicity we therefore use a baseline latency of $\approx 1\mu s$ per hop (156 cycles). In link level CRC/ECC mode an additional 64 cycles (number of payload flits) latency is added at the end of each hop to account for the fact that store and forward switching would be required in this instance, as opposed to virtual cut through.

5.3.2.2 ECC Bandwidth Overhead

There is a large corpus of research on Error Correcting Codes (ECC), which are generally categorized into three classes [200]; classical ECC, concatenated codes, and LDPC (Low Density Parity Check) or turbo codes. The most suitable, LDPC codes, are routinely implemented within FPGAs [193] and are used in many standards such as WiMAX [201] and 10GBase-T Ethernet [202]. However, for the purpose of the following analysis we will use values for the overhead of ECC on bandwidth of 12.5% from [203]. This value was chosen as it is the most optimistic value for ECC, being easily among the lowest overhead we found in terms of coding efficiency [193]. The performance of the ECC mode will therefore likely be lower than this in any real world implementation on the FPGA. The ECC overhead is accounted for in the delay network by transferring additional packets to make up for the bandwidth overhead.

5.3.2.3 Link-Level Retransmission

There are two methods for performing link-level retransmissions; with CRC per-flit, and a CRC per-packet. We choose not to implement either of these retransmission methods in our experiments. The most important reason for this is that it is very difficult to estimate the latency for retransmission given that it will be heavily implementation dependent. First a negative ACK must be sent back upstream, and then the retransmission buffer must be allocated resources within the switch before resending the packet/flit. There are too many unknowns to put a sensible estimate on these values without some form of implementation.

The second reason that we have chosen not to implement *flit-level* CRC retransmissions is that using the ECC mechanism gives an acceptable indication of how this flit-level retransmission may perform. A flit-level CRC and retransmission would incur additional bandwidth overhead like ECC, and for our ECC model there is no additional retransmission latency (as we recover from the error). As such we can assume that the ECC would perform better in the event of retransmissions, and similar under conditions of correct transmission. This is true as long

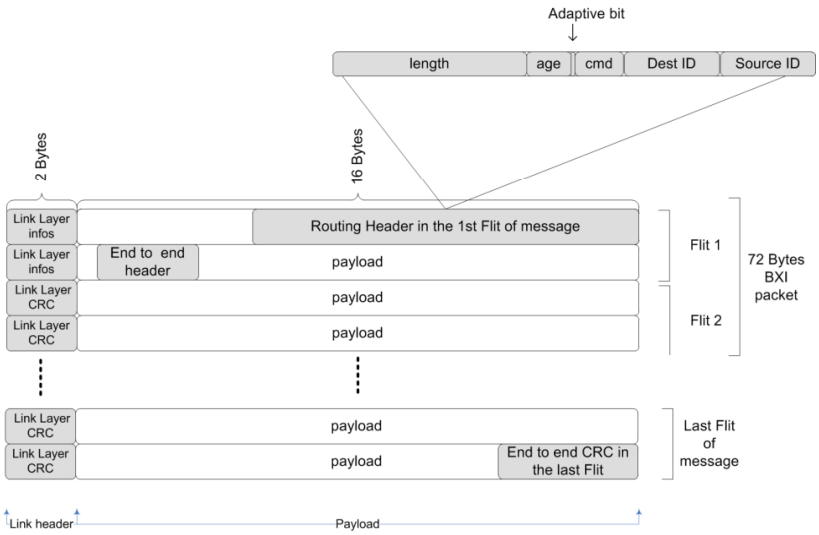


Figure 5.9: Message format for BXI. Taken from [153].

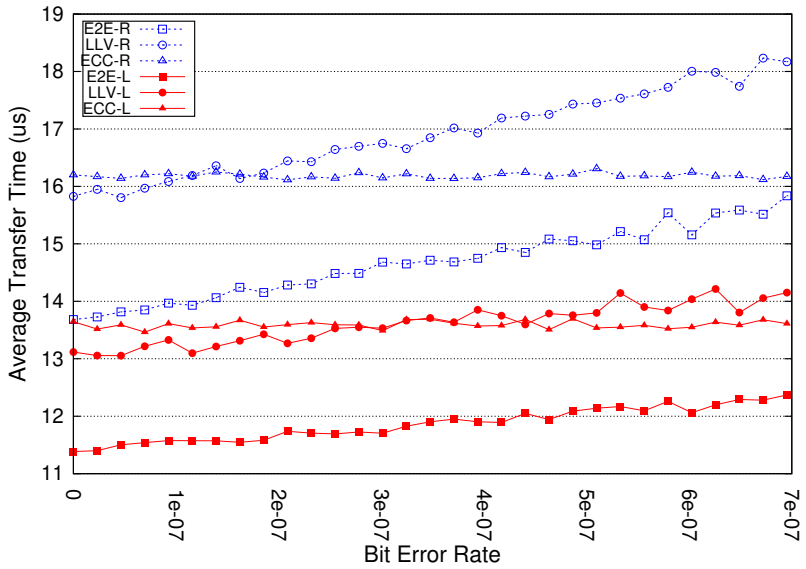
as the bandwidth overheads are the same. We can see in examples of other state-of-the-art interconnect technology using link-level CRC-checks that this is the case. In the BXI interconnect [153] they perform link-level retransmissions and require an additional overhead of up to 12.5% (depending on the amount of payload per packet). For packets with negligible overhead from header/footer we see that the CRC overhead is therefore very similar to the bandwidth overhead we use in our ECC calculations. Figure 5.9 shows the message format for the BXI interconnect. It is therefore correct to assume that our ECC model would perform the same or better than a flit-level retransmission mechanism.

5.3.3 Results and Discussion

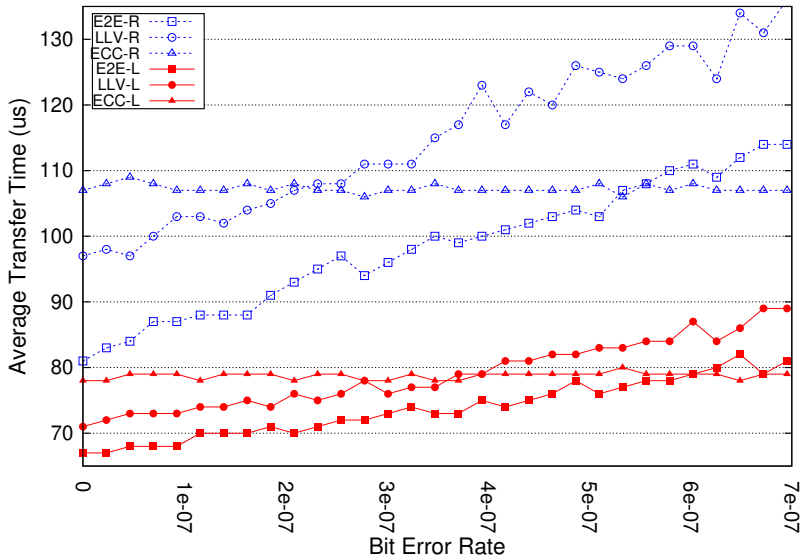
Latency

Using the *local* (L) and *remote* (R) traffic distributions discussed above we find the following results for our three fault tolerance techniques, shown in Figure 5.10. These graphs show the average latency for data transmission and acknowledgement for different transfer sizes as we vary the Bit-Error-Rate of the fault injection IP. Each graph shows the time for the end-to-end (E2E), link-level CRC (LLV) and link-level ECC (ECC) mechanisms.

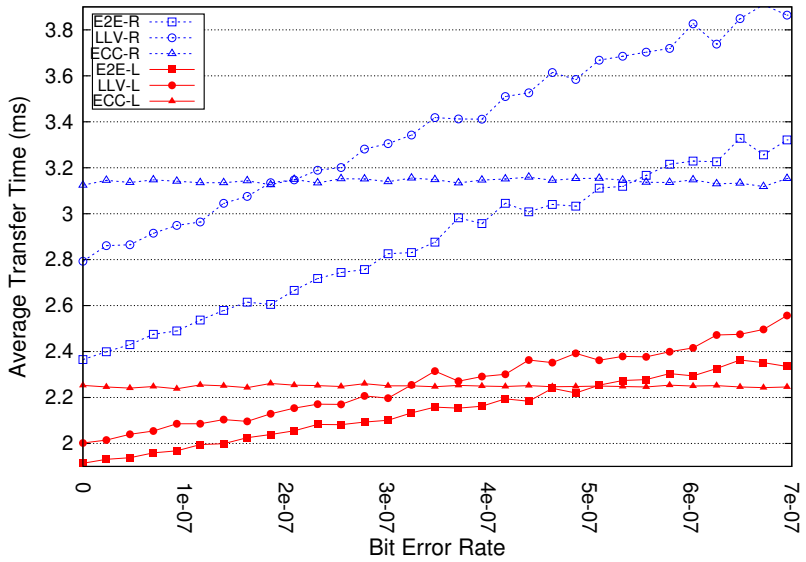
Notice that the BER of the links used for testing starts from zero (completely error free link), but then reaches very aggressive levels in the order of 10^{-7}. This



(a) Average latency for 4KB transfer..



(b) Average latency for 64KB transfer.



(c) Average latency for 2MB transfer.

Figure 5.10: Average latency of transfers through the fault injection IP.

is far beyond the expected error rates of 10^{-12} required for 10G SFP+ cables. However, we use these high error rates in order to find the error rate at which ECC becomes a viable alternative to end-to-end error checking. It can be seen that even for relatively small message sizes ECC cannot outperform end-to-end checking until a bit-error-rate of $\approx 5 \times 10^{-7}$ is reached. If we were to run these tests around a standard BER of 10^{-12}, the incidence of bit errors would be so low that the latency would appear to be near constant.

The results suggest that there is no real value in performing link-level CRC checks. The only rationale behind the decision to use this feature would be if the switch architecture required implementation using a store and forward technique regardless of the error checking. The additional average latency on normal packets far outweighs any benefits from retransmission. There is a reason that the higher average latency can never be amortized by the reduced round trip time for erroneous packets even with extreme link error rates. This is caused by the implementation of the retransmission scheme. The NACK'd packets are not handled until the whole of the initial DMA transfer has been issued, meaning that even if the NACK'd packets arrive back at the source earlier they will not be serviced straight away unless they are at the end of the full DMA transfer.

For smaller transfers the implications of the ECC bandwidth overheads are less pronounced because link saturation is nowhere near achieved for these smaller message sizes (see Figure 5.11). Even under these conditions with small transfers we see that the link-level error correction does not begin to outperform the end-to-end mechanism until a very high bit-error-rate of $\approx 3 \times 10^{-7}$ for the local traffic distribution. For the remote traffic distribution the ECC performs even worse comparatively.

For larger message sizes of 64K and 2M (where we begin to see link saturation) the results show similar, higher crossover points for performance of end-to-end and link-level techniques at $\approx 5 \times 10^{-7}$. This is for both the local and the remote traffic distributions. Given the difference between these results and those at much smaller message sizes and the similarity between local and remote distributions for larger sizes, this suggests that the performance of these techniques is more affected by the message size than the distance the messages travel. This is perhaps due to the

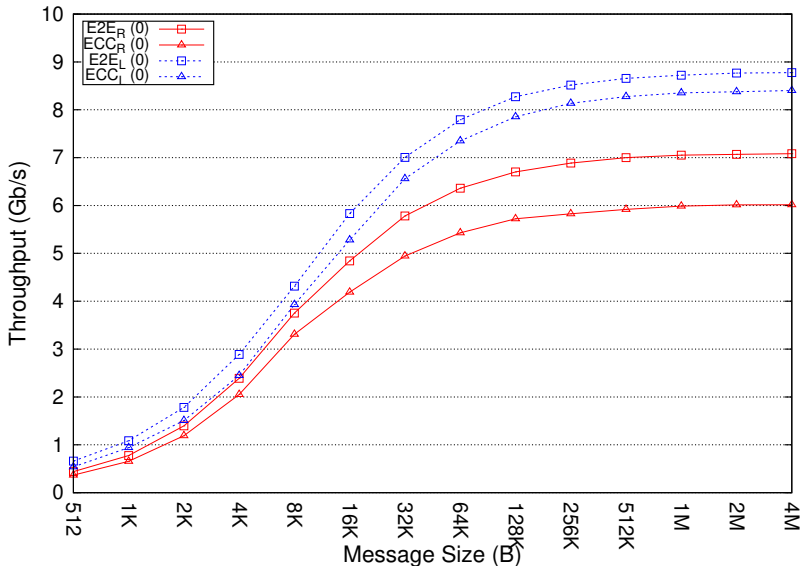


Figure 5.11: Throughput for RDMA transfers of differing message sizes.

low diameter network which we target, so the latency for retransmission is not a huge factor when compared to the higher average latency when using a store and forward mechanism.

Throughput

The achievable throughput and saturation point for ECC and E2E is shown for different message sizes in Figure 5.11. This shows us two things. Firstly is the extent of the negative effect of the ECC overhead on the saturation point for the payload throughput. Secondly is the fact that the remote distributions lower the achievable throughput. For the local traffic distribution we see link saturation at the levels achieved in Section 3.4.3, but this is not the case when extended out. This is due to limitations within the system for issuing concurrent RDMA transactions. The RDMA transaction table currently only permits 16 entries (in-flight transactions) as this is the maximum that the Xilinx CDMA IP [16] can allow for. Our work on allowing early acknowledgements for AXI write transactions within the NI overcomes this issue, and will permit the issuing of a larger number of concurrent transactions. This is discussed in Section 6.1.

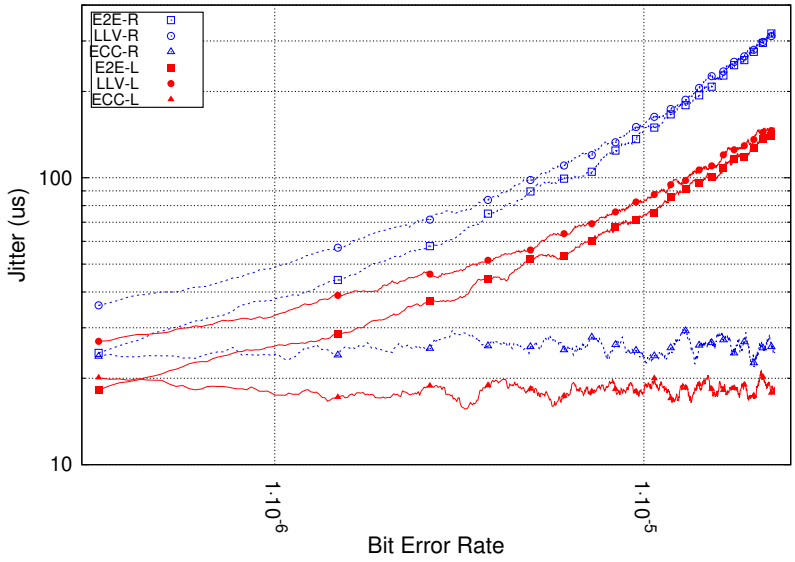


Figure 5.12: Jitter for 4KB RDMA transfers under various error conditions.

Jitter

In Figure 5.12 we show the jitter seen for small 4K message transfers within varying bit-error rate conditions. We only care about the jitter on these smaller message sizes, as small messages are far more likely to comprise latency sensitive data. The graph shows that the ECC massively outperforms the end-to-end retransmission technique for all but the lowest recorded error rate $\approx 3.3 \times 10^{-7}$. At these low levels the jitter rates are very similar for E2E and ECC. This would suggest that the jitter at these levels is merely due to the distance travelled by the packets rather than retransmissions. However, notice that this cannot be the case as the LLV solution has a much higher jitter even at the lower error rates. Given that the ECC and E2E results are so similar at lower error levels, this suggests that in all but the most highly jitter sensitive applications the E2E solution would be beneficial to use due to the large bandwidth overheads.

Discussion

These results seem to contradict the current practise in the field, which is tending towards greater levels of error correction as link speeds become higher. However, given the context of our implementation (short diameter network, area constraints in the FPGA, 10G links etc.) and the ramifications of added switch complexity and requiring store and forward latency, we argue that these results hold true. We show

that under anything approaching the expected low error rates of the links then the additional overheads are not worthwhile. It remains to be seen whether the higher jitter could cause problems within real-world applications (preventing timely synchronizations), or whether the low expected error rate on the links mean that this would not affect overall application runtime too heavily. In order to evaluate this we would require a full prototype system, able to present realistic workloads with real dependencies. Unfortunately getting to this point would require many person-years more work, and so is obviously well beyond the scope of this thesis.

Network Load

These results do not take into account any possible network load within the system, which obviously effects the latency of transfers through the network due to packet buffering etc. In an instance where high network load causes significant delays to packets reaching the allocation stage of the next router it is possible that the entire packet may already be buffered within the switch, masking the additional latency of performing link-level CRC checks. In this case the link-level CRC or ECC techniques may outperform a simple end-to-end solution. However, given the fact that such high BERs are required to make the solution worthwhile, it seems unnecessary to provide such a solution given the additional FPGA area overhead incurred for these features. This is particularly true given the low diameter of our target network, and the small effect that the chosen mechanism has on the jitter at low error rates (which are still quite high error rates in reality, given the 10^{-12} rate on standard link technology).

5.4 Concluding Remarks

In this Chapter we detailed the design of a fully hardware-offloaded transport layer which is implemented within the fabric of the FPGA and enables reliable data transfer without the need for CPU intervention. Our design maintains a connectionless approach, and we show that we can maintain reliability without maintaining information regarding connection state. The only state information required is with regards to current outstanding transfers in the network, making it inherently more

scalable than connection based approaches. Our novel solution uses two separate transport mechanisms depending on the communication types. This means that our solution has a low memory footprint within the FPGA.

Our analysis at the end of the Chapter attempts to determine whether link-level error checking or correction techniques should be employed within the system. Given the fact that our system is targeted towards a low diameter Dragonfly topology and that we utilize the 10G links of the FPGA, our analysis shows that an end-to-end reliability scheme is easily sufficient given the bit-error rate on modern high speed links.

The work in this Chapter provides contributions in the following ways:

1. A novel hardware implementation for a transport layer designed for FPGA based communications which has *both* the following properties; it is reliable and connectionless. Previous techniques are either best-effort, or require connection state to be held, thereby requiring dedicated per-flow send and receive buffers. The transport layer uses two separate retransmission strategies to provide less latent retransmissions for NUMA-like remote memory accesses, as opposed to RDMA messages.

2. An analysis based upon our specific system properties which suggests that end-to-end reliability is sufficient for our needs, and that modern ECC techniques are unnecessary for such short diameter networks as ours and for the expected error rates of the transceivers.

Chapter 6

Performance Enhancements

The basic structure of the Network Interface and our custom, hardware-offloaded, reliable transport layer has been described in Chapters 3 and 5. In this Chapter we discuss several enhancements to the architecture and demonstrate their effectiveness. Firstly we discuss the method for providing *early acknowledgement* of write transactions within the NI, and show the performance benefits which arise from this. We discussed in detail the implications of using this technique on the consistency model in Chapter 4.

In this Chapter we also describe a technique to improve the performance of RDMA operations at the receiver by registering certain operations and tracking their progress. In doing this we are able to provide notifications of RDMA operation completion directly to the receiver from its own Network Interface. The standard method of acknowledgement from the sender is bypassed in this instance, saving a round-trip latency for the full communication.

Closely associated with the analysis of the experiment performed is a further optimization that can help enhance the productivity of computing on our system by allowing overlapping communication and computation to a much higher degree. We discuss the possibility for segmenting RDMA transfers into smaller chunks in order to enable computation to be performed earlier than would be expected at the receive side. We show how our Network Interface can be simply modified in order to accommodate this optimization and discuss the possible implications of this.

6.1 Early Acknowledgement for AXI Writes

As mentioned in Section 3.2.2, we provide a method of Early Acknowledgements for AXI write transactions in order to increase the performance of the system, and reduce stalling in the pipeline of the in-order A53 processor. This performance enhancement is included for both shared-memory accesses and RDMA transfers. Obviously there is no performance enhancement that can speed up remote read operations as the data is required to allow the processor to continue. Therefore the systems software must aim to reduce the number of remote read operations using better memory management techniques and scheduling decisions, as they will have a severe effect on the performance of the system. This is a well understood limitation when programming on NUMA systems [204], [205].

It seems counter-intuitive to include such a mechanism for RDMA transfers, as the DMA engine does not suffer the same pipeline stalling issue as the Cortex-A53. However, the IP which is used to perform the DMA transfers, the Xilinx AXI CDMA engine (Central Direct Memory Access [16], providing memory-mapped interfaces) has a limitation in the number of concurrent outstanding transactions it permits, which limits the capability of the RDMA engine to saturate the network over long distances. This phenomenon is shown in Figure 6.1, and we have seen and discussed the implications of this issue arising in the Network Interface during our experiments in Section 5.3.3. Therefore we will include this mechanism in the RDMA transport in order to extend the number of transactions in future. The value required will need to be calculated based upon the number of packets which can form a fully pipelined flow to each end of the system, and must be set accordingly in a real world system.

Another reason for doing this is that the DMA engine is incapable of handling erroneous transfers and will simply set an interrupt pin high when an error is encountered. This must then be dealt with by some external mechanism such as a controller which will raise an interrupt to the OS and allow software handling. Since the design of our Network Interface includes a hardware mechanism for handling erroneous transfers there is no need to rely on software. As such, all transfers from the DMA engine are acknowledged as successful once the data has been transferred

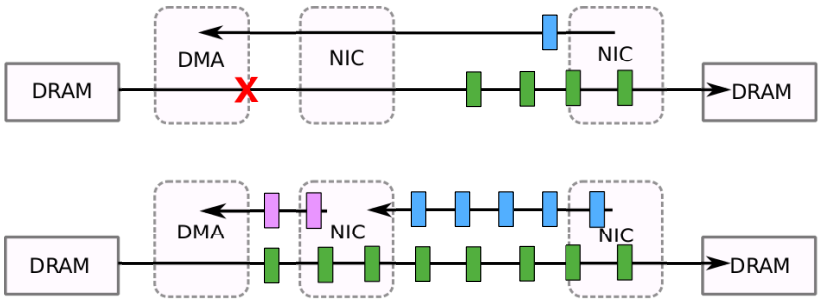


Figure 6.1: Limiting the number of outstanding operations limits the saturation of the network at certain distances.

into the NI from local memory/accelerator etc.

In order to allow for the early acknowledgement of the AXI write transactions a bank of timers is used; each of which has an associated ID. Upon the arrival of a write request, once the transport mechanism issues a timer the early acknowledgement response packet will be formed and returned to the initiator. Currently the design uses only 16 outstanding timers on the RDMA transactions. This is easily sufficient to saturate a 10G link at a distance of one hop, but should be extended in a full scale system to accommodate for an uninterrupted stream of data to be issued over a larger network. This value was initially used because the maximum possible concurrent transactions that the Xilinx CDMA can accept is 16 (although this is not the default configuration [16]), and so is an artefact of the implementation prior to the use of the early ACKs. An appropriate value for the maximum number of possible outstanding transactions is dependent on many factors, such as the network diameter and topology, network load, workload characteristics due to typical transfer sizes, routing algorithms, congestion control mechanisms etc. As such it is very difficult to determine an appropriate number of table entries to cater for with a prototype at this scale and sophistication.

If the timer overflows then a message is sent to the retransmission mechanism within the NI to handle the issue. This will either reinitialise the transaction, or in the event of multiple time-outs raise a system level interrupt in order to initialise the checkpoint-restart mechanism, as described in Section 4.1. If a correct acknowledgement of the packet is seen then the timer is simply stopped and replaced in the bank of unused timers. A notification is given on the receipt of the genuine acknowledgement to allow the user to poll to confirm the operation was successful.

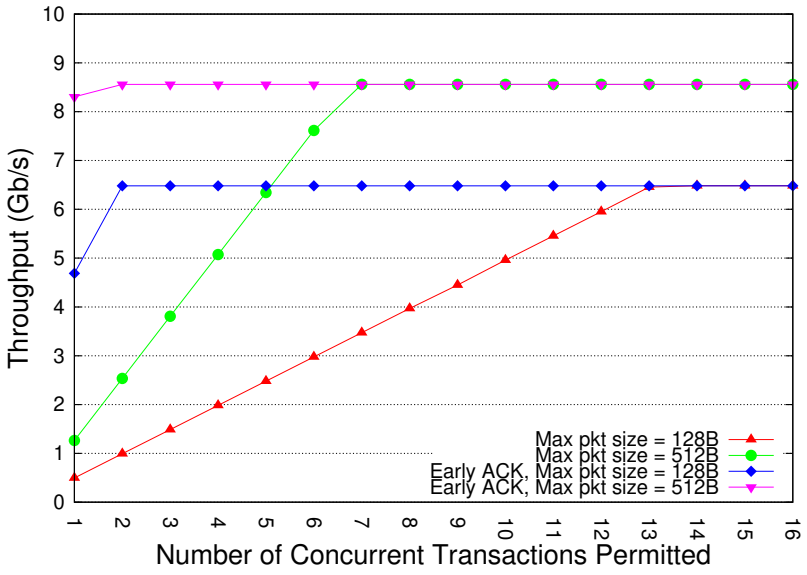


Figure 6.2: Throughput over outstanding packets, showing quicker saturation using early acknowledgement system.

The early ACK mechanism is bypassed when exclusive (atomic) accesses are issued (`AXLOCK=1`, see Appendix C). This allows for hardware exclusive accesses to be used. The system must wait for the full remote response in this instance, with no notification being sent but a direct AXI response packet being returned. In this manner the programmer must be aware of whether hardware or software based locking/synchronization messages are being used. As we discussed in Section 4.1.4.2 a method to allow this at the remote node is required, due to the inability of the CPU to give remote access based upon the specific source node. Unfortunately this is not yet implemented.

In Figure 6.2 we compare the results presented in Section 3.4.3 with the early acknowledgements mechanism. It is seen that the latency of waiting for acknowledgements is masked completely if more than a single packet can be in flight at any one time. The reason that there is degraded performance in the event that we have a single in-flight transaction limit comes from the implementation of the timer module for the early acknowledgement. When a transaction is presented a request for an available timer to begin takes 2 clock cycles. This timer can only be requested once the footer has been presented. What this means is that there is a bubble in the pipeline which is only masked when a separate request can begin writing data into the NI while the timer is being started for the previous transaction.

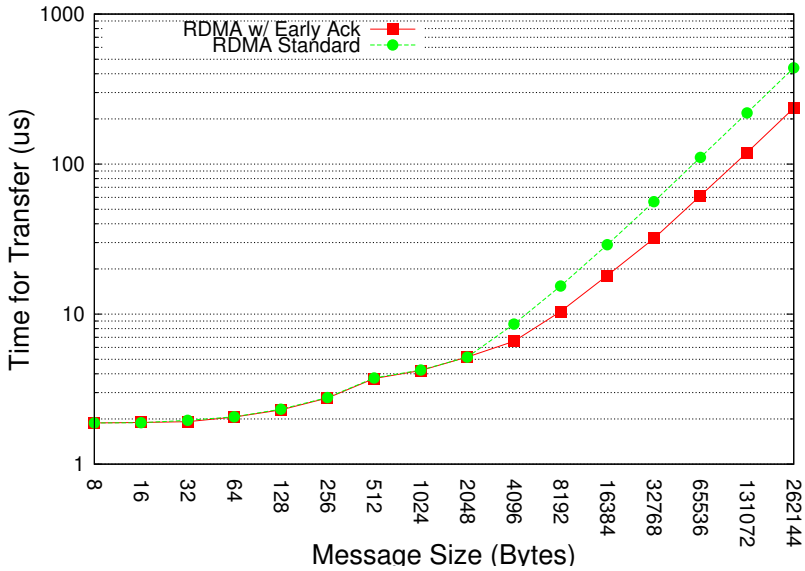


Figure 6.3: Reduced latency for transfer when using Early Acknowledgements mechanism.

We have set up an experiment which uses the full RDMA engine to send varying sizes of messages, pulling memory from the DRAM in the PS and sending it over the network, writing into the DRAM of the remote PS and waiting for every transaction to be properly acknowledged. This shows the effectiveness of adding early acknowledgements to the NI in improving the performance of the RDMA transfers.

Figure 6.3 shows the time taken for an RDMA operation to process and receive (genuine) acknowledgement for a given message size. The baseline for small sized messages is around 1.9μs, which stays consistent until the message size grows to over 64B. This latency is higher than the baseline we see in the results in Section 3.4.1, which is due two reasons. The first is that in this instance additional latency is incurred as the DMA destination is remote DRAM, as opposed to BRAM on the remote FPGA fabric. This DRAM access is obviously slower than accessing the static RAM on the FPGA.

The second difference is that the setup for these experiments uses a different, open-source 10G MAC/PHY layer[1]. We saw that the latency for transmission over the serial links was actually higher in this configuration than when using the Aurora PHY from Xilinx [17]. This was found to be due to a store and forward within this open-source MAC layer at the receiver.

[1]The IP was created by Alex Forencich, and can be found at-`https://github.com/alexforencich/verilog-ethernet`, accessed 2017.

We see the divergence in performance in the results such that for larger messages adding early acknowledgements to the system results in a transfer time of roughly half. This difference arises because the RDMA engine reaches its limit for concurrent transactions per channel. This difference would become even more pronounced if the distance between source and destination were more than a single hop, as the latency between issuing new transactions would grow even further.

6.2 Receiver Registration

In order to reduce the latency of RDMA operations at the receiver an optimization is required to enable the receiver to identify when an operation has been completed. The standard method for notifying the receiver that new RDMA data has been posted is to have the sender issue a notification/acknowledgement packet for the whole operation (this is issued separately, following on from the individual packet acknowledgements that the receiver posts back to the sender).

Figure 6.4 shows how we hope to improve the latency of RDMA transfers for the receiver. In this example there is an RDMA operation initiated, which is comprised of three data packets D0, D1, D2. In the unregistered (unoptimized) version the sender must see that all the packets have been correctly acknowledged before issuing a notification that the entire operation is complete. This is because the receiver has no knowledge of how much data it is to expect, as no state or connection information is held and the operation is one-sided.

In the registered (optimized) version we see that the NI at the receiver is now made aware of how many transactions it is expecting within the entire RDMA operation, and so can notify the processor or acceleration resources as soon as it sees all the data arrive correctly, saving a full round-trip latency for the notification.

6.2.1 Implementation

Figure 6.5 shows the NI with the *Receiver Registration* portion highlighted. Note that the module can be bypassed, necessitating the use of the unoptimized RDMA notification path. This is required and even desirable in certain situations which are discussed in Section 6.4. Operations are registered using a special packet type in the

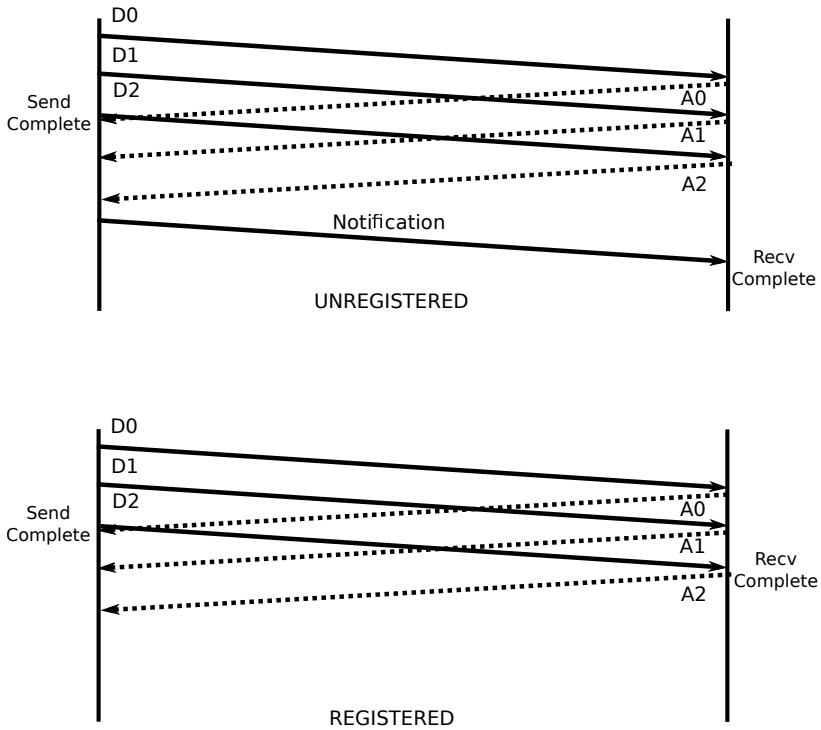


Figure 6.4: Time taken for send and receive operations to complete for registered and unregistered transfers.

"Type" field of the header. There are three possible "Types" for incoming RDMA data; RDMA_WRITE, RDMA_WRITE_REGISTER, and RDMA_UNREGISTERED (see Table 3.3 for all possible packet types). If the header indicates an RDMA_WRITE or RDMA_WRITE_REGISTER then the incoming packets are directed to the registration module. If the packet type is RDMA_UNREGISTERED then the packet bypasses the module and uses the standard path.

The first RDMA transfer which is sent (i.e. from the base virtual address of the sender) is given the RDMA_WRITE_REGISTER type. The address range for the RDMA operation is logged in a table. The upper range for the addresses is found by using another field in the header which contains the number of expected packets in the entire RDMA operation. Subsequent packets associated with the same RDMA transfer are then checked to confirm whether they lie within the expected range (and are from the same sending node). Once all of the expected transactions have been seen the entry is wiped from the table and the notification is built and sent to the local notification queue. Figure 6.6 shows how the registration is entered into the table when the RDMA_WRITE_REGISTER packet arrives. The associated data for the packet is queued inside the NI until the entry has been added, or in the case

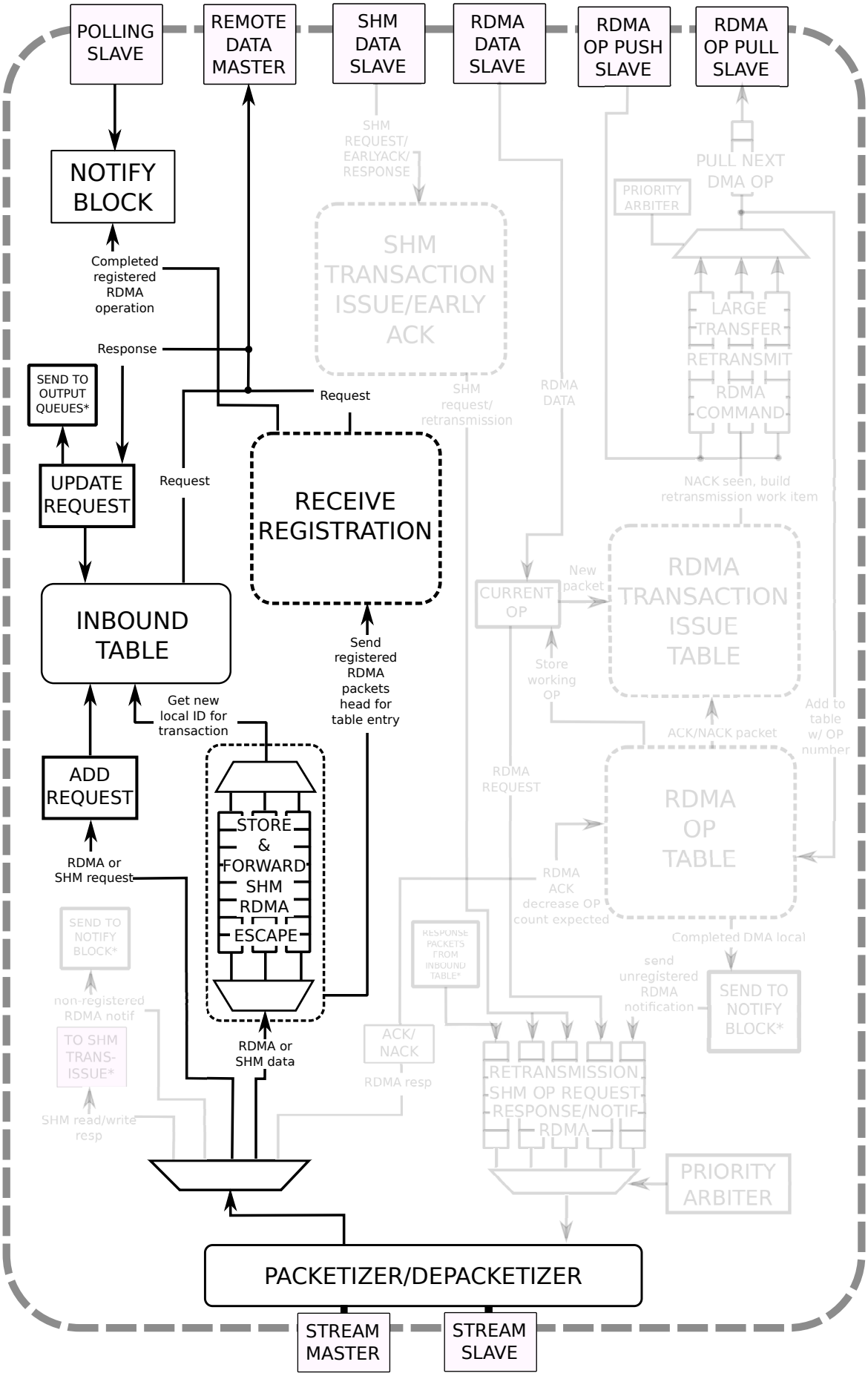


Figure 6.5: Highlighting of the receive-side of the Network Interface with registration architecture.

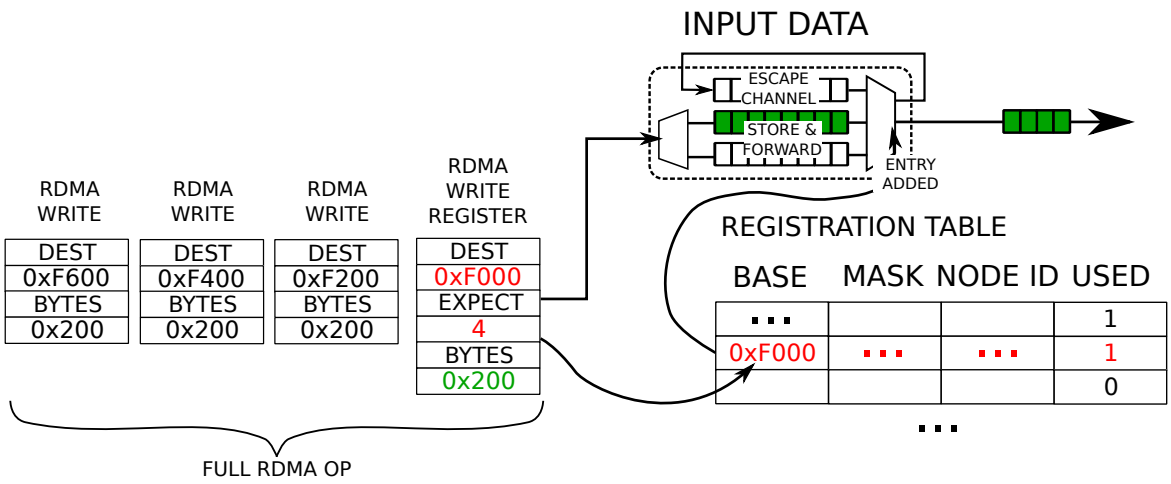


Figure 6.6: Adding a Registration Table entry when an RDMA_WRITE_REGISTER packet is seen, with subsequent packets in the full RDMA transfer arriving behind.

of subsequent packets the corresponding table entry has been found. Once the table entry has been successfully allocated the data is permitted to progress.

There are several architectural features here which mitigate against new issues presented by performing this optimization. In the unoptimized version of the RDMA transfer the intrinsic properties of the RDMA communications mean these issues do not present themselves normally. These issues are as follows:

- Maintaining a connectionless transport within the NI; any information stored regarding the RDMA operation must be transient, only being stored while the operation is in progress. This property is implicit in our implementation.
- Supporting Out-Of-Order (OOO) packet delivery; the system switch allows for OOO delivery of packets. In order to properly register and track the RDMA operation special measures must be taken to handle OOO delivery.
- Duplicate packets; under normal operation this is not an issue but given that the sender and receiver become further decoupled during the communication process duplicates must now be considered.

6.2.1.1 Out-Of-Order Delivery

Out-of-order packet delivery is handled by using the escape channel (Figure 6.7) for any packets which currently have not had a table entry created. Out-of-order

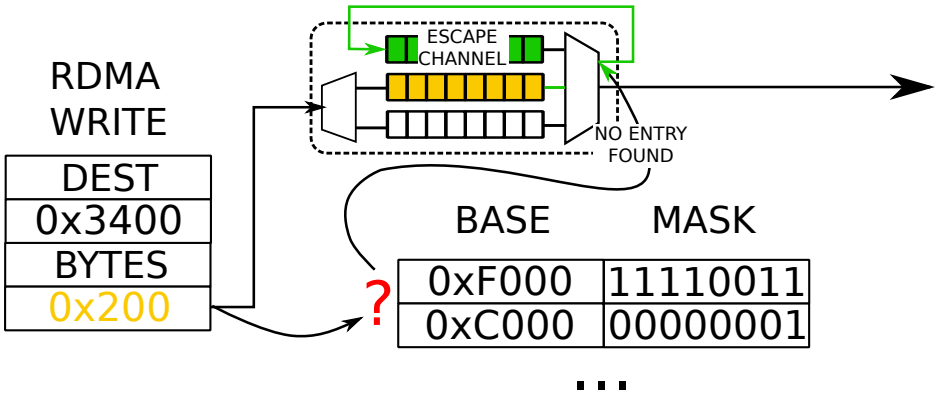


Figure 6.7: RDMA data arriving not associated with an active table entry, sent to the escape channel to prevent blocking.

packet delivery does not matter for data beyond the first RDMA packet which registers the transaction (see Figure 6.6). If an RDMA_WRITE packet is received before the corresponding RDMA_WRITE_REGISTER packet then there will be no valid table entry for the packet to match with.

Until the first packet arrives any out-of-order packets which arrive prior to this will be put in the escape channel in order not to stall the pipeline within the NI. We are able to do this because the data that enters the NI is written to memory in a store and forward fashion. The data can never be allowed to be written into memory until a CRC has confirmed the validity of the packet. This means there is N cycles latency between the first flit arriving and the packet beginning to be written to memory, where N is the number of flits within the full packet. During this time we are able to drain the previous packet into the escape channel, meaning that no additional latency is seen. Any packets sent to the escape channel have a timer associated with them. Once this timer expires the packet will be dropped. This can happen with OOO packets if the first packet in the operation is severely delayed or lost in the network, so registration never happens. In this case dropping the packet is safe because the sender will have received no acknowledgement or negative acknowledgement, and will itself time out and retransmit the packet. In normal operation once the RDMA_WRITE_REGISTER packet is seen the escape channel can be drained and the data sent to memory. In the event that duplicate RDMA_WRITE_REGISTER packets occur, these packets are also checked against the current table entries regardless, and so duplicates will be handled in the way we discuss below.

6.2.1.2 Masking for Duplicates and Completion

Once the RDMA operation has been registered a bitmask is used to determine when all corresponding packets have been received for the operation, and to drop duplicates. A `'b1` representing a packet that has arrived, and a `'b0` representing a packet yet to arrive. An initial mask needs to be calculated at the time of registration. This is required to account for the fact that an operation may be smaller than the maximum possible registered operation size ($Maskbitwidth \times Packetsize$).

This mask is created by a barrel shifter which uses a field in the header of the RDMA_WRITE_REGISTER packet which denotes the number of expected packets. We shift in zeroes to form the appropriate initial operation state. A single 1 is added to the end of this process (as the RDMA_WRITE_REGISTER packet has obviously arrived to begin this registration process). For example, if we assume a 4KB operation, a 16KB maximum mask size and a packet size of 512B, the initial mask after registration would be:

`'b1111_1111_1111_1111_1111_1111_0000_0001`

This is because 8 packets are needed for the transfer $512B \times 8 = 4KB$ and the first one has been received (the payload along with the RDMA_WRITE_REGISTER header). The bit mask has a width of 32 as each bit represents a 512B packet (for a 16KB maximum).

Every time a new packet arrives the table is checked in order to determine whether any existing entry matches with the new packet, calculating whether the incoming destination is within the range of the entry currently being checked. If the table entry being checked is found to match the incoming data then it proceeds to update the mask. Figure 6.8 shows how the mask is updated upon receiving a new packet. The base address of the entry and the number of bytes of the operation are used to calculate whether the operation being checked in the table relates to the new packet arriving. An offset is then created for a barrel shifter, which generates a mask to cause a bit flip. If the mask is found to be all `1`'s then the operation must be completed. If $Newmask = Originalmask$ then the packet must be a duplicate and can be dropped immediately.

Duplicate packets can also occur in the instance where the corresponding operation has already completed, and there is no longer a corresponding table entry.

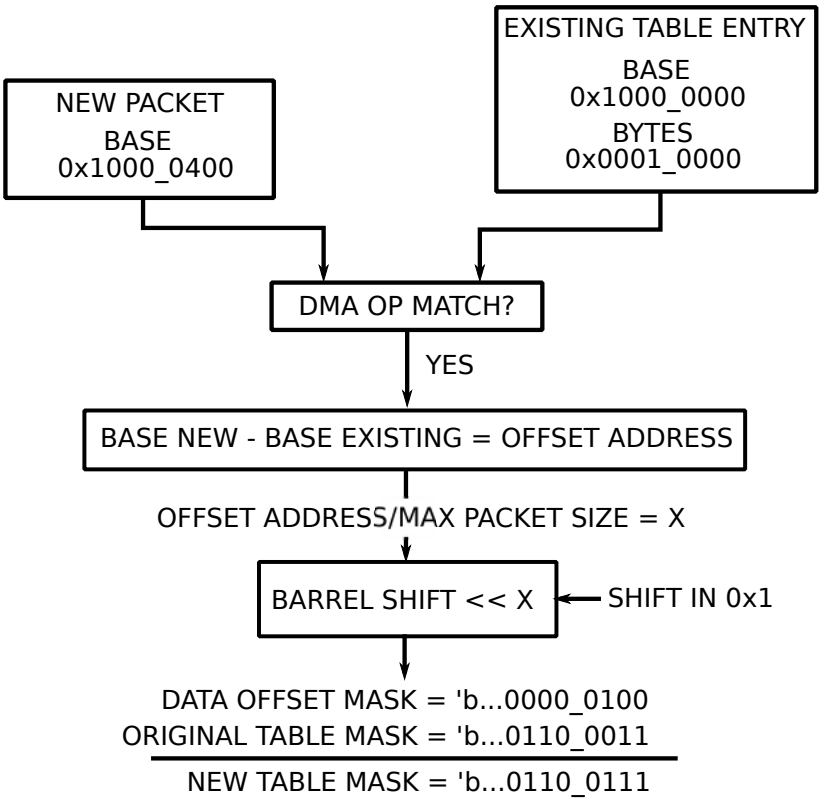


Figure 6.8: Creating the bitmask for the new DMA data, to check for duplicates and completion status.

In this case the packet will therefore be sent to the escape channel. Once the timer expires the packet is finally dropped. In this case dropping the packet is completely safe because the previous packet must have sent a correct acknowledgement back to the sender. In the event that data is found to correspond to an entry in the table but is a duplicate, the data can be safely dropped straight away and there is no need for the timer.

The stale duplicate data awaiting timeout can cause a blockage in the escape channel for the period before timeout occurs. This is self correcting after a time but may cause packet drops if the escape channel overflows. This should be an incredibly rare event however, and should occur only under exceptional circumstances. The possible negative effects on performance here being acceptable given the simplicity of implementation.

6.3 Segmentation

A key method for improving the performance of any distributed applications is to overlap computation with communication, ensuring that no resources are left starved of data to process. In our RDMA communication technique one of the main issues that arise from the transport mechanism is that an entire RDMA operation must be completed before computation can begin on the remote node.

In TCP for example this issue does not arise because of the use of send/receive queues and guaranteed ordering on data. To this end the receiver knows implicitly which data has arrived if it is aware of how many bytes it has received. In our transport layer this is not the case. As data may arrive out of order the receiver can only be notified of message reception at the granularity of a full RDMA operation. For a large RDMA data transfer this obviously causes huge issues for concurrent communication and computation, as the receiver must await notification of the entire RDMA operation's completion before it can work on the data.

In order to ameliorate the effects of this issue we aim to reduce the granularity for RDMA transfer notification from the operation level down to an arbitrary block level granularity. We do this by simply splitting a single operation into a number of smaller operations. Doing this obviously gives more frequent notifications at the receiver and allows them to use the data which has already been acknowledged. An example of this is shown in Figure 6.9 where in one instance a 3MB transfer is acknowledged in 1MB portions, allowing work to begin before the whole transfer is finished and thus completing work before the normal instance. Obviously the use of this technique requires that data dependencies do not affect computation on the data which has been acknowledged.

This technique can be performed very simply in the sender NI, as we catch issued RDMA requests before they are processed by the DMA engine. We can use this to create modifications to the RDMA operation which are completely transparent to the sending node. These modifications will segment an RDMA operation into multiple smaller operations, providing notifications to the receiver for each one.

Figure 6.10 shows how this segmentation mechanism works. If a given RDMA

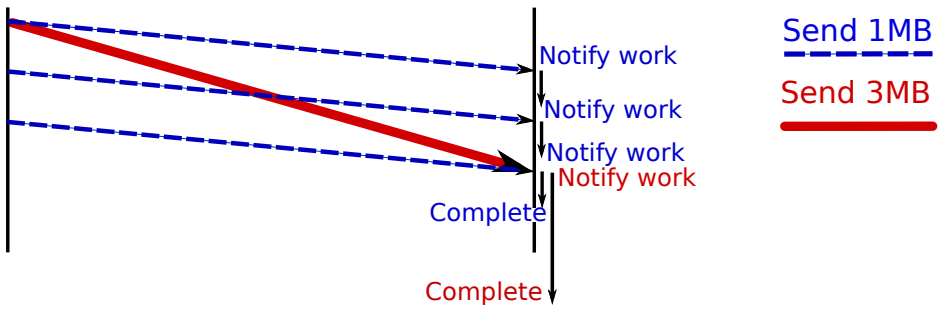


Figure 6.9: Timing for computation and communication with and without segmentation.

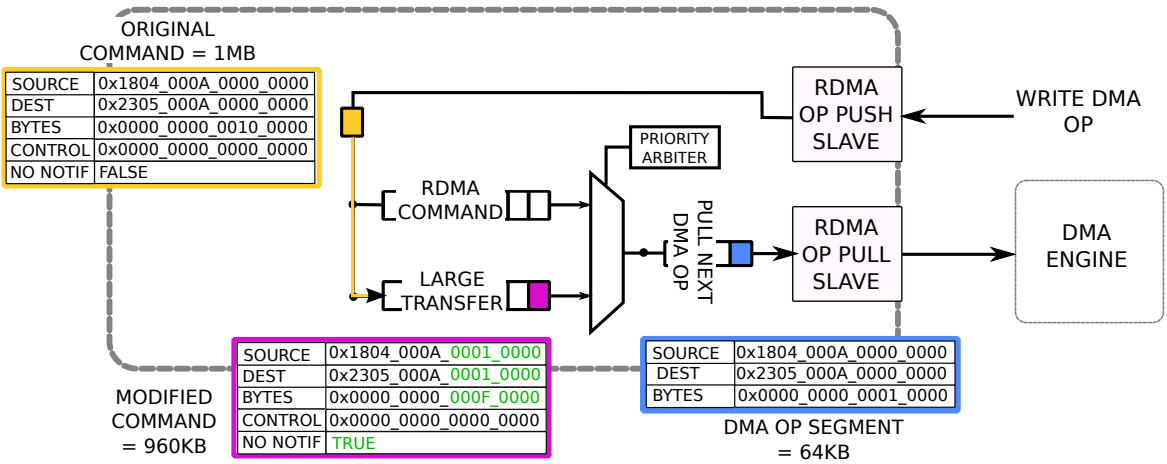


Figure 6.10: Segmentation of a 1MB transfer into 64KB segments.

work descriptor is seen on the NI and involves a transfer over a given size (configurable at runtime) it is sent to a special *large transfer* queue. In this instance when the operation comes to be processed and sent to the DMA engine it is assigned a special *NO NOTIF* status flag as it is issued an entry in the *RDMA Operation Table* (described fully in Section 5.2.3). Whenever an operation with this status flag set is completed we do not issue a local notification of completion to the sender. Only a notification to the receiver is posted.

If the command is of size M, and the maximum segment size is N, then the head of the *Large Transfer Queue* will remain in place for M/N RDMA operations. Following the issuing of each partial (segmented) operation we simply update the base address and number of bytes for transfer in the queue entry. We must also maintain a copy of the original base address and number of bytes for the eventual local send notification. As the last RDMA operation is issued (when the bytes to be transferred is below the threshold segmentation level) the *NO NOTIF* status flag is

deasserted. In this instance the local send notification is issued upon completion.

The threshold level at which this segmentation may be useful is highly dependent on many parts of the system. The structure of the application will have a great effect; with data dependencies within computation, the amount of data being transferred and any other unrelated processing all affecting effective exploitation of this technique. As well as this things such as the system scale and application task mapping within the system will have an effect. The use of this feature should therefore be left to the discretion of the application programmer, possibly with hints from the runtime system in order to exploit overlapping communication and computation during configuration of the NI. It is therefore very difficult to set up a blanket test to evaluate the effectiveness of this feature.

Once configured the use of this technique is transparent to the sender, as local notification only happens upon receipt of the full transfer. The user is therefore free to transfer very large blocks of data using a single work entry to the RDMA engine, with the NI handling the segmentation. An extension of this modification to the NI is the ability to schedule smaller RDMA operations before larger ones. We naturally separate out larger transfers and segment them, and also priority can easily be given to one of the other queues (retransmission or normal operation). Adding more queues for more complex arbitration schemes would be a trivial extension, but is beyond the scope of this thesis.

6.4 Performance of Receive Block

The receiver registration block can obviously provide benefits to communications with low-latency requirements. The question remains as to how far these benefits will be seen within the context of actual communications within a large scale system. There may be types of communications for which it is undesirable to provide fast receive notifications, such as in large-scale many-to-one/many-to-many collective operations. In instances such as these there may be too many simultaneous operations directed to a single node, causing the table within the receive registration module to fill to capacity. If this happens then packet dropping can occur. On the other hand these collective operations are likely to be comprised of small

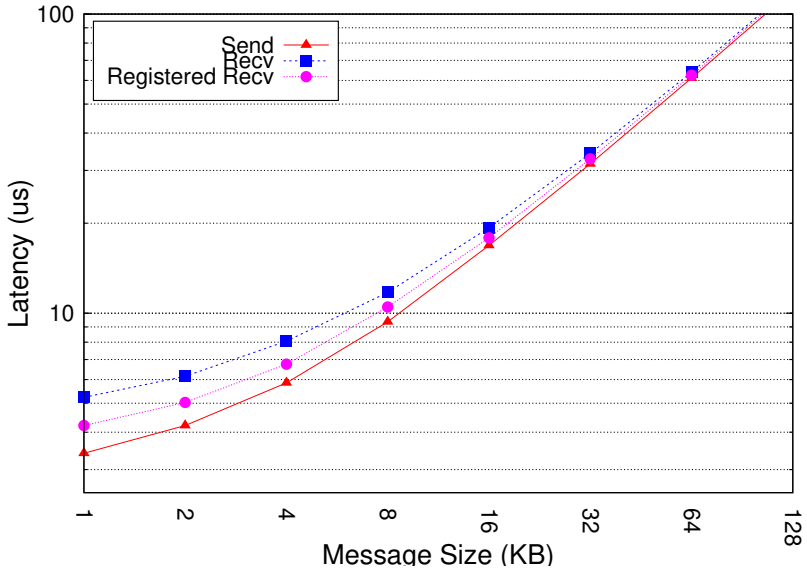


Figure 6.11: Latency for send, receive and registered receive operations over a single hop distance.

to medium sized messages, meaning that the registration table entry may be very short lived, in which case the hardware may be able to cope with this.

Another way of avoiding this issue is for collective operations to be implemented such that they use algorithms which share data in tree-like structures, reducing the volume of messages dramatically. Performing collective operations as a series of point-to-point messages with optimized algorithms for data-size is standard practise within MPI implementations; using algorithms such as recursive doubling or binomial tree for broadcasts for example [206].

Another important aspect in the decision of whether or not to register the receive operation is the latency savings that can be found against the overall time for RDMA operations to complete. If the latency of sending the data is large enough, then the additional latency of the round-trip for notification will be amortized. Figure 6.11 shows the results of an experiment to show the performance benefits of registered receive side transactions and the gradual amortization of the performance benefits as the transfer size increases.

The results show the latency for the transfer of data and notification of completion in a user-space application for a single hop transfer. The latency of the send operation is the time taken to configure the DMA engine from user-space and for

receipt of the transaction to be given back to the user, indicating that the DMA engine has pushed all the data into the network. The measurement we take is thus for a `send()` function within our API (see Section 3.3.4.3); the time for the data to simply enter the network. A blocking `send_blocking()` function would have higher latency than the registered receive operation since it must wait for the last ACK to arrive, ensuring the data has been correctly received. The latency of receive operations are measured from the time that the sender begins to initialize the transfer until the receiver gets notification that the RDMA data is placed in memory; either by local notification from the NI (Registered) or as a notification packet from the sender (Non-Registered).

It is important to note that conceptually these functions are different from the non-blocking and blocking MPI functions, in that the blocking `MPI_Send()` will return once the send buffer is free (assuming `MPI_Recv()` has been posted). This means that the data could simply be copied out to another buffer rather than having been actually sent to the network and seen at the receiver. The non-blocking `MPI_ISend()` will return regardless of whether the receiver has posted an `MPI_Recv()` or whether the send buffer is free for reuse.

We see that the latency of a receive operation for a 1KB transfer is around 5.23μs, and for a registered receive is only 4.21μs, cutting $\approx$20% from the latency of the operation for the receiver. We also see that the performance gains from this registration technique diminish with transfer size and become insignificant at around 32KB. At much larger transfers the measured latency for send/recv/registered recv are very similar, as is seen in the convergence of the results.

What this means in practice is that registered transactions will only show significant benefits within a certain range of smaller message sizes. The extent of this range is highly dependent on the distance from the destination and the network load, as both of these properties of the network increase the average latency of transmission. As the distance between source and destination increase or the load of the network goes up, we would see larger and larger message sizes be able to benefit from receive side registration.

Given the destination address we are able to determine very simply the distance between sender and receiver. This is because the geographic addressing

scheme employed within the system [32] makes this a simple arithmetic operation. It should therefore be very simple to adjust the threshold levels for performing receiver registration based upon the source-destination distance. This is particularly true given the fact that the technique can be employed transparently from the point of view of the application. Unfortunately there is currently no simple way of adjusting the threshold value for receiver registration and notification based upon the network load. Doing this would require input from monitors within the network, and then fine tuning of latency models based upon these inputs. This is complex work which lies well beyond the scope of this thesis.

6.5 Receive Module Scalability

As we discussed above in Section 6.4 there will be high variability in the expected performance gains which can be found from utilizing the registered receive module. This depends on the distance of communications, the expected size of RDMA operations, the network load and the communication patterns (e.g. many collectives or lots of point-to-point messages). It is therefore appropriate to provide information regarding different possible configurations for the module, in order to trade off the capabilities of the module against the overheads (in terms of area) on the FPGA.

In our parameter sweep we vary the width of the bit-mask which determines the maximum possible RDMA operation size that can be registered and tracked. We also vary the number of table entries, which determines the number of possible concurrent outstanding registered operations. Table 6.1 shows the area utilization under different configurations. We consider bit-masks which provide for maximum RDMA transfers of between 32KB and 512KB. This uses a maximum packet payload length of 512B. This small packet length is used throughout the thesis, as we wish to maintain a small packet size in order to maximise the load balancing within the network when used in conjunction with the proposed packet-level adaptive routing algorithm (Section 3.1.3). The number of outstanding operations is varied between 64 and 1024 entries.

From the results we see that varying the maximum possible size for the RDMA

Table 6.1: Area utilization (% total) for various combinations of maximum packet size and table depth. Total LUTs = 274080, total BRAMs = 912. Implemented on the Xilinx Zynq xczu9eg-ffvb1156-2-i FPGA.

		Bitmask Vector Size									
		64 (32KB)		128 (64KB)		256 (128KB)		512 (256KB)		1024 (512KB)	
		LUT	BRAM	LUT	BRAM	LUT	BRAM	LUT	BRAM	LUT	BRAM
Table Size	64	2230 (0.81)	47 (5.15)	2936 (1.07)	49 (5.37)	4286 (1.56)	53 (5.81)	6589 (2.40)	60 (6.58)	11072 (4.04)	74 (8.11)
	128	2282 (0.83)	47 (5.15)	2942 (1.07)	49 (5.37)	4289 (1.56)	53 (5.81)	6501 (2.37)	60 (6.58)	11092 4.04)	74 (8.11)
	256	2266 (0.82)	47 (5.15)	2973 (1.08)	49 (5.37)	4366 (1.59)	53 (5.81)	6842 (2.49)	60 (6.58)	11124 (4.05)	74 (8.11)
	512	2298 (0.84)	47 (5.15)	2974 (1.08)	49 (5.37)	4367 (1.59)	53 (5.81)	6843 (2.49)	60 6.58)	10965 (4.00)	74 (8.11)
	1024	2273 (0.83)	47 (5.15)	2976 (1.09)	52 (5.70)	4363 (1.59)	57.5 (6.30)	6575 (2.39)	68 (7.45)	11088 (4.04)	89.5 (9.81)

operations has little effect on the number of LUTs used. This is because the logic to decode/encode the mask is not significant, compared with other components in the module. The number of BRAMs jumps considerably at certain boundaries, which is due to the odd bit width of the table entries. Effectively this creates a scenario where we can gain "free" entries to the table because of the fixed size BRAMs on the FPGA being utilized more efficiently. It is also worth noting that the number of BRAMs for the smallest 64x64 configuration does not arise from the implementation of the table. This utilization is because the storage for the data in the escape channel is set to enable 64 full packets to be held in the NI. This uses 43 BRAMs, which is why we still see this baseline value for the BRAM utilization even in small configurations. This value is acceptable and not prohibitive for the implementation of accelerators in combination with our NI; with the largest possible configuration only requiring 10% of the total BRAMs and uses no DSP slices, which are key for performing efficient floating point arithmetic in the accelerator.

6.6 Concluding Remarks

In this Chapter we have addressed some of the limitations which present themselves in the design of our basic Network Interface and transport layer, describing several optimizations to enhance the performance of the system. This work provides contributions in the following ways:

1. We present a novel microarchitecture for a module at the receiving side of the NI, which permits true one-sided communication and eliminates the requirement for the sender to acknowledge the completion of an RDMA operation.

2. We analyse the performance benefits that can be seen when using the registered receive module, showing that latency improvements of around 20% can be seen when sending small amounts of data over short distances (single hop) when compared with a standard receive operation. We also demonstrate the diminishing returns in performance benefits which are seen as the latency of posting completion notifications from the sender becomes amortized by the overall data transfer time.

3. We show the scalability of the receive-side module solution, and show that modest area overheads are seen even in the most aggressive implementations.

4. We provide simple architectural enhancements which can segment large RDMA transfers very easily into smaller ones, and show that providing early acknowledgement for AXI transactions can lead to higher performance due to the widening window of concurrent transactions within the network.

Chapter 7

Enabling Standalone FPGA Computing

In the previous Chapters we have presented a novel Network Interface design, capable of providing reliable shared-memory and RDMA transfers between distributed resources. The design includes hardware offloading of the transport layer and a connectionless approach which maintains little state information within the NI. This is necessary for facilitating the uptake of FPGAs within the context of HPC.

Our solution provides the capability for the FPGA to gain access to remote resources within a fully global address space, with memory, accelerators and storage all accessible as memory-mapped regions within this space. This facilitates not only the use of the FPGA as a full peer on the network, decoupled from the CPU resources, but also enables the FPGA to be upgraded to the status of the main compute element within the system. The aim is to reduce the role of the CPU to that of a mere control unit, managing the computation and datapath on the FPGAs, leaving the vast majority of the computational work to be performed within the FPGA. This is viewed by many as the only way to effectively exploit the extreme parallelism and reduced energy consumption afforded by FPGAs, and is key to their uptake within the wider HPC community.

As we discussed in Section 2.4, there are many works which have attempted either to bring the compute closer to the network, or have tried to provide a shared-memory environment for the FPGA by attaching it to the system-bus of the processor. However, there has been very little work in marrying these two concepts,

enabling irregular memory accesses over the network in a distributed FPGA environment. This is the main way in which our system goes beyond the current state of the art. We provide a general interconnect which enables tight FPGA-system memory coupling (for both local and remote resources) as well as tight FPGA-network coupling, all using a simplistic programming model for applications designers.

We have already demonstrated the additional flexibility our NI offers with regards to disaggregating CPU and FPGA resources for network functionality, whilst maintaining the ability of NUMA-like shared-memory accesses between CPU and FPGA. In this Chapter we demonstrate the effectiveness of our solution, showing the performance gains that can be seen when using our fully hardware-offloaded transport layer. We compare our solution against the control and data path that would be necessary for a software based approach to support reliable transmission over the network. Our experiments show that the reduced path complexity and memory copying can provide great benefits to the latency and throughput characteristics of the system.

We will also demonstrate how our results can be used to estimate the memory-bound bandwidth of the network when using arbitrary accelerator blocks. This can be used to allow potential applications developers to gain insight into the suitability and scalability of their accelerator on the system, and help them to ensure that the network does not become a bottleneck to performance.

7.1 Reduced Complexity in Data/Control Path

One of the main reasons for decoupling the CPU from the FPGA and providing a hardware-offloaded transport layer is to reduce the complexity of the control and data path when performing networked operations from the FPGA. As discussed in Section 2.4 the traditional means of allowing FPGAs to communicate with one another without the use of the CPU is by using a dedicated network, segregated from the CPU. This typically limits the scalability of the distributed FPGA resources to a single rack, as these separate inter-FPGA networks are typically formed of point-to-point links only [111], [112], [114], requiring communication through a separate network accessed via the CPU to communicate beyond the rack. These often simple

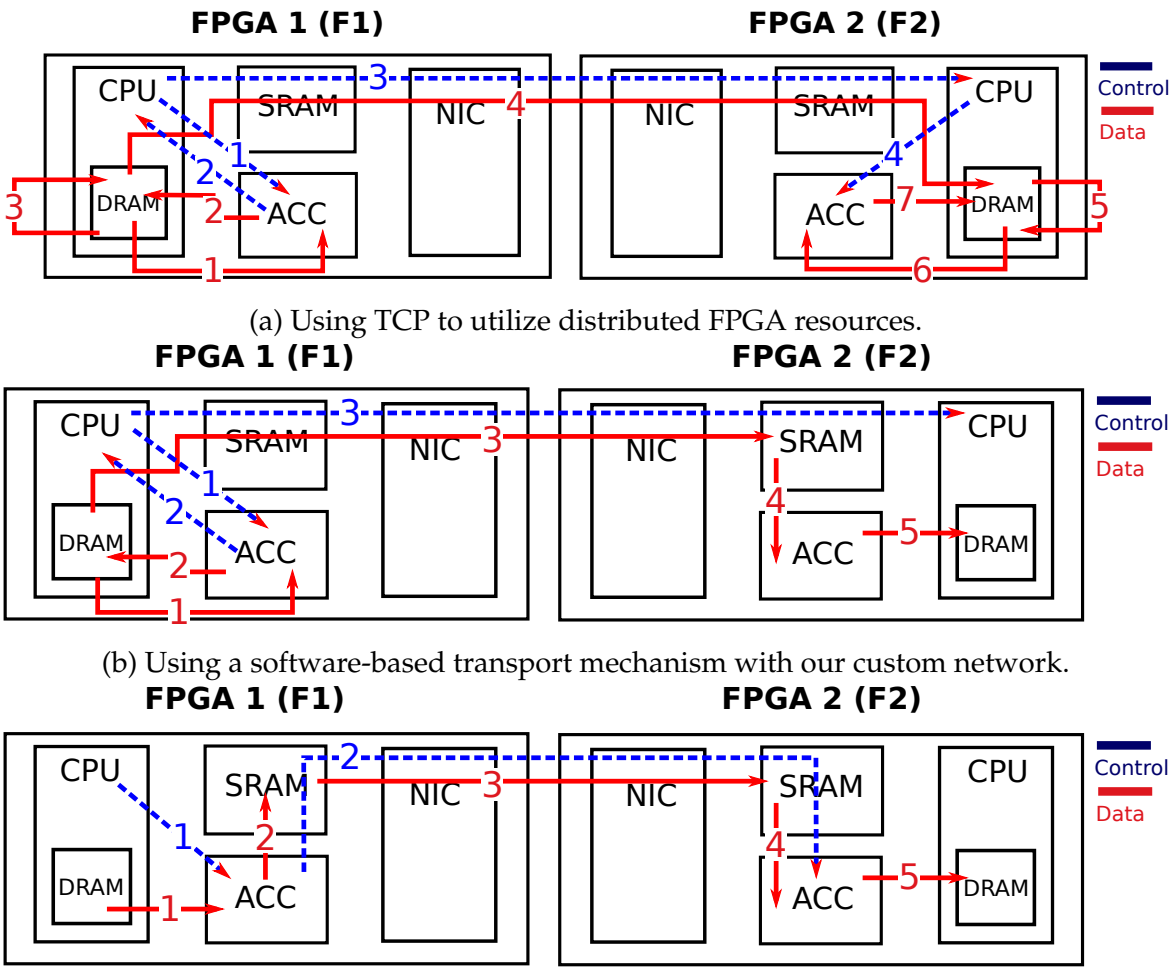
chip-to-chip communication layers may also prevent interesting topologies from being built, as they may be incapable of switching. Every FPGA must be an endpoint. This increases the diameter of the network and limits potential workloads to those which form large rings of dataflow for example.

The other typical method for inter-FPGA communication is typically via the software networking stack. OS intervention is often required with the FPGA only able to communicate with its local CPU via the local system bus, or worse still over PCIe lanes. Relying on the CPU for performing inter-FPGA communications is antithetical to the idea of the FPGA as a standalone compute resource, and will often involve extra data copying or additional stages of synchronization between the CPU and FPGA. All this does not lend itself to a dataflow paradigm, or to low latency access to remote resources (key for irregular memory accesses/pointer chasing [56]). In the following Sections we will show how our solution reduces the complexity of inter-FPGA communications.

In each of the following scenarios (shown in Figure 7.1) we see a different setup for data and control flow in distributed FPGA computing. The aim is for data to be sent from local DRAM to an accelerator, the products of the accelerator block on a first FPGA (F1) are then to be sent to a remote accelerator block on a second FPGA (F2) for further processing. The results of this second computational stage are then moved into main memory on the second FPGA. This provides an example of a simple dataflow type application utilizing distributed FPGA resources for computation. In each of the setups we show the complexity of the control and data paths accordingly.

7.1.1 TCP Communications

In Figure 7.1a we see the critical path for data and control flow when using a standard software TCP solution to perform the distributed FPGA computing task. In order to perform reliable transfers from source to destination TCP sets up dedicated send and receive buffers for every connection. The TCP protocol requires copies of the data to be made and placed in the send buffer for transfer, which creates additional latency. A copy is also required between the receive buffers and user-space memory in F2. This is why we see two redundant data copies to the



(a) Using TCP to utilize distributed FPGA resources.

(b) Using a software-based transport mechanism with our custom network.

(c) Using our hardware offloaded transport layer.

Figure 7.1: Flow of data and control when using distributed FPGA resources using different transport layers.

same RAM. Offloading of these buffers to hardware to be closer to the Network Interface would still require an additional copy stage, and also use excessive memory resources. Doing this would require limiting the scalability of either the window size, or the number of concurrent TCP connections that can be kept track of.

As well as this additional data copying the CPU at F2 needs to be responsible for notifying its accelerator of new work to be done, requiring additional control information to be sent. Therefore additional latency is added as the CPU in F1 cannot initialize work directly on the remote FPGA (F2). Our solution alleviates this requirement by writing directly into memory, and using a geographic addressing scheme as a means to locate the destination node.

7.1.2 Software Based Transport Using our Networking Stack

In Figure 7.1b is shown a solution which utilizes our custom network protocol, with the ability to write directly into memory-mapped regions within a global address space. However, this scenario still uses a software based transport layer. In this instance the CPU at F1 is able to submit work directly to the remote accelerator at F2 using a direct shared-memory communication mechanism similar to our own.

However, the data to be transferred must still first be copied back into DRAM by the F1 accelerator in order for the transport layer to send it. This is because the CPU has no knowledge of the accelerator's work status, and the accelerator has no knowledge of the RDMA transfer status. In order to copy directly from the block RAM in the FPGA fabric to the remote accelerator there would have to be an additional synchronization stage between CPU and accelerator. This is to ensure that data in the process of being copied to remote locations is not overwritten. This solution also results in additional communication requirements between the CPU and accelerator. As shown in the diagram there is an additional stage of notification between the accelerator and CPU in the critical path, used to inform the CPU that it has completed its work and the RDMA transfer can be initiated. This is not required if the accelerator can initiate network transfers itself.

7.1.3 Fully Hardware-Offloaded Transport, CPU Bypass

Figure 7.1c shows the control and data path when using our hardware-offload transport solution. In this instance once the accelerator has completed its work it issues an RDMA operation directly to the NI, and then writes shared memory operations to the F2 accelerator's work buffer. This informs the F2 accelerator that there is new work to be performed. Once this is completed then the F2 accelerator notifies its local CPU that the work has been done and it has new data to process.

As is shown, this solution is far more amenable to dataflow type processing, allowing for simpler pipelining of data through the distributed FPGA resources than in a traditional software approach. Since the accelerator is in control of both the RDMA transfer to the remote note as well as its own work scheduling, it can be used to simply coordinate the transfers and destination buffers for work. This

creates a solution where it is feasible for the accelerator to issue RDMA transfers directly from its own internal memory.

While several other solutions [11], [111], [124] allow for this sort of dataflow processing it is typically performed using only point-to-point links between the FPGAs, severely limiting the topologies which can be created [11]. This in turn limits the scalability of the utilization of FPGA resources to those located within a single rack (inter-FPGA connections are typically not performed using suitable wider network protocols, but on their own internal dedicated network). Combined with our switch design [32], our solution can exploit modern HPC topologies such as Jellyfish, Dragonfly and Fat-Trees. As well as this there is typically no mechanism to enable tightly coupled shared-memory between CPU and FPGA, or between distributed FPGA resources. This has been identified in [9] as a key barrier to enabling a number of HPC workloads to be efficiently implemented on FPGAs.

7.2 Experiments

Our solution offers an architectural advancement over the current state of the art, in that it allows full disaggregation of the FPGA fabric from CPU resources, whilst maintaining the possibility for tight coupling between CPU and accelerator memories. We also wish to show the benefits which are achievable in terms of latency and throughput for distributed acceleration resources, which the experiment in this Section provides. Figure 7.2 shows the complete system networking stack, which includes all the necessary interconnection components to support interfacing for accelerator IP and the Processing System. Table 7.1 gives a breakdown of the individual components on the FPGA. The full stack uses around 17% of the logic resources of the FPGA. This is a typical range [207], and we see that the resource consumption of our Network Interface is in line with others in the literature which provide similar performance/connectivity [12].

7.2.1 Simple TCP Test

Our experiment following on from this Section in 7.2.2 and 7.2.3 tests the scenarios discussed above in Section 7.1. Setting up the TCP example scenario to utilize

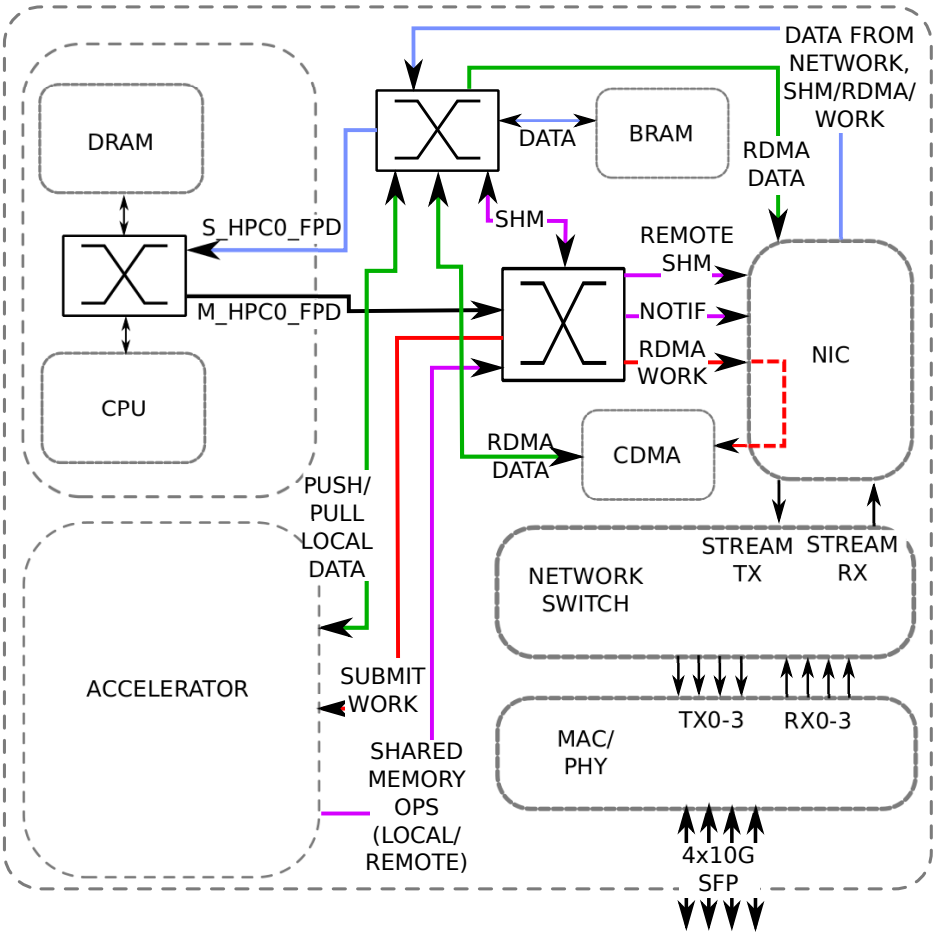


Figure 7.2: Full networking stack supporting accelerator and CPU access to the NI, implemented within the Xilinx Zynq Ultrascale+ device.

Table 7.1: Resource consumption of the full networking stack, and breakdown of individual components, implemented on the Xilinx Zynq xczu9eg-ffvb1156-2-i FPGA.

	LUTs (%)		REG (%)		BRAMs (%)	
Available	**274080**	100	**548160**	100	**912**	100
Total Networking Stack	**46915**	17.1	**58734**	10.7	**140.5**	15.4
Network Interface	**22670**	8.27	**19486**	3.55	**95.5**	10.5
Shared Memory Transaction	4204	1.15	5042	0.9	0	0
RDMA OP	2839	1.0	903	0.2	9	1
RDMA Transaction	1978	0.7	3421	0.6	0	0
Inbound Messages	3294	1.2	1512	0.3	0	0
Receive Registration	3347	1.2	559	0.1	49	5.4
Other	7008	2.5	8049	1.5	37.5	4.1
Processing System Interconnect	**5589**	2.0	**4031**	0.7	**0**	0
CDMA Engine	**2822**	1.0	**4043**	0.7	**0**	0
Memory Interconnect	**6060**	2.2	**6124**	1.1	**0**	0
Network Switch (4 port +1)	**2020**	0.7	**2211**	0.4	**25**	2.7
Aurora IPs and FIFOs (4x10G)	**3672**	1.3	**8564**	1.6	**20**	2.2
Other	**4082**	1.4	**14275**	2.6	**0**	0

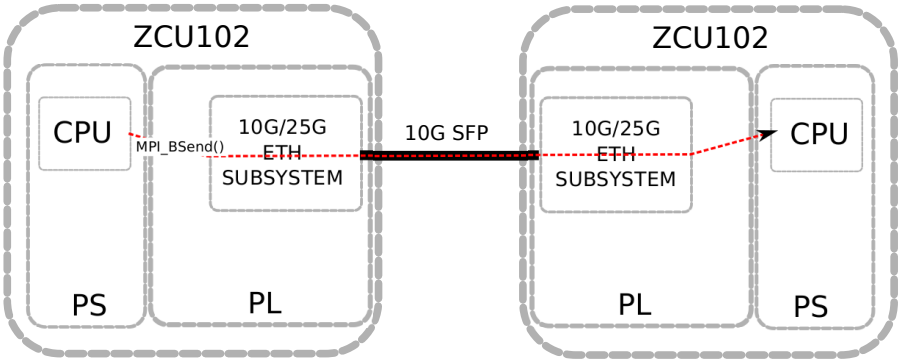


Figure 7.3: Simple setup to test MPI over a standard 10G TCP connection.

distributed acceleration resources would be a cumbersome and unnecessary process, as we are already aware of the severe limitations in using TCP. Instead we provide a simple comparison between two send operations to show the performance degradation which arises when using this more traditional FPGA networking solution.

We have set up two Zynq Ultrascale+ ZCU102 boards up to utilize the 10G SFP ports, as is shown in Figure 7.3. We have then run a very simple test, sending data using MPI over sockets based TCP over these 10G links. The MAC and PHY layer are provided by the standard Xilinx 10/25G MAC/PHY IP [208], which is implemented within the fabric of the FPGA. While the Zynq chip provides many transceivers, these are only accessible via the programmable logic. The only Ethernet port available directly from the hardened processing system is a 1G connection. It is for this reason we must go through the FPGA fabric.

Table 7.2 shows the results of a simple `MPI_BSend()` operation of a small and large message, against a `send()` operation using our hardware-offloaded, custom solution. The `MPI_BSend()` *MPI Buffered Send* is a send operation which uses a pre-allocated buffer (in our case set to 1MB in size), which will return as soon as the send buffer is freed, meaning that the data to be sent must have been copied out of this buffer and into the network. This enables a fairer comparison versus our sending function when compared with a standard `MPI_Send`.

It is very difficult to create a completely fair test in this manner, as our hardware currently has no available runtime system for distributed applications, so the operations act at a lower level than when running MPI. While this fact may affect the latency of the smaller packet transfer, this overhead would be easily amortized in

Table 7.2: Latency for a `send()` using our network protocol, and a buffered `MPI_Bsend()` MPI message over a 10G TCP/IP connection.

	Send Transfer Size	
	1KB	**1MB**
10G Ethernet (TCP)	8.5μs	1.58ms
Custom 10G NI	3.4μs	957μs

the larger message by the network transfer. Since we see a dramatic difference in the latency of large transfers, we can see that using a standard TCP solution (as many FPGA based networking solutions do) is inferior to using our custom interface.

7.2.2 Using Distributed FPGA Resources

We currently have no available runtime environment for running distributed applications on our hardware, so in order to measure the latency of the system accurately at the application level we are restricted to the use of a single FPGA for our experiments. We do this in order to ensure clock consistency and accurate timing measurements. By measuring the time from a single user-space application controlling two separate networking stacks there is no need for synchronization between distributed CPUs. Nor is there any need to perform averaging of ping-pong communications. We can run this experiment with no interference between the two stacks at the application level due to the nature of the `send()` operation in our API, discussed in Sections 3.3.4.3 and 6.4.

To ensure we emulate the distributed setup with complete accuracy the entire hardware acceleration and networking stack is implemented twice within the same FPGA fabric. We then use a partitioned memory space to maintain complete independence between the two portions. Not only are the memory spaces segregated, but the data path to the memory hierarchy of the CPU is also completely segregated. Every component of the networking stack and the accelerator block is implemented on the board twice (with the exception of the MAC/PHY block, which cannot be duplicated as there are only 4 SFP ports on the development board). A simplified version of this setup is shown in Figure 7.4. The second instantiation interfaces with the hard-core Processing System via separate master and slave interface ports, and the two subsystems are connected to each other as a network via separate TX/RX

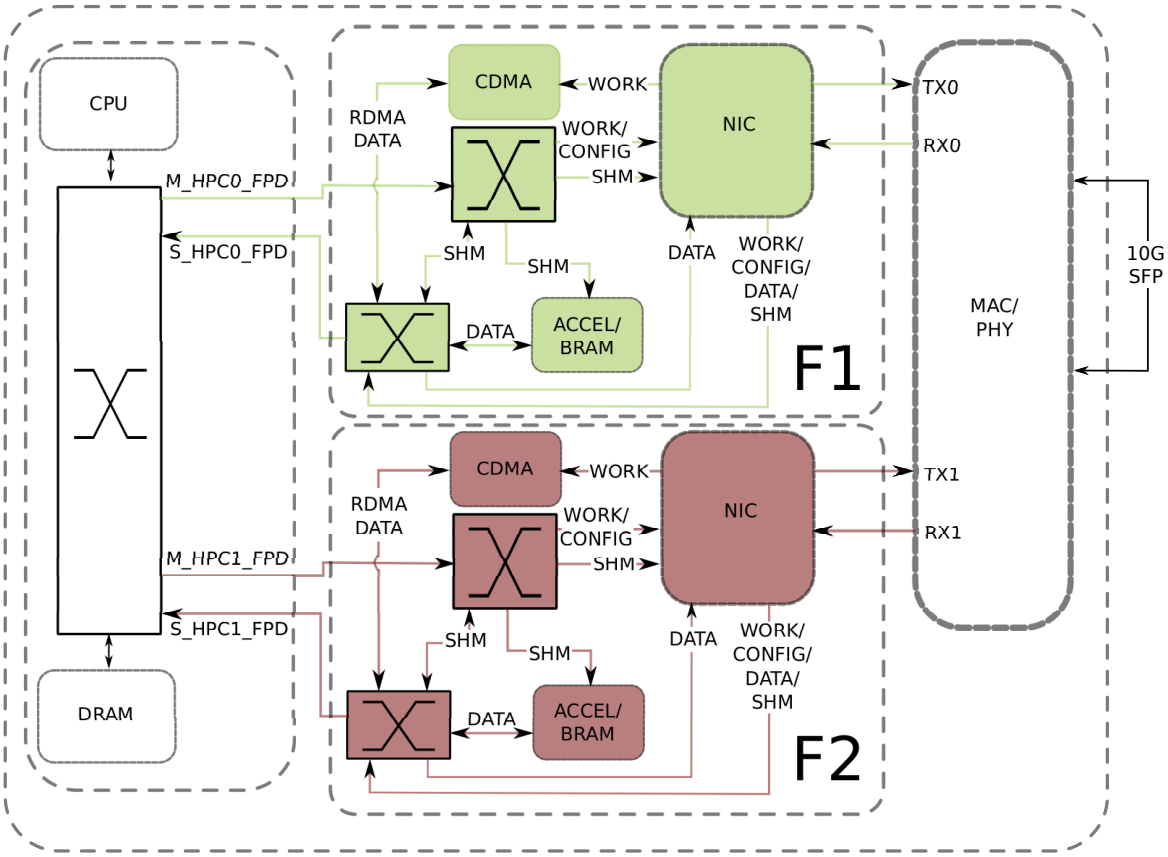


Figure 7.4: Setup emulating distributed system on a single FPGA, with logic for two implementations containing completely isolated address maps on a single FPGA.

ports of the transceivers and interfaces of the MAC/PHY block. This eliminates any contention for resources, and emulates accurately a fully distributed environment.

In the experiment we create the two scenarios described in Figure 7.1b and 7.1c. A user-space application takes a system time-stamp before submitting work to the local F1 accelerator. We use a dummy accelerator block which does not perform any function, but merely adds a latency between blocks of data being pulled in and written back out. The accelerator blocks on both F1 and F2 instances are configured with the same latency, emulating the *computation time* for the IP block. For simplicity equal amounts of data are written out as are read in. Once the CPU is notified of completion of the final write stage from the accelerator at F2 into memory it will take another system time-stamp to determine the overall time for the application to run at the application-level.

It is worth noting that in the setup for Figure 7.1b, we *only* perform the additional communications for the control and datapath that would be required when using a software based transport layer. Additional performance degradation would

be seen in any genuine implementation due to the additional overheads of the software transport layer (system calls/additional memory copies etc.).

In our system the user is able to issue RDMA commands and perform NUMA operations directly from user-space with no OS intervention. This is currently performed by `mmap()`'ing regions of dedicated memory space, but could be done using a global address space translation mechanism if the available IOMMU within the hard-core processing system is properly configured. This limitation is discussed in greater detail in Section 3.2.1 and Appendix B.

7.2.2.1 Data Blocks into Accelerator

Data must be transferred to the accelerator and worked on in blocks, as would be the case in typical implementations for HPC operations such as matrix-matrix or matrix-vector multiplication. This is opposed to creating a fully streamed pipeline of data in and out of the accelerator block (as in FIR filtering perhaps). This model emulates how a common HLS accelerator implementation may process data, copying chunks in using the `memcpy()` function available in the Vivado HLS tool for example[1].

We adjust the block granularity for the data and adjust the *computation latency* with relation to the pure communication time for a given solution (latency for transfer of data of a given block size and control information through system). A computation/communication ratio of zero denotes instantaneous processing time; data (at block level granularity) is simply written into the accelerator block and back out. A communication/computation ratio of R denotes that the system spends R times as long computing inside the accelerator as the communication path on moving data and control information around the system. For instance, with $R = 1$, each accelerator will spend the same time computing as the whole system does communicating. Table 7.3 shows the latency for transfer through the whole data/control path at $R = 0$ for different block sizes at zero computation latency. So for example when we perform the experiment using a 1KB block size at $R = 2$, the latency of the accelerator block will be set to 2814 cycles.

[1]Vivado High-Level Synthesis- `https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html`, accessed April 2019.

Table 7.3: Latency for data and control path through distributed accelerator with zero computation latency.

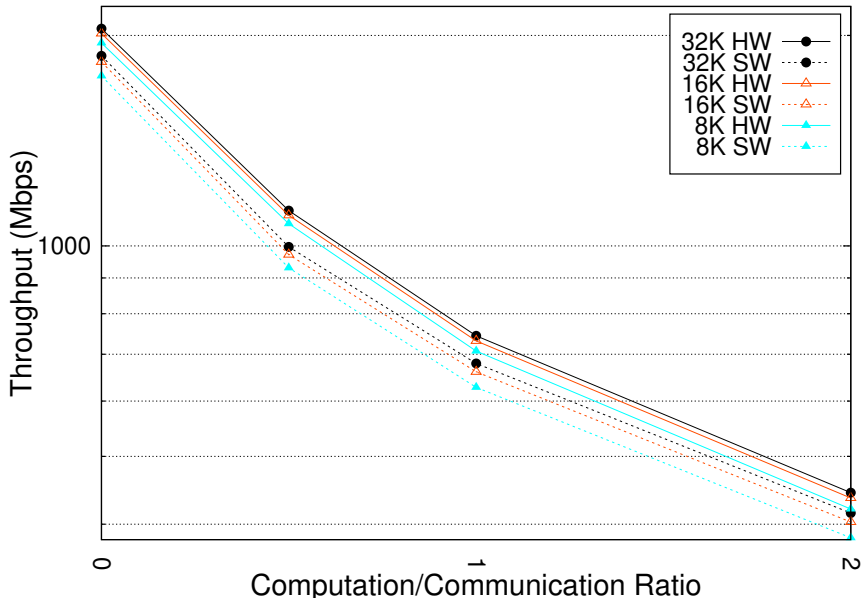
Data block size	**Latency of operation (μs)**	**Latency of operation (cycles at 156.25MHz)**
512B	6.33	989
1KB	9.01	1407
2KB	14.5	2268
4KB	25.5	3984
8KB	475	7422
16KB	911	14231
32KB	1787	27925

7.2.3 Results

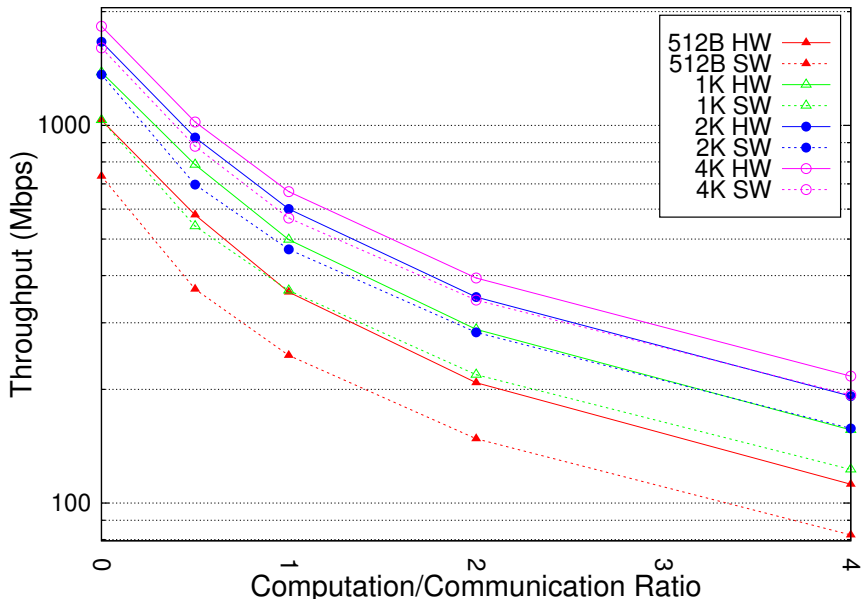
Figure 7.5 shows throughput results for the experiment setup discussed above. Here it is important to note that throughput is not in reference to the communication throughput over the links or the raw computing power in FLOPS. It is a measure of the throughput at which we are able to process data, when we have a distributed acceleration resource performing computation across two FPGAs on dependant data. In these results we see how the achievable throughput diminishes with decreasing communication block sizes and with increasing computational latency. Figure 7.6 shows the average latency for a single block to be processed through the two accelerators and the network.

7.2.3.1 Block Size

As one would expect, the larger the block size for computation the higher the sustainable throughput, as more data is transferred for every set of notification/ACKs/control messages which must be sent. We are able to achieve a gain in the maximum achievable throughput of 8.6% over the software transport solution. However, there appears to be a saturation point towards the upper limit of the block sizes, suggesting that further increases in the maximum block size for a single accelerator module will not lead to much higher performance. The current solution is not optimized in terms of overlapping computation and communication. After the accelerator has performed computation and initializes the RDMA to the remote accelerator, it must wait for completion notification of the RDMA before beginning



(a) Large block sizes.



(b) Small block sizes.

Figure 7.5: Data processing throughput for network-bound compute.

to write out new data to the local intermediate buffer. This could be solved by means of double buffering so to overlap the RDMA transfer and computation more efficiently.

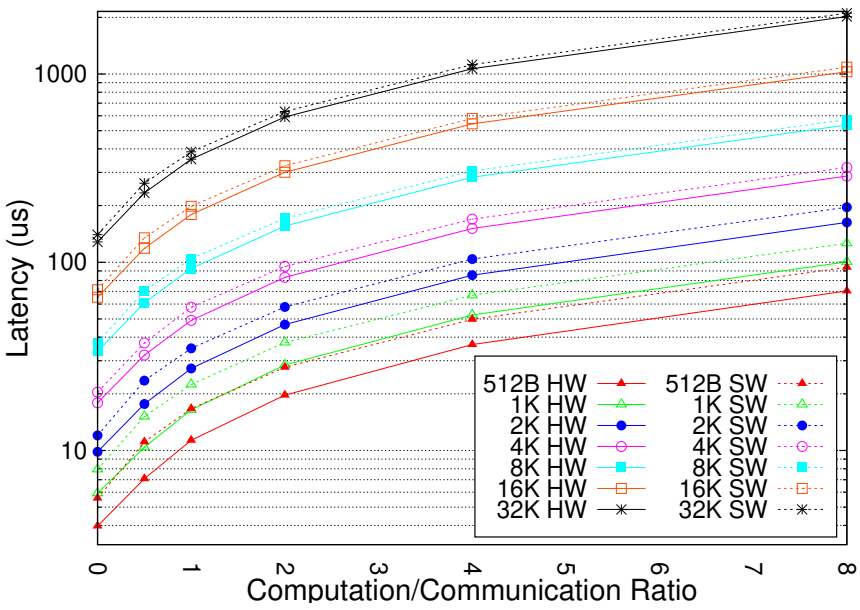


Figure 7.6: Average latency for a single block of data to be processed through the distributed accelerator network with two accelerators used.

7.2.3.2 Hardware Offloading

The results clearly show the benefit of offloading the transport layer into hardware, reducing the complexity of the control messages and delays awaiting notification. We see a higher throughput is achievable, as well as much improved latency for small and medium block sizes (see Figure 7.6). This reduced latency could have dramatic effects on applications which exchange small data blocks or control messages with irregular parallelism and access patterns between distributed resources. These sorts of access patterns can be seen in workloads which involve large memory-resident data structures involving pointer-chasing, such as lists, trees and graphs [56]. The interest in these workloads on FPGA has been growing with the emergence of shared memory systems with tightly-coupled memory between CPU and accelerator. Our NI extends this paradigm out and provides the accelerator with a method of performing shared memory operations directly on remote nodes. We see that for smaller data transfers we can reduce the latency of communication by as much as ≈29%. These results show that enabling the accelerator to reliably issue shared memory operations directly to remote resources may be highly beneficial to workloads exhibiting these irregular memory access patterns over distributed resources.

7.2.3.3 Link Saturation

As discussed above, the achievable throughput for a single processing element could be enhanced by overlapping between communication and computation using additional buffering to pre-fetch data for future computation. Another way of extracting additional memory bandwidth to enable us to saturate the 10G links between FPGA devices would be to use additional accelerator blocks. Typical HPC workloads which can benefit from FPGA acceleration will be able to take advantage of the spatial parallelism within the FPGA [11], [209], as well as finer grained parallelism. Having many blocks performing the same function in parallel should enable saturation of the links in this scenario, which we discuss in the following section.

7.3 Estimating Peak Computing Throughput

The data-processing throughput results we have presented in Section 7.2.3 can be used as a tool to estimate the achievable FLOPS in our system for network-bound compute. This can be done for any arbitrary synthesized accelerator block, and simply requires the latency and utilization of a given block. This knowledge should give an application designer a good idea of the limitations of their design if they are aware of the memory access patterns when distributing workloads over multiple accelerators. We use methods which are similar to those presented in [210] and [207].

As a basic example to show how this estimation can be found we have generated a simple accelerator IP block using the Vivado HLS tools. The functionality of the block is to perform a simple double-precision matrix-matrix multiplication, and we have used a modified version of the standard example code provided by Xilinx[2]. This block is very simple and the only additions we have made is to include `#pragma` directives for loop unrolling and pipelining of the solution, and to modify the interfaces to the block. We have altered the matrix data inputs to the block to use AXI interfaces, to ensure that the resource consumption is made realistic by

[2]The example code can be found in version 2018.2 at: `<install location>/Xilinx/Vivado/2018.2/examples /design/linear_algebra/matrix_multiply_alt`

maintaining compatibility with our networking stack.

By giving the data processing throughput results in terms of the block size for data sent to the accelerator, we enable design space exploration in this regard. Using different granularity of accelerators will enable either higher spatial parallelism, or greater processing power within a single processing element. Knowing the limitations of the network-bound FLOPS can aid in achieving an optimal design.

In this IP we feed 1KB blocks of data in for processing, meaning we will take values from the 1KB entry on the graph (Figure 7.5b). This means that the block receives 128 double precision (64-bit) floats to perform an 8×8 matrix-matrix multiplication, with 64 entries in each matrix. We use this operation as it is typical in many HPC applications and the output could be quickly synthesized from example code produced by Xilinx. This operation and matrix size gives us 1024 FLOPs per IP block, made up of 512 multiply-add operations.

The output of the Vivado HLS tools indicate that the block has a latency of 288 cycles. Looking at our results, given a zero computation/communication ratio latency of 1,407 cycles for a 1K transfer (see Table 7.3) this gives a data processing throughput of $\approx$1.1Gb/s when we extrapolate the computation/communication ratio out to $\approx$0.20 (288 cycles).

This means that we can make approximately 134,277 block transfers per-second. (1.1×10^9/8,192 bits per block transfer.) Using this we see that we can extract approximately 137 MFLOPS per IP block (1,024 FLOP/block transfer). According to the HLS synthesis output (shown in Table 7.4) each block requires 30194 Flip-Flops, with the implementation being Flip-Flop (FF) bound on the ZCU102 device. (Note that the results take into account the networking stack implementation. We have added this row to the table.)

Taking into account the fact that the entire networking stack when implemented on the FPGA requires around 10.7% of the FF resources (Table 7.1), given our HLS block's utilization we could theoretically fit 16 IP blocks on a single FPGA ((548160 total FFs*0.893)/30194 FFs per-block). However, 16 blocks would exceed the single 10G link bandwidth (although aggregating the link bandwidth within our solution would allow for even higher performance) and full utilization of the FPGA is simply impossible for place and route tools to achieve. Given these facts we will allow

Table 7.4: Vivado HLS Synthesis Report for device utilization on the ZCU102, with highlighted values taking into account available area once the networking stack is also implemented.

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	20	-
FIFO	-	-	-	-	-
Instance	12	56	21687	8838	-
Memory	36	-	0	0	-
Multiplexer	-	-	-	2864	-
Register	-	-	8507	0	-
Total	48	56	30194	11722	0
Available	1824	2520	548160	274080	0
Utilization (%)	2	2	5	4	0
Available w/ Networking stack	1543	2520	489426	227165	0
Utilization w/ Networking stack (%)	3.11	2	6.17	5.16	0

for 9 blocks transmitting over a single link. This is a conservative estimate as the switch can scale to many more ports, and multiple FPGAs can feasibly be accessed simultaneously. This would allow for a much larger aggregated bandwidth.

These calculations give a total expected memory-bound computing power in this instance of 1.233 GFLOPS (double precision) per FPGA (9 Blocks $\times$ 137 MFLOPS). These results are completely bound by the network, rather than the much larger bandwidth to main memory. Comparing these results against recent literature [11] calculating the network-bound compute, they give a theoretical peak of 8.9 GFLOPS over 8 FPGAs (1.11 GFLOPS per FPGA) for their solution. Given these results we see that our communication solution is very effective for the achievable computing power for network bound communications. This is particularly true given that in [11] they are limited to a basic ring topology, given the simple point-to-point network they offer. This creates a completely non-scalable solution.

As well as this, it is worth noting that the IP block created is very crude, with only simple optimizations; it has a very low usage of the DSP logic slices and a very high usage of Flip-Flops as opposed to BRAMs. It is likely that higher spatial locality could be achieved with suitable optimizations. This work is simply shown as

an example of how readily we can estimate the computational capacity for applications constrained by the network, and how easily the standard output of Vivado's HLS tools can be mapped onto our networking hardware and provide a model for dataflow type workloads.

7.4 Concluding Remarks

This Chapter has shown the ability of our system to support direct and simple communications between distributed FPGA resources, providing an effective model for dataflow style processing. Our experiments have shown the efficiencies that can be gained when using our hardware-offloaded transport as opposed to a software based mechanism. We have also shown that the area footprint on the FPGA is sufficiently low to accommodate a reasonable number of real-world accelerator blocks. The main contributions of this Chapter are as follows:

- Demonstration of the system supporting direct communication and data movement between distributed accelerator resources.
- An evaluation of the reduced complexity in control and dataflow through the system using our hardware-offloaded transport, as opposed to using a software based mechanism.
- An analysis which shows the real world potential of the system and shows that our results are comparable with those of other works in the literature, whilst providing higher capabilities with regards to the network.

Chapter 8

Conclusions and Future Work

Until the previous decade system architects had been able to rely on scaling transistor technology and increasing clock speeds for increasing performance. Since the breakdown of Dennard scaling and *scale out* has become the new paradigm, the interconnect has become a serious bottleneck in the performance of HPC systems. We argue that the FPGA is now well placed to penetrate into the domain of HPC, owing to the strict power requirements and flexibility that the FPGA can afford. However, as we have identified in this work, placing the FPGA within the context of current architectures will not work. Major architectural advancements are required; in terms of both the role of the accelerator within the system, and the interconnection between accelerator, network, and traditional compute elements.

8.1 Conclusions

In this thesis we have addressed some of the failings/limitations of existing interconnect technologies to adequately provide for the requirements for future FPGA based HPC systems, which are quickly becoming a reality. Interest in these systems is now burgeoning not only because of the energy characteristics of these devices, but also due to the evolution of data-intensive workloads more suitable for FPGA execution. We have identified that the two most important requirements for the interconnect are in **(i)** enabling direct communication between the fabric of distributed FPGA resources, and **(ii)** maintaining tight coupling between system memory and the FPGA (both local and remote).

With respect to the first point: we have shown that common solutions in which the FPGA is dependent on the CPU for network communication are a fundamental impediment to achieving high performance in distributed FPGA based systems. Decoupling these two resources from one another and allowing the FPGA to act as a full peer within the network is essential for low overhead communications and for enabling dataflow style processing over multiple FPGAs.

However, addressing the second requirement: typical solutions which provide this standalone networking capability limit the ability of the system to take advantage of tight system memory coupling. There are a number of workloads which benefit from tight coupling between system memory and accelerator, such as those exhibiting irregular memory accesses; graph traversal and the like. Keeping this tight coupling over a distributed system is key for performance when scaling beyond single host-accelerator algorithms.

As such we have developed a Network Interface solution which is able to achieve both of these goals by utilizing a fully hardware offloaded, reliable transport layer which eliminates the scalability issues of traditional connection based mechanisms. Scalability is afforded to the hardware due to the fact that the transport layer requires only a small volume of transient data to be stored within the Network Interface, which tracks in-flight transactions within the network. This is opposed to the work of prior hardware-offloaded transport technologies which use traditional connection based approaches; requiring non-scalable storage of connection state information or sacrificing core network functionality. The scalability of our solution is therefore not bound by the number of concurrent communicators, but but the number of possible in-flight transactions issued to the network from a given source node.

The Network Interface provides hardware primitives to support two distinct communication paradigms; RDMA and shared memory operations. As we detail in the thesis, this is done because like many others we see that future application scaling and efficient accelerator exploitation are dependent on new models of communication which MPI alone cannot support. Properly supporting one-sided communication and PGAS shared-memory operations in the hardware is key to enabling the breakdown of the host/accelerator copy-in, process, copy-out model of

computing, and for enabling algorithm scaling beyond the bounds of a single accelerator. Providing a simple method of allowing the accelerator to use the network independently of the CPU is one of the key achievements within this thesis.

This work demonstrates how the capabilities of our interconnect solution go beyond currently available technology in creating a solution specifically designed for FPGA based High Performance Computing. Existing solutions are either limited in terms of network capability of the FPGAs, constrained by the fact they are dependent on the CPU for issuing transfers, or they are too heavy for implementation within the FPGA fabric. We show that our system provides a lightweight solution which sits in the fabric of the FPGA, and provides direct access over a distributed global shared-memory space for accelerator and CPU resources alike.

Our baseline system shows that allowing for a dedicated hardware path in the NI for shared-memory accesses can reduce the latency of small transfers by over 25%. The target for these shared memory operations being small control/synchronization messages (which are typically highly latency sensitive), with larger data transfer being left to the RDMA transport, and being more dependent on throughput. Our baseline solution shows effective throughput for these RDMA transfers, achieving up to 8.56Gb/s bandwidth for payload over 10G links. We have shown that performance enhancements to the architecture can further aid in reducing the latency of these RDMA transfers at the receiver by as much as 20%. Finally, we have presented an analysis of the performance of our system when using distributed FPGA resources, and have shown that the latency and throughput characteristics of our system can outperform an equivalent software-based transport layer by 8.6% and 29% respectively.

As well as the presentation of the performance aspects of our system, we provide several analyses for our interconnect. We choose an end-to-end reliability mechanism for our system. This is because an investigation into the benefits of providing link-level error checks or forward error correction show that under normal operating conditions the impact of these techniques on performance is too great. We also provide an analysis of the consistency model and network errors which can occur within our system. Despite the complications we see given that our interconnect must allow for out-of-order packet reception our interconnect is able to

maintain consistency under the majority of normal operating circumstances, and we have addressed those in which we cannot by providing appropriate solutions.

8.2 Future Work

This thesis has tackled some of the main problems in providing an adequate interconnect solution for FPGA based HPC. However, there remains a number of critical elements to be developed, as well as several possible extensions to the architecture which could have a dramatic effect on the performance of the system.

8.2.1 Global Virtual Addressing

As we have discussed in Section 3.2.1, one of the key components which is required is a translation mechanism to support a global, virtual address space. In our particular platform this means configuration of the IOMMU on the Zynq Ultrascale+ (Xilinx name this the SMMU in their documentation). This will allow transactions which are received from the NI to be translated into physical addresses as they access the main memory or cache-coherent interconnect of the processing system.

Unfortunately, this alone is insufficient for all of the requirements of the system. The SMMU is only designed to give master devices within the Programmable Logic the use of virtual addresses. The interface between the Programmable Logic and the Processing System has a physical address at the interface when a transaction is produced from the PS side (PS master). While this is sufficient for virtualized RDMA (as the data is first read out locally so uses a slave interface to the PL), this causes issues with respect to the shared-memory path through the network. In this instance a page table will be required in the programmable logic which can convert physical addresses which are routed out, into global virtual addresses. These can then be passed through the network and into the IOMMU or the remote acceleration resources etc. at the remote side.

In the case where remote physical resources on the FPGA are being targeted rather than a remote DRAM access, one might wonder how translation works in this case. If the hardware requires support for virtual addressing then another stage

of translation will be required in the receiver. If not then merely the upper bits of the virtual address (corresponding to the node ID) can be stripped off, and having a one-to-one mapping between global virtual addresses and physical addresses in the lower bits.

8.2.2 Virtualization of Transport Layer

Another issue we have identified (discussed in Section 5.2.3.1) is that the current prototype is limited in the number of processes which can access the hardware simultaneously. Currently the notification mechanism for the transport layer uses dedicated queues (at separate physical addresses) in order to service notifications. In a production system this would require a method of forming logical queues at the same physical interface in order to provide notifications to a larger number of processes. This would be relatively simple to provide and could be done in a number of ways.

The first would be to assign each process an ID which relates to the queue they are provided for notifications, and tokens to dictate individual transfers. In this manner the CPU or accelerator could issue multiple transfers and simply poll the individual queues for notifications (the queue being accessed by an address offset in the map for the interface). The precise notification would then be handled internally by the process performing the access.

The second method would be to store all process and transfer IDs within the NI, and handle requests to the notification block individually, using a more complex structure within the NI to sort the notifications. In either scenario the effects on the scalability of the system will be negligible as the number of processes in each node will be limited, given we target HPC applications.

8.2.3 Atomic Operations at the System Level

As we discussed in Section 4.1.4, a mechanism is required in the receiver in order to facilitate hardware exclusive accesses. Using the load/store exclusive instructions within the CPU locks registers according to the process and transaction ID, which is insufficient for use over a distributed system. Since multiple nodes

will be capable of issuing the same process+transaction IDs we must take into account the source node and block any accesses within the NI to previously locked locations. It is currently unclear as to the impact of implementing this mechanism within the NI, as this will depend on the extent of the support and any additional performance enhancing features which may be desired.

This solution of locking will suffer performance issues as it requires that the data be read back to the issuing node over the network, and then written. The data must be pulled back toward the master which issued it. Obviously a more performant solution would send a full atomic operation as a single request, bundling the data we wish to update the exclusive register with. In this way it can be locked, written and unlocked in the same transfer. Adding support for this can come from one of two places.

The first is to implement this within the Network Interface, providing a dedicated messaging unit which will build full atomic transactions. We must do this because the alternative is to use a different system bus interface which provides this capability. Our solution uses AXI4, encapsulating and extending the protocol to make it suitable for larger off-chip networking. In the recent AMBA AXI5 specification atomic operations are provided as part of the interface standard. If this protocol were available then no additional hardware would be required at the sender (other than adjusting the encapsulation mechanism). The additional hardware already required at the receiver for handling exclusive accesses with multiple AXI domains could be extended to take into account these operations also.

8.2.4 Extension of Transport Mechanism Scalability

Currently the number of concurrent in-flight transactions which are serviced by the RDMA engine is limited to 16, as we use a simple implementation for the table which was used to match the CDMA engine prior to the inclusion of the *early acknowledgements* mechanism. Extending this out will benefit a full scale system greatly by preventing stalling in DMA transaction issuing over long network paths. As well as this, we do not envisage that doing this will have a great impact on the area overheads for the system. This is because the amount of data stored for each transaction and its associated timer is small. An additional latency penalty will be

incurred in this instance however, as checking the returning packets will need to be performed in a sequential manner. Extending the number of concurrent match lookups beyond 16 *will* negatively affect the area overheads, as the CAM's area scales N^2 with the number of parallel checks it must perform. It could also possibly cause problems in allowing the implementation to reach timing constraints during place and route. This is owing to the long combinatorial circuit which is required of the matching logic.

8.2.5 Hardware Offloading for Collective Operations

Another area which could see huge performance gains over standardized implementations is in offloading collective operations into the hardware. In order for MPI implementations to maintain portability the vast majority use default software implementations to realize collective operations. Typically these collectives are built up of a series of point-to-point messages, and so are very inefficient in terms of latency, bandwidth consumption and software intervention (2-sided calls, OS intervention etc.). Offloading the capability into hardware in order to reduce either the number of messages through the network or the number of steps in the algorithm for collectives could have a dramatic effect on the network load or the latency of collectives, and as such the performance of the system.

A traditional software implementation such as MPICH may use several point-to-point communication algorithms in order to optimize collective messages. This provides a trade-off between reducing the number of messages in the network and reducing the time it takes to transfer all the data. Actually duplicating these messages in the network and forming true multicast trees to send to multiple destinations simultaneously could have a large effect on the performance of these operations and the amount of data in the network. However, our switch and Network Interface currently have no capability for generating multicast packets.

Given that our intended topology uses a geographic addressing scheme, it should be feasible to work out possible multicast trees within the network by examining the list of destinations. Other methods could be sought which take advantage of the use of a hierarchical topology. Wildcard addresses could be used to send a single

message to a switch, which then performs a broadcast to lower tiers of the network. In this manner the upper layers of the topology will carry a single message, and then lower tiers with smaller latency communications could invoke more traditional algorithms for collectives. However, it is clear that this functionality would be complex to implement within the system, and is a significant research task in and of itself.

8.2.6 Library/Framework Integration

Obviously one of the main constraints with regards to the level of experimentation open to us is the lack of a distributed runtime system and lack of proper user-level library support. Our simple API provides basic user-level functionality to control and test the Network Interface and provide simple notifications. However, extending the hardware capabilities and extending our low-level API to provide more of a direct mapping to MPI functions or to the Portals API will certainly be required. This will only provide a port for the communication and synchronization primitives however, and even more work will be required for process initialization across the nodes for example, or setting up a global virtual address mapping for PGAS. This is all work which is far beyond the scope of this particular thesis...

8.3 Final Thoughts

There are many in the High Performance Computing community who have serious doubts about the viability of reconfigurable computing within this domain. It is certainly true that there are many impediments to the uptake of FPGA technology and many research questions still to be answered; such as standard HPC library support, portability, and reliability at scale. However, it is the hope of the author that we have provided a sufficient argument to show that there is much scope for the use of FPGAs within HPC.

While the body of work within this thesis has tackled a small piece of the puzzle– regarding the requirements of future interconnect technologies to enable the efficient exploitation of reconfigurable accelerators– we believe an inflexion point is

being reached. We see that many other important areas are reaching sufficient maturity to push the use of FPGAs into mainstream heterogeneous HPC systems...

High level synthesis techniques have progressed enormously, as has the architecture of FPGAs themselves. We are seeing ever denser SoCs, enhanced floating-point capabilities, hardened high-speed transceiver technology and more complex memory systems. Coupling this with growing system power constraints and burgeoning data-intensive workloads, we see that demand is also growing. It is our view that as the pressures on Moore's Law become greater, so will the pressure to seek alternative technologies.

Bibliography

[1] D. J. Frank, R. H. Dennard, E. Nowak, P. M. Solomon, Y. Taur, and H.-S. P. Wong, "Device scaling limits of si mosfets and their application dependencies," *Proceedings of the IEEE*, vol. 89, no. 3, pp. 259–288, 2001.

[2] A. Sodani and C Processor, "Race to exascale: opportunities and challenges," in *Keynote at the Annual IEEE/ACM 44th Annual International Symposium on Microarchitecture*, 2011.

[3] A. J. Hey, S. Tansley, K. M. Tolle, *et al.*, *The fourth paradigm: data-intensive scientific discovery*. Microsoft research Redmond, WA, 2009, vol. 1.

[4] A. Tate, A. Kamil, A. Dubey, A. Größlinger, B. Chamberlain, B. Goglin, C. Edwards, C. J. Newburn, D. Padua, D. Unat, *et al.*, "Programming abstractions for data locality," PADAL Workshop 2014, April 28–29, Swiss National Supercomputing Center, 2014, p. 7.

[5] S. Mittal and J. S. Vetter, "A survey of methods for analyzing and improving gpu energy efficiency," *ACM Computing Surveys (CSUR)*, vol. 47, no. 2, p. 19, 2015.

[6] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim, *et al.*, "A cloud-scale acceleration architecture," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, IEEE Press, 2016, p. 7.

[7] V. Milutinović, J. Salom, N Trifunović, and R. Giorgi, *Guide to dataflow supercomputing, Basic Concepts, Case Studies, and a Detailed Example.* Springer, 2015.

[8] K. D. Underwood, K. S. Hemmert, and C. D. Ulmer, "From silicon to science: the long road to production reconfigurable supercomputing," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 2, no. 4, p. 26, 2009.

[9] F. A. Escobar, X. Chang, and C. Valderrama, "Suitability analysis of fpgas for heterogeneous platforms in hpc," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 2, pp. 600–612, 2016.

[10] J. Cong, Z. Fang, M. Lo, H. Wang, J. Xu, and S. Zhang, "Understanding performance differences of fpgas and gpus," in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, IEEE, 2018, pp. 93–96.

[11] R. S. Correa and J. P. David, "Ultra-low latency communication channels for fpga-based hpc cluster," *Integration, the VLSI Journal*, vol. 63, pp. 41–55, 2018.

[12] A. T. Markettos, P. J. Fox, S. W. Moore, and A. W. Moore, "Interconnect for commodity fpga clusters: standardized or customized?" In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, IEEE, 2014, pp. 1–8.

[13] R. Thakur, P. Balaji, D. Buntinas, D. Goodell, W. Gropp, T. Hoefler, S. Kumar, E. Lusk, and J. L. Träff, "Mpi at exascale," *Procceedings of SciDAC*, vol. 2, pp. 14–35, 2010.

[14] J. Dinan, P. Balaji, E. Lusk, P Sadayappan, and R. Thakur, "Hybrid parallel programming with mpi and unified parallel c," in *Proceedings of the 7th ACM international conference on Computing frontiers*, ACM, 2010, pp. 177–186.

[15] M. G. Katevenis, "Interprocessor communication seen as load-store instruction generalization," in *In The Future of Computing, essays in memory of Stamatis Vassiliadis, K. Bertels ea (Eds.), Delft, The Netherlands*, Citeseer, 2007.

[16] *Axi central direct memory access, logicore ip product guide*, PG034, v4.1, Xilinx Inc., Apr. 2018.

[17] *Aurora 64b/66b*, PG074, v10.0, Xilinx Inc., Apr. 2015.

[18] J. Impagliazzo and J. A. Lee, *History of Computing in Education: IFIP 18th World Computer Congress, TC3/TC9 1st Conference on the History of Computing in Education, 22-27 August 2004, Toulouse, France*. Springer, 2004.

[19] R. M. Russell, "The cray-1 computer system," *Communications of the ACM*, vol. 21, no. 1, pp. 63–72, 1978.

[20] H. Falk, "What went wrong v: reaching for a gigaflop: the fate of the famed illiac iv was shaped by both research brilliance and real-world disasters," *IEEE spectrum*, vol. 13, no. 10, pp. 65–70, 1976.

[21] A. Marowka, "Back to thin-core massively parallel processors," *Computer*, vol. 44, no. 12, pp. 49–54, 2011.

[22] W. D. Hillis and L. W. Tucker, "The cm-5 connection machine: a scalable supercomputer," *Communications of the ACM*, vol. 36, no. 11, pp. 30–41, 1993.

[23] R. Esser and R. Knecht, "Intel paragon xp/s-architecture and software environment," in *Supercomputer'93*, Springer, 1993, pp. 121–141.

[24] S. L. Scott and G. M. Thorson, "The cray t3e network: adaptive routing in a high performance 3d torus," in *HOT Interconnects IV*, Citeseer, 1996.

[25] M. S. Warren, J. K. Salmon, D. J. Becker, M. P. Goda, T. Sterling, and W Winckelmans, "Pentium pro inside: i. a treecode at 430 gigaflops on asci red, ii. price/performance of $50/mflop on loki and hyglac," in *SC'97: Proceedings of the 1997 ACM/IEEE Conference on Supercomputing*, IEEE, 1997, pp. 61–61.

[26] D. J. Becker, T. Sterling, D. Savarese, J. E. Dorband, U. A. Ranawak, and C. V. Packer, "Beowulf: a parallel workstation for scientific computation," in *Proceedings, International Conference on Parallel Processing*, vol. 95, 1995, pp. 11–14.

[27] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *2011 38th Annual international symposium on computer architecture (ISCA)*, IEEE, 2011, pp. 365–376.

[28] K. M. Bresniker, S. Singhal, and R. S. Williams, "Adapting to thrive in a new economy of memory abundance," *Computer*, vol. 48, no. 12, pp. 44–53, 2015.

[29] K. J. Barker, K. Davis, A. Hoisie, D. J. Kerbyson, M. Lang, S. Pakin, and J. C. Sancho, "Entering the petaflop era: the architecture and performance of roadrunner," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, IEEE Press, 2008, p. 1.

[30] N. R. Adiga, G. Almási, G. S. Almasi, Y Aridor, R. Barik, D Beece, R Bellofatto, G Bhanot, R Bickford, M Blumrich, *et al.*, "An overview of the bluegene/l supercomputer," in *SC'02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, IEEE, 2002, pp. 60–60.

[31] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, *et al.*, "Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu," *ACM SIGARCH computer architecture news*, vol. 38, no. 3, pp. 451–460, 2010.

[32] C. Concatto, J. A. Pascual, J. Navaridas, J. Lant, A. Attwood, M. Lujan, and J. Goodacre, "A cam-free exascalable hpc router for low-energy communications," in *International Conference on Architecture of Computing Systems*, Springer, 2018, pp. 99–111.

[33] K. O'brien, I. Pietri, R. Reddy, A. Lastovetsky, and R. Sakellariou, "A survey of power and energy predictive models in hpc systems and applications," *ACM Computing Surveys (CSUR)*, vol. 50, no. 3, p. 37, 2017.

[34] D. B. Thomas, L. Howes, and W. Luk, "A comparison of cpus, gpus, fpgas, and massively parallel processor arrays for random number generation," in *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, ACM, 2009, pp. 63–72.

[35] H. Lange, F. Stock, A. Koch, and D. Hildenbrand, "Acceleration and energy efficiency of a geometric algebra computation using reconfigurable computers and gpus," in *2009 17th IEEE Symposium on Field Programmable Custom Computing Machines*, IEEE, 2009, pp. 255–258.

[36] T. Hamada, K. Benkrid, K. Nitadori, and M. Taiji, "A comparative study on asic, fpgas, gpus and general purpose processors in the o (n$\hat{2}$) gravitational n-body simulation," in *2009 NASA/ESA Conference on Adaptive Hardware and Systems*, IEEE, 2009, pp. 447–452.

[37] S. Kestur, J. D. Davis, and O. Williams, "Blas comparison on fpga, cpu and gpu," in *2010 IEEE computer society annual symposium on VLSI*, IEEE, 2010, pp. 288–293.

[38] B. Betkaoui, D. B. Thomas, and W. Luk, "Comparing performance and energy efficiency of fpgas and gpus for high productivity computing," in *2010 International Conference on Field-Programmable Technology*, IEEE, 2010, pp. 94–101.

[39] H. M. Hussain, K. Benkrid, A. T. Erdogan, and H. Seker, "Highly parameterized k-means clustering on fpgas: comparative results with gpps and gpus," in *2011 International Conference on Reconfigurable Computing and FPGAs*, IEEE, 2011, pp. 475–480.

[40] C. De Schryver, I. Shcherbakov, F. Kienle, N. Wehn, H. Marxen, A. Kostiuk, and R. Korn, "An energy efficient fpga accelerator for monte carlo option pricing with the heston model," in *2011 International Conference on Reconfigurable Computing and FPGAs*, IEEE, 2011, pp. 468–474.

[41] B. Duan, W. Wang, X. Li, C. Zhang, P. Zhang, and N. Sun, "Floating-point mixed-radix fft core generation for fpga and comparison with gpu and cpu," in *2011 International Conference on Field-Programmable Technology*, IEEE, 2011, pp. 1–6.

[42] B. Van Essen, C. Macaraeg, M. Gokhale, and R. Prenger, "Accelerating a random forest classifier: multi-core, gp-gpu, or fpga?" In *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, IEEE, 2012, pp. 232–239.

[43] M. Birk, M. Balzer, N. Ruiter, and J. Becker, "Comparison of processing performance and architectural efficiency metrics for fpgas and gpus in 3d ultrasound computer tomography," in *2012 International Conference on Reconfigurable Computing and FPGAs*, IEEE, 2012, pp. 1–7.

[44] D. Zou, Y. Dou, and F. Xia, "Optimization schemes and performance evaluation of smith–waterman algorithm on cpu, gpu and fpga," *Concurrency and Computation: Practice and Experience*, vol. 24, no. 14, pp. 1625–1644, 2012.

[45] K. Benkrid, A. Akoglu, C. Ling, Y. Song, Y. Liu, and X. Tian, "High performance biological pairwise sequence alignment: fpga versus gpu versus cell be versus gpp," *International Journal of Reconfigurable Computing*, vol. 2012, p. 7, 2012.

[46] K. Pauwels, M. Tomasi, J. D. Alonso, E. Ros, and M. M. Van Hulle, "A comparison of fpga and gpu for real-time phase-based optical flow, stereo, and local image features," *IEEE Transactions on Computers*, vol. 61, no. 7, pp. 999–1012, 2011.

[47] J. Fowers, G. Brown, J. Wernsing, and G. Stitt, "A performance and energy comparison of convolution on gpus, fpgas, and multicore processors," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 9, no. 4, p. 25, 2013.

[48] G. Singh, "Xilinx 16nm datacenter device family within-package hbm and ccix interconnect," in *2017 IEEE Hot Chips 29 Symposium (HCS)*, IEEE, 2017, pp. 1–22.

[49] M. Wissolik, D. Zacher, A. Torza, and B. Day, *Virtex ultrascale+ hbm fpga: a revolutionary increase in memory performance*, WP485, v1.1, Xilinx Inc., Jul. 2019.

[50] U. Sinha, "Enabling impactful dsp designs on fpgas with hardened floating-point implementation," *Altera White Paper, WP-01227-1.0 (Aug. 2014)*, 2014.

[51] J. Backus, "Acm turing award lectures," in, New York, NY, USA: ACM, 2007, ch. Can Programming Be Liberated from the Von Neumann Style?: A Functional Style and Its Algebra of Programs, ISBN: 978-1-4503-1049-9.

[52] M. Nüssle, H. Fröning, S. Kapferer, and U. Brüning, "Accelerate communication, not computation!" In *High-Performance Computing Using FPGAs*, Springer, 2013, pp. 507–542.

[53] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, *et al.*, "The landscape of parallel computing research: a view from berkeley," Technical Report UCB/EECS-2006-183, Tech. Rep., 2006.

[54] G. Wellein, G. Hager, T. Zeiser, M. Wittmann, and H. Fehske, "Efficient temporal blocking for stencil computations by multicore-aware wavefront parallelization," in *2009 33rd Annual IEEE International Computer Software and Applications Conference*, IEEE, vol. 1, 2009, pp. 579–586.

[55] A. Rafique, G. A. Constantinides, and N. Kapre, "Communication optimization of iterative sparse matrix-vector multiply on gpus and fpgas," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 1, pp. 24–34, 2015.

[56] G. Weisz, J. Melber, Y. Wang, K. Fleming, E. Nurvitadhi, and J. C. Hoe, "A study of pointer-chasing performance on shared-memory processor-fpga systems," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ACM, 2016, pp. 264–273.

[57] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, D. Firestone, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, *et al.*, "Configurable clouds," *IEEE Micro*, vol. 37, no. 3, pp. 52–61, 2017.

[58] D. Unnikrishnan, S. G. Virupaksha, L. Krishnan, L. Gao, and R. Tessier, "Accelerating iterative algorithms with asynchronous accumulative updates on fpgas," in *2013 International Conference on Field-Programmable Technology (FPT)*, IEEE, 2013, pp. 66–73.

[59] D. Unat, A. Dubey, T. Hoefler, J. Shalf, M. Abraham, M. Bianco, B. L. Chamberlain, R. Cledat, H. C. Edwards, H. Finkel, *et al.*, "Trends in data locality abstractions for hpc systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 10, pp. 3007–3020, 2017.

[60] T. Geng, T. Wang, A. Sanaullah, C. Yang, R Xuy, R. Patel, and M. Herbordt, "Fpdeep: acceleration and load balancing of cnn training on fpga clusters," in *Proc. IEEE Symp. on Field Programmable Custom Computing Machines*, 2018.

[61] E. Kadric, K. Mahajan, and A. DeHon, "Kung fu data energy-minimizing communication energy in fpga computations," in *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, IEEE, 2014, pp. 214–221.

[62] G. Tan, C. Zhang, W. Tang, P. Zhang, and N. Sun, "Accelerating irregular computation in massive short reads mapping on fpga co-processor," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 5, pp. 1253–1264, 2016, ISSN: 1045-9219.

[63] H. Fu and R. G. Clapp, "Eliminating the memory bottleneck: an fpga-based solution for 3d reverse time migration," in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, ACM, 2011, pp. 65–74.

[64] E. Nurvitadhi, G. Venkatesh, J. Sim, D. Marr, R. Huang, J. Ong Gee Hock, Y. T. Liew, K. Srivatsan, D. Moss, S. Subhaschandra, *et al.*, "Can fpgas beat gpus in accelerating next-generation deep neural networks?" In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ACM, 2017, pp. 5–14.

[65] R. Zhao, W. Song, W. Zhang, T. Xing, J.-H. Lin, M. Srivastava, R. Gupta, and Z. Zhang, "Accelerating binarized convolutional neural networks with software-programmable fpgas," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ACM, 2017, pp. 15–24.

[66] D. W. Walker and J. J. Dongarra, "Mpi: a standard message passing interface," *Supercomputer*, vol. 12, pp. 56–68, 1996.

[67] T. Hoefler, J. Dinan, R. Thakur, B. Barrett, P. Balaji, W. Gropp, and K. Underwood, "Remote memory access programming in mpi-3," *ACM Transactions on Parallel Computing*, vol. 2, no. 2, p. 9, 2015.

[68] S. Amarasinghe, D. Campbell, W. Carlson, A. Chien, W. Dally, E. Elnohazy, M. Hall, R. Harrison, W. Harrod, K. Hill, *et al.*, "Exascale software study: software challenges in extreme scale systems," *DARPA IPTO, Air Force Research Labs, Tech. Rep*, pp. 1–153, 2009.

[69] C. A. R. Hoare, "Communicating sequential processes," in *The origin of concurrent programming*, Springer, 1978, pp. 413–443.

[70] S. Potluri, K. Hamidouche, A. Venkatesh, D. Bureddy, and D. K. Panda, "Efficient inter-node mpi communication using gpudirect rdma for infiniband clusters with nvidia gpus," in *2013 42nd International Conference on Parallel Processing*, IEEE, 2013, pp. 80–89.

[71] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "Ompss: a proposal for programming heterogeneous multi-core architectures," *Parallel Processing Letters*, vol. 21, no. 02, pp. 173–193, 2011.

[72] A. Filgueras, E. Gil, D. Jimenez-Gonzalez, C. Alvarez, X. Martorell, J. Langer, J. Noguera, and K. Vissers, "Ompss@ zynq all-programmable soc ecosystem," in *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*, ACM, 2014, pp. 137–146.

[73] *Upc language specifications v1. 2*, UPC Consortium, 2005.

[74] R. W. Numrich and J. Reid, "Co-array fortran for parallel programming," in *ACM Sigplan Fortran Forum*, ACM, vol. 17, 1998, pp. 1–31.

[75] P. N. Hilfinger, D. Bonachea, K. Datta, D. Gay, S. Graham, A. Kamil, B. Liblit, G. Pike, J. Su, and K. Yelick, "Titanium language reference manual," *Computer Science*, 2006.

[76] V. Saraswat, B. Bloom, I. Peshansky, O. Tardieu, and D. Grove, "X10 language specification," *Specification, IBM, janvier*, 2012.

[77] B. L. Chamberlain, D. Callahan, and H. P. Zima, "Parallel programmability and the chapel language," *The International Journal of High Performance Computing Applications*, vol. 21, no. 3, pp. 291–312, 2007.

[78] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith, "Introducing openshmem: shmem for the pgas community," in *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, ACM, 2010, p. 2.

[79] M. De Wael, S. Marr, B. De Fraine, T. Van Cutsem, and W. De Meuter, "Partitioned global address space languages," *ACM Computing Surveys (CSUR)*, vol. 47, no. 4, p. 62, 2015.

[80] W. Gropp and M. Snir, "Programming for exascale computers," *Computing in Science & Engineering*, vol. 15, no. 6, pp. 27–35, 2013.

[81] S. Potluri, P. Lai, K. Tomko, S. Sur, Y. Cui, M. Tatineni, K. W. Schulz, W. L. Barth, A. Majumdar, and D. K. Panda, "Quantifying performance benefits of overlap using mpi-2 in a seismic modeling application," in *Proceedings of the 24th ACM International Conference on Supercomputing*, ACM, 2010, pp. 17–25.

[82] H. Shan, N. J. Wright, J. Shalf, K. Yelick, M. Wagner, and N. Wichmann, "A preliminary evaluation of the hardware acceleration of the cray gemini interconnect for pgas languages and comparison with mpi," *ACM SIGMETRICS Performance Evaluation Review*, vol. 40, no. 2, pp. 92–98, 2012.

[83] J. Jose, S. Potluri, H. Subramoni, X. Lu, K. Hamidouche, K. Schulz, H. Sundar, and D. K. Panda, "Designing scalable out-of-core sorting with hybrid mpi+ pgas programming models," in *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, ACM, 2014, p. 7.

[84] J. Jose, S. Potluri, K. Tomko, and D. K. Panda, "Designing scalable graph500 benchmark with hybrid mpi+ openshmem programming models," in *International Supercomputing Conference*, Springer, 2013, pp. 109–124.

[85] R. Nikhil, "Bluespec system verilog: efficient, correct rtl from high level specifications," in *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE'04.*, IEEE, 2004, pp. 69–70.

[86] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: constructing hardware in a scala embedded language," in *DAC Design Automation Conference 2012*, IEEE, 2012, pp. 1212–1221.

[87] S. Swan, "An introduction to system level modeling in systemc 2.0," *Cadence Design Systems, Inc., draft report*, 2001.

[88] *Vivado design suite user guide, high-level synthesis*, UG902, v2017.1, Xilinx Inc., Apr. 2017.

[89] *Intel high level synthesis compiler: reference manual*, MNL-1083, Intel, Sep. 2019.

[90] A. Papakonstantinou, K. Gururaj, J. A. Stratton, D. Chen, J. Cong, and W.-M. W. Hwu, "Fcuda: enabling efficient compilation of cuda kernels onto fpgas," in *2009 IEEE 7th Symposium on Application Specific Processors*, IEEE, 2009, pp. 35–42.

[91] A. Munshi, "The opencl specification," in *2009 IEEE Hot Chips 21 Symposium (HCS)*, IEEE, 2009, pp. 1–314.

[92] I. Mavroidis, I. Papaefstathiou, L. Lavagno, D. S. Nikolopoulos, D. Koch, J. Goodacre, I. Sourdis, V. Papaefstathiou, M. Coppola, and M. Palomino, "Ecoscale: reconfigurable computing and runtime system for future exascale systems," in *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, 2016, pp. 696–701.

[93] J. Liu, J. Wu, and D. K. Panda, "High performance rdma-based mpi implementation over infiniband," *International Journal of Parallel Programming*, vol. 32, no. 3, pp. 167–198, 2004.

[94] M. J. Rashti and A. Afsahi, "10-gigabit iwarp ethernet: comparative performance analysis with infiniband and myrinet-10g," in *2007 IEEE International Parallel and Distributed Processing Symposium*, IEEE, 2007, pp. 1–8.

[95] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye, and M. Lipshteyn, "Rdma over commodity ethernet at scale," in *Proceedings of the 2016 ACM SIGCOMM Conference*, ACM, 2016, pp. 202–215.

[96] D. Chen, N. A. Eisley, P. Heidelberger, R. M. Senger, Y. Sugawara, S. Kumar, V. Salapura, D. L. Satterfield, B. Steinmacher-Burow, and J. J. Parker, "The ibm blue gene/q interconnection network and message unit," in *SC'11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, IEEE, 2011, pp. 1–10.

[97] Y. Ajima, T. Inoue, S. Hiramoto, Y. Takagi, and T. Shimizu, "The tofu interconnect," *IEEE Micro*, vol. 32, no. 1, pp. 21–31, 2012.

[98] R. Alverson, D. Roweth, and L. Kaplan, "The gemini system interconnect," in *High Performance Interconnects (HOTI), 2010 IEEE 18th Annual Symposium on*, IEEE, 2010, pp. 83–87.

[99] R. Ammendola, A. Biagionil, O. Frezza, F. L. Cicero, A. Lonardo, P. S. Paolucci, D. Rossetti, F. Simula, L. Tosoratto, and P. Vicini, "Design and implementation of a modular, low latency, fault-aware, fpga-based network interface," in *2013 International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, IEEE, 2013, pp. 1–6.

[100] *Xilinx embedded target rdma enabled*, PG294, v1.0, Xilinx Inc., Apr. 2018.

[101] G. Kalokerinos, V. Papaefstathiou, G. Nikiforos, S. Kavadias, M. Katevenis, D. Pnevmatikatos, and X. Yang, "Fpga implementation of a configurable cache/scratchpad memory with virtualized user-level rdma capability," in *2009 International Symposium on Systems, Architectures, Modeling, and Simulation*, IEEE, 2009, pp. 149–156.

[102] B. Alverson, E. Froese, L. Kaplan, and D. Roweth, "Cray xc series network," *Cray Inc., White Paper WP-Aries01-1112*, 2012.

[103] R. H. Arpaci, D. E. Culler, A. Krishnamurthy, S. G. Steinberg, and K. Yelick, "Empirical evaluation of the cray-t3d: a compiler perspective," in *ACM SIGARCH Computer Architecture News*, ACM, vol. 23, 1995, pp. 320–331.

[104] H. Fröning and H. Litz, "Efficient hardware support for the partitioned global address space," in *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, IEEE, 2010, pp. 1–6.

[105] C. Ulmer, R. Hilles, and D. Thompson, "Reconfigurable computing aspects of the cray xd1," *Proceedings of the CUG 2005*, 2005.

[106] S. Shida, Y. Shibata, K. Oguri, and D. A. Buell, "Implementation of a barotropic operator for ocean model simulation using a reconfigurable machine," in *2007 International Conference on Field Programmable Logic and Applications*, IEEE, 2007, pp. 589–592.

[107] *Reconfigurable application-specific computing user's guide*, 007-4718-004, Silicon Graphics Inc., Mar. 2006.

[108] *Sgi numalink industry leading interconnect technology*, J14820, Silicon Graphics Inc., 2005.

[109] C. Kachris and D. Soudris, "A survey on reconfigurable accelerators for cloud computing," in *2016 26th International conference on field programmable logic and applications (FPL)*, IEEE, 2016, pp. 1–10.

[110] M. Showerman, J. Enos, A. Pant, V. Kindratenko, C. Steffen, R. Pennington, W.-m. Hwu, *et al.*, "Qp: a heterogeneous multi-accelerator cluster," in *Proc. 10th LCI International Conference on High-Performance Clustered Computing*, 2009.

[111] R. Baxter, S. Booth, M. Bull, G. Cawood, J. Perry, M. Parsons, A. Simpson, A. Trew, A. McCormick, G. Smart, *et al.*, "Maxwell-a 64 fpga supercomputer," in *Second NASA/ESA Conference on Adaptive Hardware and Systems (AHS 2007)*, IEEE, 2007, pp. 287–294.

[112] A. D. George, M. C. Herbordt, H. Lam, A. G. Lawande, J. Sheng, and C. Yang, "Novo-g#: large-scale reconfigurable computing with direct and programmable interconnects," in *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, IEEE, 2016, pp. 1–7.

[113] R. Sass, W. V. Kritikos, A. G. Schmidt, S. Beeravolu, and P. Beeraka, "Reconfigurable computing cluster (rcc) project: investigating the feasibility of fpga-based petascale computing," in *15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2007)*, IEEE, 2007, pp. 127–140.

[114] A. G. Schmidt, S. Datta, A. A. Mendon, and R. Sass, "Investigation into scaling i/o bound streaming applications productively with an all-fpga cluster," *Parallel Computing*, vol. 38, no. 8, pp. 344–364, 2012.

[115] L. Ling, N. Oliver, C. Bhushan, W. Qigang, A. Chen, S. Wenbo, Y. Zhihong, A. Sheiman, I. McCallum, J. Grecco, *et al.*, "High-performance, energy-efficient platforms using in-socket fpga accelerators," in *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, ACM, 2009, pp. 261–264.

[116] H. J. Yang, K. Fleming, M. Adler, and J. Emer, "Leap shared memories: automating the construction of fpga coherent memories," in *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, IEEE, 2014, pp. 117–124.

[117] *The convey hc-2 computer, architectural overview*, CONV-12-030.2, Convey Computer Corporation, 2012.

[118] D. Sidler, G. Alonso, M. Blott, K. Karras, K. Vissers, and R. Carley, "Scalable 10gbps tcp/ip stack architecture for reconfigurable hardware," in *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*, IEEE, 2015, pp. 36–43.

[119] J. Weerasinghe, R. Polig, F. Abel, and C. Hagleitner, "Network-attached fpgas for data center applications," in *2016 International Conference on Field-Programmable Technology (FPT)*, IEEE, 2016, pp. 36–43.

[120] J. Weerasinghe, F. Abel, C. Hagleitner, and A. Herkersdorf, "Enabling fpgas in hyperscale data centers," in *2015 IEEE 12th Intl Conf on Ubiquitous Intelligence and Computing and 2015 IEEE 12th Intl Conf on Autonomic and Trusted Computing and 2015 IEEE 15th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom)*, IEEE, 2015, pp. 1078–1086.

[121] S.-W. Jun, M. Liu, S. Xu, *et al.*, "A transport-layer network for distributed fpga platforms," in *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, IEEE, 2015, pp. 1–4.

[122] R. Balasubramonian, J. Chang, T. Manning, J. H. Moreno, R. Murphy, R. Nair, and S. Swanson, "Near-data processing: insights from a micro-46 workshop," *IEEE Micro*, vol. 34, no. 4, pp. 36–42, 2014.

[123] R. Nair, "Evolution of memory architecture," *Proceedings of the IEEE*, vol. 103, no. 8, pp. 1331–1345, 2015.

[124] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, *et al.*, "A reconfigurable fabric for accelerating large-scale datacenter services," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 13–24, 2014.

[125] G. Kestor, R. Gioiosa, D. J. Kerbyson, and A. Hoisie, "Quantifying the energy cost of data movement in scientific applications," in *2013 IEEE international symposium on workload characterization (IISWC)*, IEEE, 2013, pp. 56–65.

[126] R. Perlman, D Eastlake 3rd, D Dutt, S. Gai, and A. Ghanwani, "Routing bridges (rbridges): base protocol specification," Tech. Rep., 2011.

[127] J. Postel, "Dod standard transmission control protocol," *SIGCOMM Comput. Commun. Rev.*, vol. 10, no. 4, pp. 52–132, Oct. 1980, ISSN: 0146-4833.

[128] S. Seth and M. A. Venkatesulu, *TCP/IP architecture, design and implementation in Linux*. John Wiley & Sons, 2009, vol. 68.

[129] Intilop Corporation, *10 G bit TCP Offload Engine + PCIe/DMA SOC IP*. 2012.

[130] D. Freimuth, E. C. Hu, J. D. LaVoie, R. Mraz, E. M. Nahum, P. Pradhan, and J. M. Tracey, "Server network scalability and tcp offload.," in *USENIX Annual Technical Conference, General Track*, 2005, pp. 209–222.

[131] D. Sidler, Z. István, and G. Alonso, "Low-latency tcp/ip stack for data center applications," in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, IEEE, 2016, pp. 1–4.

[132] A. Ford, C. Raiciu, M. Handley, S. Barre, J. Iyengar, *et al.*, "Architectural guidelines for multipath tcp development," *IETF, Informational RFC*, vol. 6182, pp. 2070–1721, 2011.

[133] R. L. Graham, S.-E. Choi, D. J. Daniel, N. N. Desai, R. G. Minnich, C. E. Rasmussen, L. D. Risinger, and M. W. Sukalski, "A network-failure-tolerant message-passing system for terascale clusters," *International Journal of Parallel Programming*, vol. 31, no. 4, pp. 285–303, 2003.

[134] F. Petrini, W.-c. Feng, A. Hoisie, S. Coll, and E. Frachtenberg, "The quadrics network: high-performance clustering technology," *Ieee Micro*, vol. 22, no. 1, pp. 46–57, 2002.

[135] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su, "Myrinet: a gigabit-per-second local area network," *IEEE micro*, vol. 15, no. 1, pp. 29–36, 1995.

[136] M. J. Koop, S. Sur, Q. Gao, and D. K. Panda, "High performance mpi design using unreliable datagram for ultra-scale infiniband clusters," in *Proceedings of the 21st annual international conference on Supercomputing*, ACM, 2007, pp. 180–189.

[137] T. Ben-Nun and T. Hoefler, "Demystifying parallel and distributed deep learning: an in-depth concurrency analysis," *ACM Computing Surveys (CSUR)*, vol. 52, no. 4, p. 65, 2019.

[138] C. Yu, H. Tang, C. Renggli, S. Kassing, A. Singla, D. Alistarh, C. Zhang, and J. Liu, "Distributed learning over unreliable networks," in *International Conference on Machine Learning*, 2019, pp. 7202–7212.

[139] R. E. Grant, M. J. Rashti, P. Balaji, and A. Afsahi, "Scalable connectionless rdma over unreliable datagrams," *Parallel Computing*, vol. 48, pp. 15–39, 2015.

[140] S. Sur, L. Chai, H.-W. Jin, and D. K. Panda, "Shared receive queue based scalable mpi design for infiniband clusters," in *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*, IEEE, 2006, 10–pp.

[141] M. J. Koop, R. Kumar, and D. K. Panda, "Can software reliability outperform hardware reliability on high performance interconnects?: a case study with mpi over infiniband," in *Proceedings of the 22nd annual international conference on Supercomputing*, ACM, 2008, pp. 145–154.

[142] M. J. Koop, J. K. Sridhar, and D. K. Panda, "Scalable mpi design over infiniband using extended reliable connection," in *2008 IEEE International Conference on Cluster Computing*, IEEE, 2008, pp. 203–212.

[143] *Mellanox ib-verbs api (vapi)*, 2088AN, Mellanox Technologies, 2001.

[144] L. Oden, H. Fröning, and F.-J. Pfreundt, "Infiniband-verbs on gpu: a case study of controlling an infiniband network device from the gpu," in *2014 IEEE International Parallel & Distributed Processing Symposium Workshops*, IEEE, 2014, pp. 976–983.

[145] P. Geoffray and T. Hoefler, "Adaptive routing strategies for modern high performance networks," in *2008 16th IEEE Symposium on High Performance Interconnects*, IEEE, 2008, pp. 165–172.

[146] J. C. Martinez, J. Flich, A. Robles, P. Lopez, and J. Duato, "Supporting fully adaptive routing in infiniband networks," in *Proceedings International Parallel and Distributed Processing Symposium*, IEEE, 2003, 10–pp.

[147] X.-Y. Lin, Y.-C. Chung, and T.-Y. Huang, "A multiple lid routing scheme for fat-tree-based infiniband networks," in *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, IEEE, 2004, p. 11.

[148] A. Vishnu, M Koop, A. Moody, A. R. Mamidala, S. Narravula, and D. K. Panda, "Hot-spot avoidance with multi-pathing over infiniband: an mpi perspective," in *Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid'07)*, IEEE, 2007, pp. 479–486.

[149] J. Kim, W. J. Dally, S. Scott, and D. Abts, "Technology-driven, highly-scalable dragonfly topology," in *2008 International Symposium on Computer Architecture*, IEEE, 2008, pp. 77–88.

[150] R. Bittner, E. Ruf, and A. Forin, "Direct gpu/fpga communication via pci express," *Cluster Computing*, vol. 17, no. 2, pp. 339–348, 2014.

[151] M. S. Birrittella, M. Debbage, R. Huggahalli, J. Kunz, T. Lovett, T. Rimmer, K. D. Underwood, and R. C. Zak, "Intel® omni-path architecture: enabling scalable, high performance fabrics," in *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, IEEE, 2015, pp. 1–9.

[152] D. Chen, N. Eisley, P. Heidelberger, R. Senger, Y. Sugawara, S. Kumar, V. Salapura, D. Satterfield, B. Steinmacher-Burow, and J. Parker, "The ibm blue gene/q interconnection fabric," *IEEE Micro*, vol. 32, no. 1, pp. 32–43, 2012.

[153] S. Derradji, T. Palfer-Sollier, J.-P. Panziera, A. Poudes, and F. W. Atos, "The bxi interconnect architecture," in *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, IEEE, 2015, pp. 18–25.

[154] B. W. Barrett, R. Brightwell, S. Hemmert, K. Pedretti, K. Wheeler, K. Underwood, R. Riesen, A. B. Maccabe, and T. Hudson, "The portals 4.0 network programming interface," *Sandia National Laboratories, November 2012, Technical Report SAND2012-10087*, 2012.

[155] M. Besta and T. Hoefler, "Slim fly: a cost effective low-diameter network topology," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, IEEE Press, 2014, pp. 348–359.

[156] Y. Ajima, S. Sumimoto, and T. Shimizu, "Tofu: a 6d mesh/torus interconnect for exascale computers.," *IEEE Computer*, vol. 42, no. 11, pp. 36–40, 2009.

[157] Y. Ajima, T. Kawashima, T. Okamoto, N. Shida, K. Hirai, T. Shimizu, S. Hiramoto, Y. Ikeda, T. Yoshikawa, K. Uchida, *et al.*, "The tofu interconnect d," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, IEEE, 2018, pp. 646–654.

[158] T. Toyoshima, "Icc: an interconnect controller for the tofu interconnect architecture," in *Hot Chips*, vol. 22, 2010.

[159] S. W. Moore, P. J. Fox, S. J. Marsh, A. T. Markettos, and A. Mujumdar, "Bluehive-a field-programable custom computing machine for extreme-scale real-time neural network simulation," in *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, IEEE, 2012, pp. 133–140.

[160] M. Nüssle, B. Geib, H. Fröning, and U. Brüning, "An fpga-based custom high performance interconnection network," in *2009 International Conference on Reconfigurable Computing and FPGAs*, IEEE, 2009, pp. 113–118.

[161] N. Zilberman, Y. Audzevich, G. A. Covington, and A. W. Moore, "Netfpga sume: toward 100 gbps as research commodity," *IEEE micro*, vol. 34, no. 5, pp. 32–41, 2014.

[162] R. Ammendola, A. Biagioni, O. Frezza, F. L. Cicero, A. Lonardo, P. S. Paolucci, D. Rossetti, A. Salamon, G. Salina, F. Simula, *et al.*, "Apenet+: high bandwidth 3d torus direct network for petaflops scale commodity clusters," in *Journal of Physics: Conference Series*, IOP Publishing, vol. 331, 2011, p. 052 029.

[163] M. Katevenis, R. Ammendola, A. Biagioni, P. Cretaro, O. Frezza, F. L. Cicero, A. Lonardo, M. Martinelli, P. S. Paolucci, E. Pastorelli, *et al.*, "Next generation of exascale-class systems: exanest project and the status of its interconnect and storage development," *Microprocessors and Microsystems*, vol. 61, pp. 58–71, 2018.

[164] A. Rigo, C. Pinto, K. Pouget, D. Raho, D. Dutoit, P.-Y. Martinez, C. Doran, L. Benini, I. Mavroidis, M. Marazakis, *et al.*, "Paving the way towards a highly energy-efficient and highly integrated compute node for the exascale revolution: the exanode approach," in *2017 Euromicro Conference on Digital System Design (DSD)*, IEEE, 2017, pp. 486–493.

[165] M. Ashworth, G. D. Riley, A. Attwood, and J. Mawer, "First steps in porting the lfric weather and climate model to the fpgas of the euroexa architecture," *Scientific Programming*, vol. 2019, 2019.

[166] P. M. Kogge, P. La Fratta, and M. Vance, "[2010] facing the exascale energy wall," in *2010 International Workshop on Innovative Architecture for Future Generation High Performance*, IEEE, 2010, pp. 51–58.

[167] J. Kim, W. J. Dally, and D. Abts, "Flattened butterfly: a cost-efficient topology for high-radix networks," *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2, pp. 126–137, 2007.

[168] J. A. Pascual, J. Lant, A. Attwood, C. Concatto, J. Navaridas, M. Luján, and J. Goodacre, "Designing an exascale interconnect using multi-objective optimization," in *2017 IEEE Congress on Evolutionary Computation (CEC)*, IEEE, 2017, pp. 2209–2216.

[169] A. K. Kodi, B. Neel, and W. C. Brantley, "Photonic interconnects for exascale and datacenter architectures," *IEEE Micro*, vol. 34, no. 5, pp. 18–30, 2014.

[170] J. Navaridas, J. Lant, J. A. Pascual, M. Lujan, and J. Goodacre, "Design exploration of multi-tier interconnects for exascale systems," in *To appear in the proceedings of the 48th International Conference on Parallel Processing*, ACM, 2019.

[171] R. Trobec, R. Vasiljević, M. Tomašević, V. Milutinović, R. Beivide, and M. Valero, "Interconnection networks in petascale computer systems: a survey," *ACM Computing Surveys (CSUR)*, vol. 49, no. 3, p. 44, 2016.

[172] W. J. Dally and B. P. Towles, *Principles and practices of interconnection networks*. Elsevier, 2004.

[173] L. G. Valiant, "A scheme for fast parallel communication," *SIAM journal on computing*, vol. 11, no. 2, pp. 350–361, 1982.

[174] M. Katevenis, N. Chrysos, M. Marazakis, I. Mavroidis, F. Chaix, N Kallimanis, J. Navaridas, J. Goodacre, P. Vicini, A. Biagioni, *et al.*, "The exanest project: interconnects, storage, and packaging for exascale systems," in *2016 Euromicro Conference on Digital System Design (DSD)*, IEEE, 2016, pp. 60–67.

[175] S. Murali, D. Atienza, L. Benini, and G. De Micheli, "A multi-path routing strategy with guaranteed in-order packet delivery and fault-tolerance for networks on chip," in *2006 43rd ACM/IEEE Design Automation Conference*, IEEE, 2006, pp. 845–848.

[176] A. Stratikopoulos, C. Kotselidis, J. Goodacre, and M. Luján, "Fastpath: towards wire-speed nvme ssds," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, IEEE, 2018, pp. 170–1707.

[177] *Zynq ultrascale+ device, technical reference manual*, UG1085, v1.7, Xilinx Inc., Dec. 2017.

[178] N. Rajovic, P. M. Carpenter, I. Gelado, N. Puzovic, A. Ramirez, and M. Valero, "Supercomputing with commodity cpus: are mobile socs ready for hpc?" In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ACM, 2013, p. 40.

[179] O. Peleg, A. Morrison, B. Serebrin, and D. Tsafrir, "Utilizing the iommu scalably.," in *USENIX Annual Technical Conference*, 2015, pp. 549–562.

[180] P. Vogel, A. Marongiu, and L. Benini, "Exploring shared virtual memory for fpga accelerators with a configurable iommu," *IEEE Transactions on Computers*, 2018.

[181] *Arm cortex-a53 mpcore processor*, ID021414, Revision r0p2, ARM Ltd., Feb. 2014.

[182] D. Folegnani and A. González, "Energy-effective issue logic," in *ACM SIGARCH Computer Architecture News*, ACM, vol. 29, 2001, pp. 230–239.

[183] E. Blem, J. Menon, and K. Sankaralingam, "Power struggles: revisiting the risc vs. cisc debate on contemporary arm and x86 architectures," in *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, 2013, pp. 1–12.

[184] M. A. Laurenzano, A. Tiwari, A. Jundt, J. Peraza, W. A. Ward, R. Campbell, and L. Carrington, "Characterizing the performance-energy tradeoff of small arm cores in hpc computation," in *European Conference on Parallel Processing*, Springer, 2014, pp. 124–137.

[185] *Ultrascale architecture gth transceivers, user guide*, UG576 pp. 24, v1.6, Xilinx Inc., 2019.

[186] *Amba axi and ace protocol specification*, IHI 0022D, ID102711, ARM LTD., 2011.

[187] *10g ethernet pcs/pma*, PG068, v6.0, Xilinx Inc., Oct. 2016.

[188] I. P. Egwutuoha, D. Levy, B. Selic, and S. Chen, "A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems," *The Journal of Supercomputing*, vol. 65, no. 3, pp. 1302–1326, 2013.

[189] J. D. McCalpin, "A survey of memory bandwidth and machine balance in current high performance computers," *IEEE TCCA Newsletter*, pp. 19–25, 1999.

[190] W. Bolosky, R. Fitzgerald, and M. Scott, "Simple but effective techniques for numa memory management," *ACM SIGOPS Operating Systems Review*, vol. 23, no. 5, pp. 19–31, 1989.

[191] C. Wang, F. Mueller, C. Engelmann, and S. L. Scott, "Proactive process-level live migration in hpc environments," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, IEEE Press, 2008, p. 43.

[192] D. Abts and D. Weisser, "Age-based packet arbitration in large-radix k-ary n-cubes," in *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, ACM, 2007, p. 5.

[193] P. Hailes, L. Xu, R. G. Maunder, B. M. Al-Hashimi, and L. Hanzo, "A survey of fpga-based ldpc decoders," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 2, pp. 1098–1122, 2016.

[194] D. Yin, G. Li, *et al.*, "Scalable mapreduce framework on fpga accelerated commodity hardware," in *Internet of Things, Smart Spaces, and Next Generation Networking*, Springer, 2012, pp. 280–294.

[195] W. J. Dally, L. R. Dennison, D. Harris, K. Kan, and T. Xanthopoulos, "The reliable router: a reliable and high-performance communication substrate for parallel computers," in *International Workshop on Parallel Computer Routing and Communication*, Springer, 1994, pp. 241–255.

[196] C. E. Leiserson, Z. S. Abuhamdeh, D. C. Douglas, C. R. Feynman, M. N. Ganmukhi, J. V. Hill, W. D. Hillis, B. C. Kuszmaul, M. A. S. Pierre, D. S. Wells, *et al.*, "The network architecture of the connection machine cm-5," *Journal of Parallel and Distributed Computing*, vol. 33, no. 2, pp. 145–158, 1996.

[197] W. Jiang, "Scalable ternary content addressable memory implementation using fpgas," in *Proceedings of the ninth ACM/IEEE symposium on Architectures for networking and communications systems*, IEEE Press, 2013, pp. 71–82.

[198] P. Kermani and L. Kleinrock, "Virtual cut-through: a new computer communication switching technique," *Computer Networks (1976)*, vol. 3, no. 4, pp. 267–286, 1979.

[199] E. N. Gilbert, "Capacity of a burst-noise channel," *Bell system technical journal*, vol. 39, no. 5, pp. 1253–1265, 1960.

[200] K. Ouchi, K. Kubo, T. Mizuochi, Y. Miyata, H. Yoshida, H. Tagami, K. Shimizu, T. Kobayashi, K. Shimomura, K. Onohara, *et al.*, "A fully integrated block turbo code fec for 10 gb/s optical communication systems," in *Optical Fiber Communication Conference*, Optical Society of America, 2006, OTuK4.

[201] *Air interface for fixed and mobile broadband wireless access systems*, IEEE P802. 16e D, IEEE LAN/MAN Standard Committee, p. 2005.

[202] "Ieee standard for information technology - local and metropolitan area networks - part 3: csma/cd access method and physical layer specifications - media access control (mac) parameters, physical layer, and management parameters for 10 gb/s operation," *IEEE Std 802.3ae-2002 (Amendment to IEEE Std 802.3-2002)*, pp. 1–544, 2002.

[203] F. Demangel, N. Fau, N. Drabik, F. Charot, and C. Wolinski, "A generic architecture of ccsds low density parity check decoder for near-earth applications," in *2009 Design, Automation & Test in Europe Conference & Exhibition*, IEEE, 2009, pp. 1242–1245.

[204] T. Brecht, "On the importance of parallel application placement in numa multiprocessors," in *Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS IV)*, 1993, pp. 1–18.

[205] T. David, R. Guerraoui, and V. Trigonakis, "Everything you always wanted to know about synchronization but were afraid to ask," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ACM, 2013, pp. 33–48.

[206] J. Pješivac-Grbović, T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel, and J. J. Dongarra, "Performance analysis of mpi collective operations," *Cluster Computing*, vol. 10, no. 2, pp. 127–143, 2007.

[207] J. Williams, C. Massie, A. D. George, J. Richardson, K. Gosrani, and H. Lam, "Characterization of fixed and reconfigurable multi-core devices for application acceleration," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 3, no. 4, p. 19, 2010.

[208] *10g/25g high speed ethernet subsystem*, PG210, v2.1, Xilinx Inc., Apr. 2017.

[209] K. Townsend and J. Zambreno, "Reduce, reuse, recycle (r 3): a design methodology for sparse matrix vector multiplication on reconfigurable platforms," in *2013 IEEE 24th International Conference on Application-Specific Systems, Architectures and Processors*, IEEE, 2013, pp. 185–191.

[210] D. Strenski, "Fpga floating point performance–a pencil and paper evaluation," *HPC Wire*, Jan. 2007.

[211] M. Marazakis, J. Goodacre, D. Fuin, P. Carpenter, J. Thomson, E. Matus, A. Bruno, P. Stenstrom, J. Martin, Y. Durand, *et al.*, "Euroserver: share-anything scale-out micro-server design," in *Proceedings of the 2016 Conference on Design, Automation & Test in Europe*, EDA Consortium, 2016, pp. 678–683.

[212] N. Kallimanis, M. Marazakis, and E. Skordalakis, "Use-cases for remote memory in the unimem architecture," in *ExascaleHPC: the ExaNoDe, ExaNeSt, EcoScale, and EuroEXA projects workshop at HiPEAC, Manchester*, 2018.

[213] J. A. Pascual, J. Lant, C. Concatto, A. Attwood, J. Navaridas, M. Luján, and J. Goodacre, "On the effects of allocation strategies for exascale computing systems with distributed storage and unified interconnects," *Concurrency and Computation: Practice and Experience*, e4784,

[214] A. B. Nagarajan, F. Mueller, C. Engelmann, and S. L. Scott, "Proactive fault tolerance for hpc with xen virtualization," in *Proceedings of the 21st annual international conference on Supercomputing*, ACM, 2007, pp. 23–32.

[215] Y. Liang, Y. Zhang, A. Sivasubramaniam, M. Jette, and R. Sahoo, "Bluegene/l failure analysis and prediction models," in *International Conference on Dependable Systems and Networks (DSN'06)*, IEEE, 2006, pp. 425–434.

Appendix A

Project Context

The work described in this thesis is situated within the context of a set of pan-European projects tasked with the design and build of a system employing cutting edge computer architectures to target future generations of exascale class systems. These projects aim to push forward an "everything close" and "share anything" philosophy [174] by various means, doing this as a key way of reducing data movement costs and maximising energy efficiency. Together these projects pay service to the whole stack, viewing the system holistically, from the cooling technology and applications down to the development of custom compute elements. This path to European exascale computing began with several Horizon 2020 projects[1] in 2015 targeting numerous different areas; such as storage, networking, acceleration, node density, cooling etc. Current projects are at the stage of building larger prototypes to demonstrate the scalability of these systems. These projects are:

ExaNoDe [164]- Tasked with the design of compute elements and runtime systems to support low power, heterogeneous HPC systems. The key elements of this project for reducing power consumption are the use of 3D package integration and the use of low power ARM cores to increase component density. As well as this the project further develops the use of the *UNIMEM* memory model [211], which provides a PGAS-like global address space and overcomes the scalability issues associated with global cache coherence strategies.

ExaNeSt [163]- This project is involved in the networking and storage aspects of

[1]What is Horizon 2020?- `https://ec.europa.eu/programmes/horizon2020/what-horizon-2020`, Accessed November 2019.

these future heterogeneous HPC systems. The central aim being to create a unified interconnect to carry all networking, storage and management traffic. The project's use of hyperconverged storage and the UNIMEM model aim to reduce the power consumption of the system interconnect, and bring compute and data closer together.

ECOSCALE [92]- The ECOSCALE project aims to create a scalable programming environment for reconfigurable accelerators in HPC systems. They have created a runtime system which enables the use of distributed FPGAs by use of multiple *Workers* in a partitioned global address space (PGAS), supporting MPI+OpenCL. These workers are able to use the runtime system to take ownership of remote and local logic resources, using partial reconfiguration in a fashion which allows multiple applications to utilise FPGA resources by dynamically scaling and reallocating acceleration tasks.

EuroEXA [165]- The EuroEXA project aims to create a larger scale system prototype based upon the technology and findings of the other listed projects. This project will create a multicore ARM+FPGA based compute board, integrating customized networking capability supporting a global address space and the UNIMEM memory model, developed in the ExaNeSt project. As well as this a custom ARM-based SoC ASIC is being designed and taped out, to be coupled with Xilinx accelerators for a future system testbed. The overarching goal of this project is to demonstrate a system capable of scaling to a peak performance of 400 PFLOPS with a peak system power envelope of around 30MW (roughly 4x the efficiency of today's top systems).

Node and Memory Model

It can be seen here that our use of the Xilinx Zynq Ultrascale+ FPGA, and our desire to provide a means for FPGAs to communicate with one another efficiently in a distributed manner comes from this push to create a custom compute element for HPC. This compute element will utilize low-power ARM cores tightly coupled to a reconfigurable fabric. By using acceleration elements so closely coupled to the unified interconnect enables extreme data processing as well as extreme compute

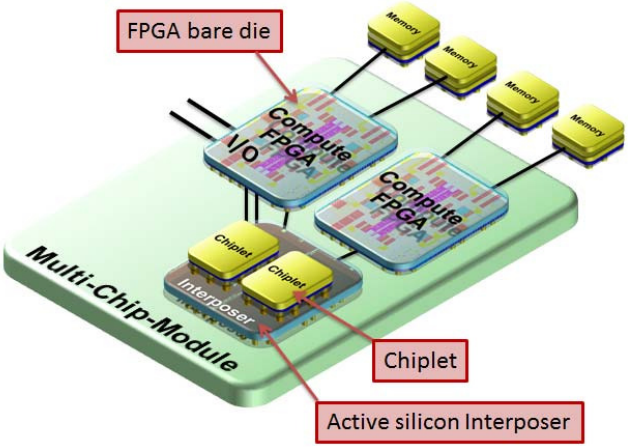


Figure A.1: Compute node developed in the ExaNoDe project. Taken from [164].

capabilities [165].

The compute node will be based upon the work performed in the ExaNoDe project, shown in Figure A.1, which uses a densely packaged 3D silicon interposer solution forming a multi-chip modules, which consists of tightly coupled ARM cores and FPGA used for compute and IO capabilities.

The node will communicate using the novel UNIMEM memory model, first developed in the EUROSERVER project [211]. In the UNIMEM model remote communication can be performed via load/store instructions or via RDMA for large data copying across the network. The model is scalable by allowing cacheability of physical pages only at a single node, which eliminates the need for complex hardware coherence protocols spanning the network. The Unimem API resembles that of a shared-memory system, and can be used by applications for synchronization and utilizing the remote memory [212].

System Architecture and Unified Interconnect

As we discussed in section 3.1, the targeted system architecture for these set of projects is designed to increase the energy efficiency. Properties of the system which promote this includes a hierarchical network topology with hyper-converged storage and a unified interconnect. The storage is kept close to the compute to reduce data movement and thus reduce power consumption and increase performance [213]. By unifying the typically separate network and storage interconnects

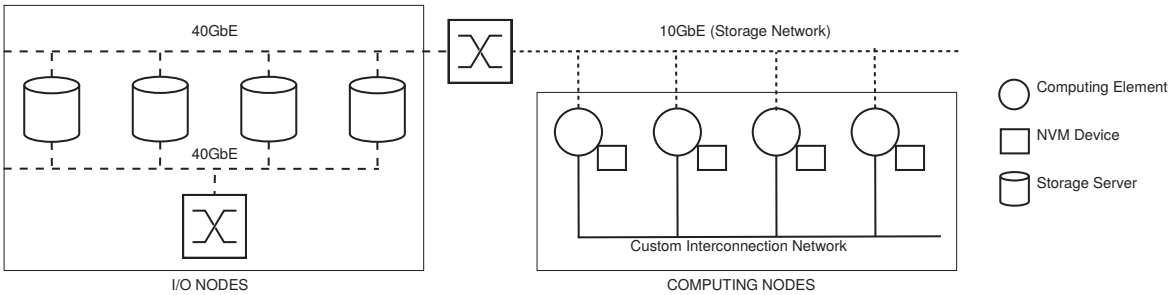


Figure A.2: The ExaNeSt storage architecture. The local NVMs are attached to the computing nodes sharing the main IN (solid).An Ethernet network is provided for central data storage. Taken from [213].

the amount of overall physical cabling is drastically reduced and thus the static power consumption of the network is reduced, even when idle.

Figure A.2 shows the storage architecture for the network. Fast NVMe (Non-Volatile Memory express) SSD disks are connected to each compute node, to be used by applications for distributed storage, accessible by local or remote nodes. This uses the BeeGFS distributed file system, originally developed at Fraunhoefer[2]. An additional storage server will need to be provided for long term storage of data, which would connected using less performant standard Ethernet technology.

The main consequence of the use of this unified interconnect on the work within this thesis is the desire to support a fully adaptive routing strategy for the custom network, as a means of increasing the performance of the system. Many prior works have characterised scientific workloads and shown that point-to-point messages form the majority of bulky transfers, with collective operations typically requiring only very small payloads [53]. Given the fact that the majority of connection based transports and standard table based routing will only allow for a single path to be used per flow, this suggests that any very heavy traffic flows (such as those which would be seen when writing data to disk) will not utilise bandwidth efficiently unless multiple paths are available for individual packets within a given flow. It is for this reason that we support a fully adaptive routing scheme and out-of-order packet delivery, to enable better load balancing and make more bandwidth available to large RDMA transfers.

[2]An introduction to BeeGFS, by Jan Heichler, November 2014- Available at `http://www.beegfs.de/docs/whitepapers/Introduction_to_BeeGFS_by_ThinkParQ.pdf`, Accessed November 2019.

Appendix B

Addressing on the Zynq Ultrascale+

Mapping

In order to cope with the fact that that we do not have virtual-physical mapping capability set up for peripheral devices within the system we provision for the use of a distributed, shared-memory system by use of a small region of physical memory, mapped into a user-space application's virtual memory.

This is done by use of the `mmap()` system call, and a userspace virtual-to-physical mapping function to retrieve the physical address. This physical address is then passed to the DMA engine or accelerator block, in order to provide access to a dedicated region of DRAM. We first must allocate a region of virtual memory, and using the process id we can then check the pagemap for the physical frame number using the virtual page in `/proc/<process ID>/pagemap`[1]. It should be noted that each page (4KB in our instance) of memory has its own mapping, and that this is not necessarily contiguous. In order to have contiguous physical memory over 4KB boundaries we must have the kernel allocate the memory using `kmalloc()`, and then have a driver expose an `mmap()` interface to the memory, similar to how we map the hardware on the FPGA into userspace.

The hardware for the NI and other FPGA resources is exposed through two physical `mmap()`'d regions. In doing this we gain read/write access to the configuration and data registers of the custom IP blocks on the FPGA fabric through a simple

[1]Kernel docuentation on pagemap- `https://www.kernel.org/doc/Documentation/vm/pagemap.txt`, accessed April 2019.

pointer with a virtual address in the application. Two separate 1MB physical regions are mapped; `A0000000` and `B0000000`, and are routed to the AXI master ports of the PS-PL interfaces on the device *M_AXI_HPM0_FPD* and *M_AXI_HPM1_FPD* respectively (see the HPM interfaces in Figure. B.1). These can be accessed using two generic user-space io device files, which are created by extending the Device Tree (Listing B.1) to expose the regions as character devices `/dev/uio0` and `/dev/uio1`[2].

Listing B.1: Device tree entries for access to devices in the programmable logic.

```
aptCores@a0000000 {
        compatible = "generic-uio";
        reg = <0x0 0xa0000000 0x0 0x100000>;
        interrupts = <0x0 0x39 0x0>;
        interrupt-parent = <0x2>;
};
aptCores@b0000000 {
        compatible = "generic-uio";
        reg = <0x0 0xb0000000 0x0 0x100000>;
        interrupts = <0x0 0x39 0x0>;
        interrupt-parent = <0x2>;
};
```

These character devices are then accessed in the user-space application by use of the `open()` system call, returning a file descriptor which is used by the `mmap()` function to provide a virtual address at which the user can write to the hardware within the reconfigurable fabric of the FPGA.

The downside of the lack of a configured IOMMU within our prototype system, and this static mapping in user-space for both hardware and RDMA buffering is that the solution requires memory pinning for the application's RDMA buffers. The virtual to physical mapping must remain static because the hardware within the FPGA (DMA engine or accelerator logic) must access the memory subsystem of the CPU using physical addresses. Pinning the memory is initially expensive to allocate and deallocate, but as the buffers are intended for reuse over the lifetime of

[2]Userspace I/O- `https://wiki.krtkl.com/index.php?title=Userspace_I/O`, accessed November 2018.

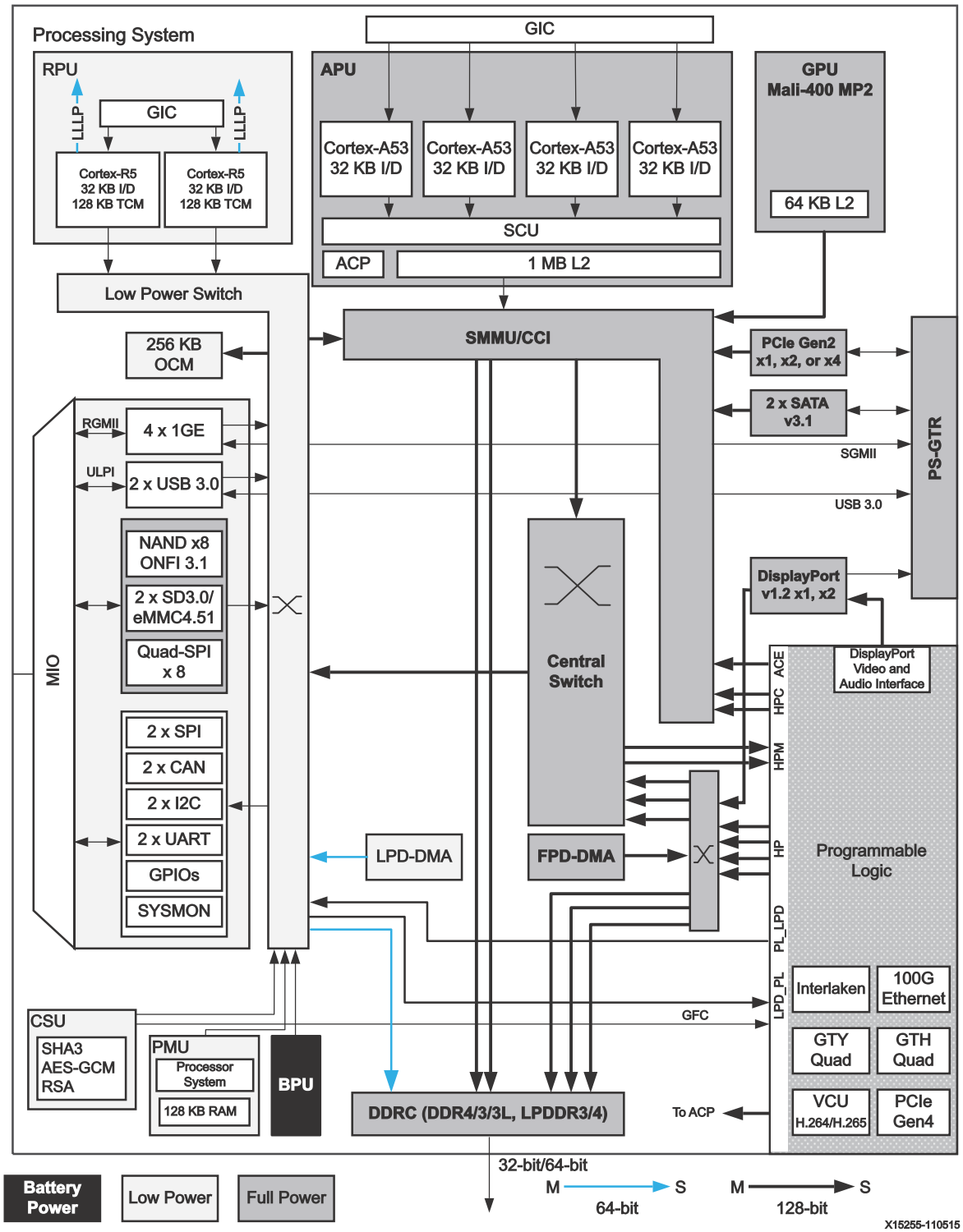


Figure B.1: Block Diagram of the Zynq Ultrascale+, taken from [177] (pp. 18).

the application, this should be amortized following Amdhal's law. The main issue with this technique when in the context of a large, distributed system is the fact that it prevents important fault tolerance techniques being employed such as page migration [191], [214]. This means that in the event of a failing node (identified using preemptive failure detection techniques [215]) any process using the failing node could not move to a new one.

Addressing Limitations

The device is unable to route virtual addresses to the PL from the PS. There is also a finite physical window of memory which can be used to access the PL from any given interface port[3]. Figure B.2 shows the address map for the device. As such the complete map of addresses for a fully global, shared virtual address space cannot be adequately covered using the IOMMU within the PS only (named the System MMU/SMMU in the Zynq reference manual [177]). While these physical window limitations are not an issue for the limited prototype described within this thesis, a complete system would require a larger addressable window of global memory, or the ability to route virtual addresses out into the logic fabric.

RDMA operations can be performed unchanged within the system, as the DMA engine can operate using virtual source and destination addresses. If the SMMU inside the PS is configured correctly, then virtual addresses can be used by the DMA engine. This is because the slave interfaces between the PL and PS pass through the TBU (Translation Buffer Unit), which store page table lookups (see Figure 3.3). The DMA engine is able to use virtual addresses with no modification because the physical mapping of the DMA ring buffer is locally static. Work for the DMA engine is provided as data in AXI write requests, which can obviously contain pointers to virtual memory locations.

However, utilizing a global address space for remote shared-memory operations is not as simple. The shared memory port for the NI must be accessed using a physical address. So a static, large region must be mapped to access the NI. In the complete system a page table will be required to maintain mappings in the

[3] 224GB for M_AXI_HPM0_FPD and M_AXI_HPM1_FPD interfaces at 0x0010_0000_0000 and 0x0048_0000_0000 respectively.

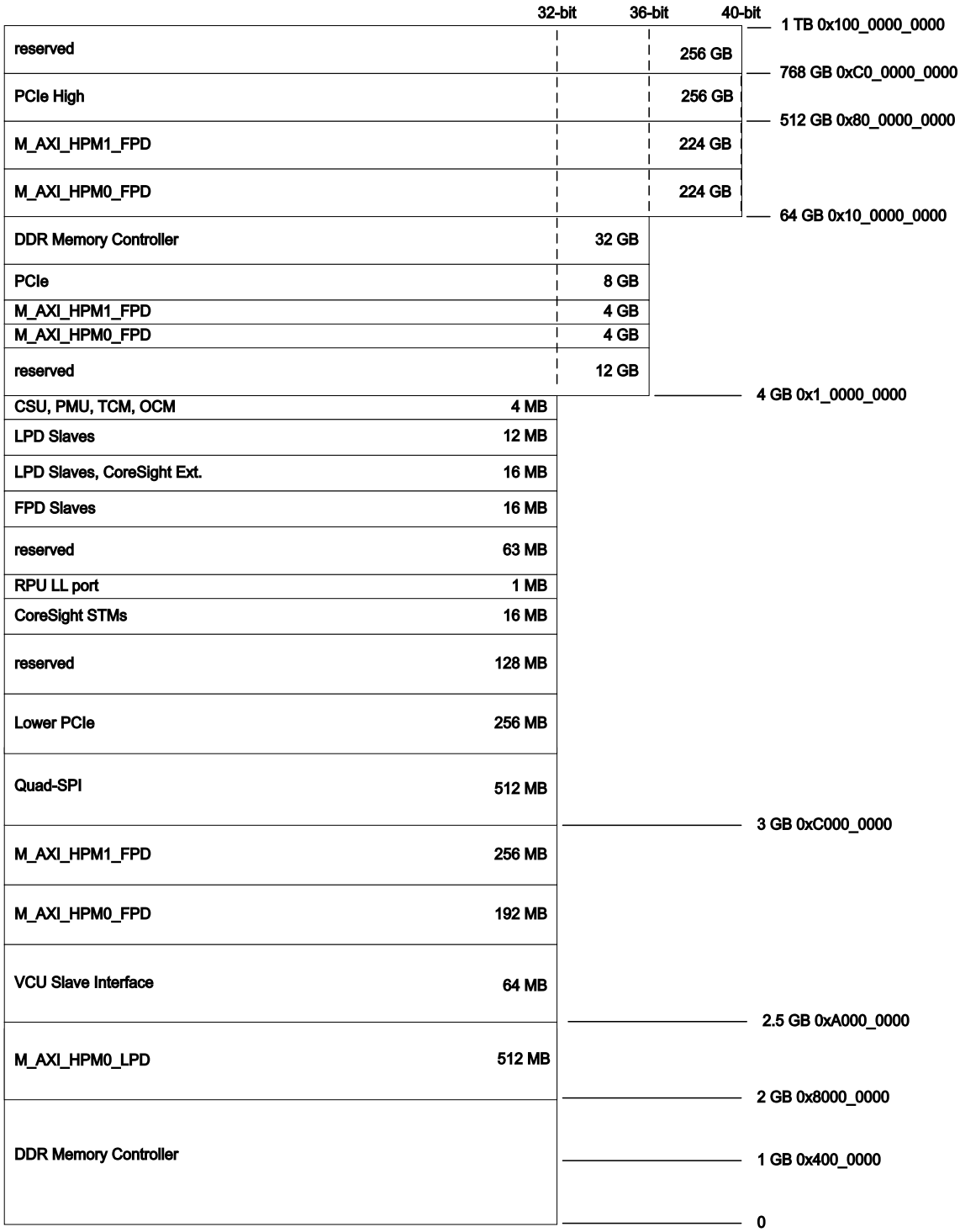


Figure B.2: Gloabl address map for the Xilinx Zynq Ultrascale+, taken from [177] pp. 222.

programmable logic that enable a translation from a physical address back to a global virtual address.

Appendix C

AXI 4 Interface Standard

The AXI protocol consists of 5 completely independent channels of communication between a master-slave pair (Figure C.1). Three of the channels form master to slave requests; Read Address **(AR)**, Write Address **(AW)** and Write Data **(W)**. The two remaining channels provide slave to master responses; Write Response **(B)** and Read Data **(R)**. Transfer of address/data/response accross the interface is performed using simple ready/valid handshaking. Multiple slaves and masters can be interconnected, and route using either memory mapping **(AxADDR)** in the master to slave request side, or using ID signals **(AxID/RID/BID)** to route the corresponding responses back to the correct master. As there is no ID signal on the write data **(W)** channel (discontinued in AXI4), this means that write transactions cannot be interleaved. The same is not true for reads, although the virtual cut-through nature of the proposed network switch (described in Section 3.1.3) means this is the case regardless.

The AXI interface is flexible, enabling connections between master/slaves with differing requirements. The address channels require one ready/valid exchange per transfer, and provides information about data width **(AxSIZE)**, burst length **(AxLEN)**, memory type **(AxCACHE)**, quality of service **(AxQOS)**, atomic access **AxLOCK** and security/privilege levels **(AxPROT)**, as well as user defined signals for any other purpose **(xUSER)**. The data is clocked out in the number of beats/phits for the corresponding transaction, as defined by the burst length signal AxLEN (an example write and read transaction are shown in Figure C.2). A full breakdown of the signals and their function is shown in Table C.1.

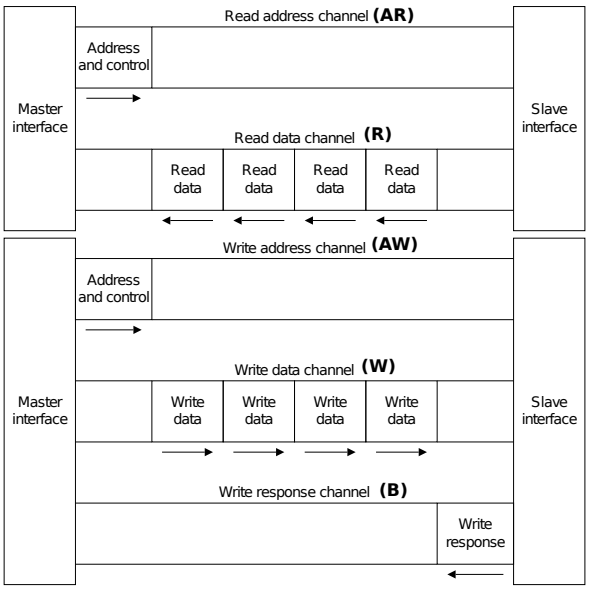
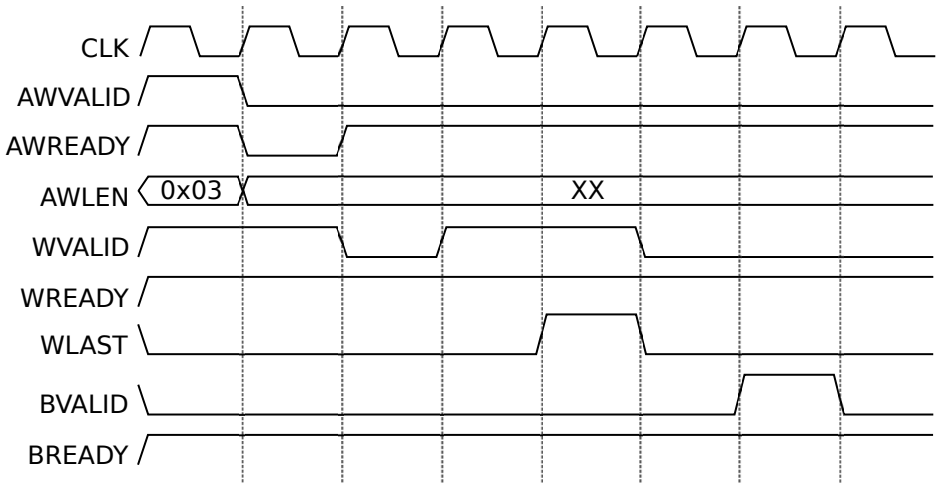
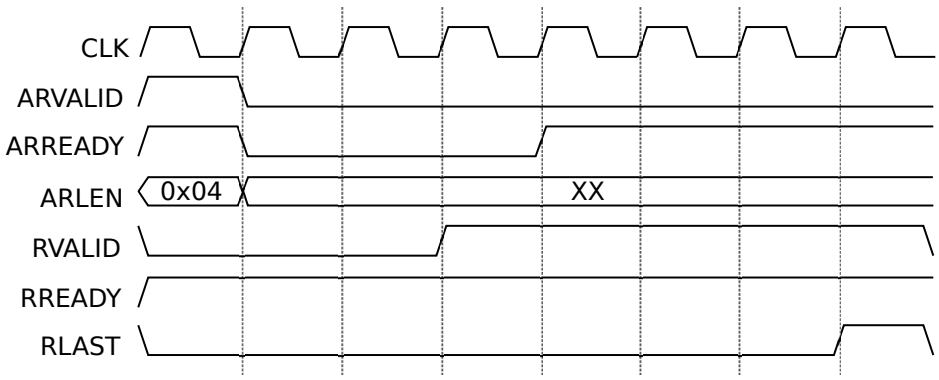


Figure C.1: Independent request and response channels in the AXI interface, taken from [186].



(a) AXI write.



(b) AXI read.

Figure C.2: Example timing diagram of AXI transactions.

Table C.1: AXI signals and the corresponding channels with which they are associated.

Name	**Channels (AW,AR,W,R,B)**					**Description**
xID	✓	✓		✓	✓	ID signals to identify the originating master of a transaction.
xADDR	✓	✓				Base destination address for the transaction
xLEN	✓	✓				Burst length/number of beats of data in a transaction. Total Transaction Data = xLEN*bitWidth(xData).
xSIZE	✓	✓				Number of bytes in a single beat/transfer.
xBURST	✓	✓				Burst type, FIXED = same address for all transfers, INCR = standard transfer incrementing address of each beat, WRAP = same as INCR, but wraps if an upper limit is reached.
xCACHE	✓	✓				Defines memory type of the transaction, bufferable/non-bufferable, cacheable/non-cacheable, write-through/write-back, read-/write allocation.
xPROT	✓	✓				Access permissions, privilieged/unprivileged, secure/non-secure, and data/instruction access.
xLOCK	✓	✓				Provides locked/exclusive accesses to slaves.
xQOS	✓	✓				Supports Quality of Service levels in the interconnect or peripheral.
xREGION	✓	✓				Used to create multiple logical interfaces from a single physical interface, each having a different location within the system address map.
xDATA			✓	✓		Data being transferred, width of signal is flexible.
xLAST			✓	✓		Indicates last transfer of data in a transaction.
xSTRB			✓			Byte-lane validity for write transactions.
xRESP				✓	✓	Detail on whether transaction completed correctly, OKAY = correct, EXOKAY = correct exclusive access, SLVERR = transaction failed at slave, DECERR = transaction failed in interconnnect, slave unreachable/unidentifiable.
xREADY	✓	✓	✓	✓	✓	Indicates the slave is ready to accept new requests/master can accept new responses.
xVALID	✓	✓	✓	✓	✓	Indicates the master has valid request/slave has valid response.
xUSER	✓	✓	✓	✓	✓	User defined signals.

Appendix D

Controllers for AXI-Network Protocol Bridging

As discussed in Section 3.3.2.1, four controllers are used to perform bridging between the on-chip AXI protocol, and the custom, off-chip network packet. The four controllers which perform the following translations are:

- Sender
  - Memory-mapped AXI request to network packet.
  - Memory-mapped AXI response to network packet.
- Receiver
  - Network packet to memory-mapped AXI request.
  - Network packet to memory-mapped AXI response.

Each of the four controllers has five main states, **HEAD1**, **HEAD2**, **BODY**, **FOOT1**, **FOOT2**, and they handle both shared-memory operations as well as RDMA traffic. The state transitions are described for each of the controllers in Figure D.1 and Tables D.1- D.4. For the receive side translation which handles incoming traffic from the network, information is stored from the header and footer in order to rebuild the associated AXI transaction that was posted at the sender. The body (payload) is sent to a buffer within the NI as it enters. This is required as a store-and forward mechanism is used for several reasons. *(i)* The tail contains a packet-level CRC hash

to detect soft-errors which may be introduced to the packet as it traverses the network, so the store and forward mechanism prevents erroneous data being written to memory. *(ii)* There are several key pieces of information which are stored within the tail of the packet which are needed to rebuild the initial transaction. *(iii)* In Section 6.2, a mechanism for increasing the performance of operations at the recevier is shown. A key part of this mechanism prevents duplicate packets from being written to memory. Since information in the tail of the packet is required, again we must store the whole packet to prevent memory inconsistencies.

At the send side the state machine is used to convert a single AXI transaction into a network packet. The AXI address information required for the 128 bit header and footer is presented to the interface in a single cycle, but they are built in four cycles. This has no effect on the performance of the NI, because as the packet is being built each flit is sent immediately to the output queue, which can only be fed into the 10G MAC layer 64 bits per cycle. The data is sent directly to the output buffer as well. However, due to the fact that the AXI write data **(W)** and write address **(AW)** channels are completely independent of one another, the data can arrive before addressing information. As such the data must only be sent to the output buffer during the **BODY** state, after the header has been formed and pushed to the MAC.

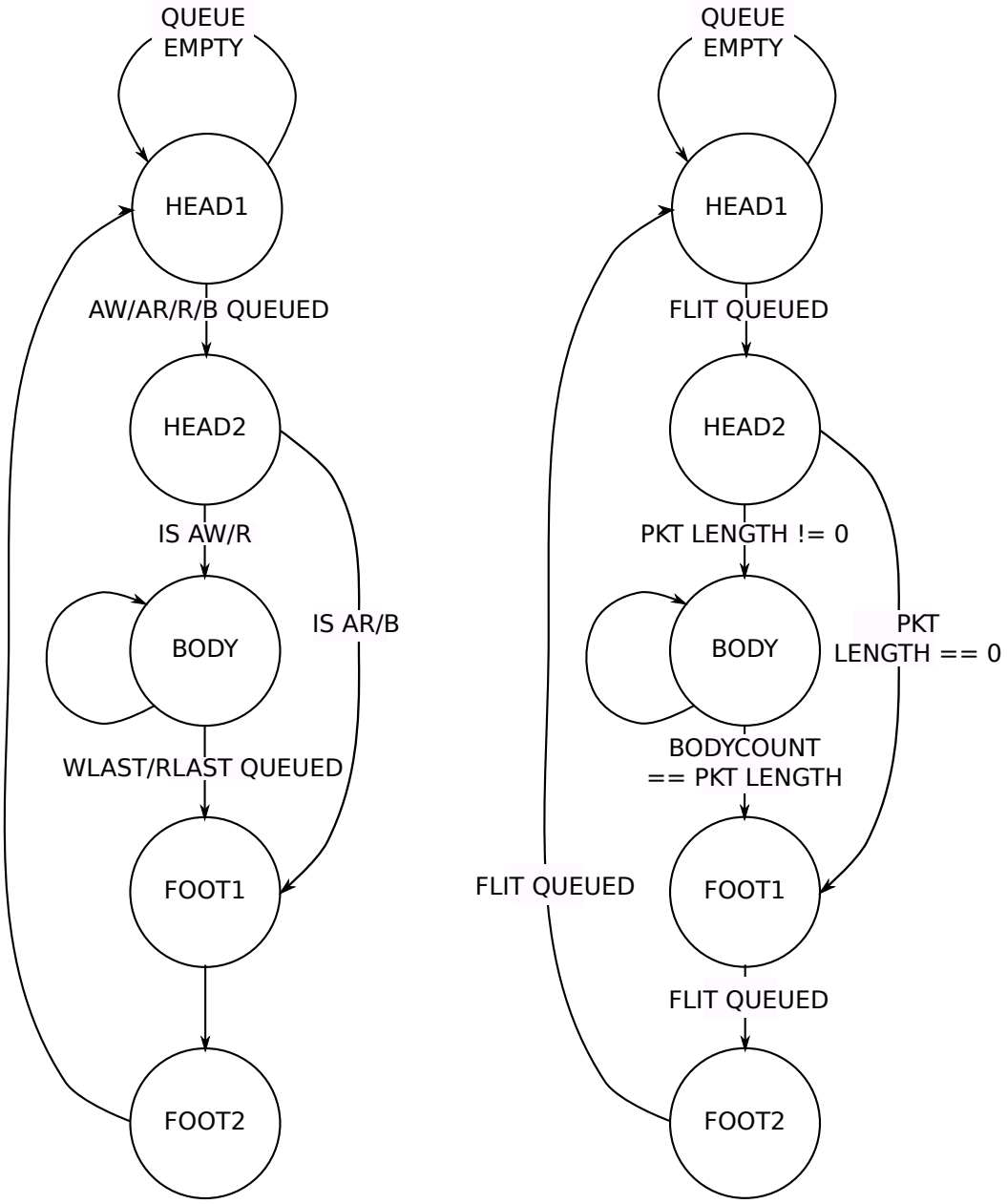


Figure D.1: State transition diagrams for the controllers which bridge between memeory mapped AXI and the network packet (left), and from network packets into memory mapped AXI transactions (right).

Table D.1: Controller for processing AXI requests to send to the network.

State	Action
HEAD1	Check the status of any requesting queue, take from; shared memory retransmission, shared memory read, shared memory write, RDMA retransmission or RDMA transfer queue with descending priority respectively. If shared memory operation add to table for retransmission and issue a new ID. If first packet of RDMA operation add relevant data to table to enable rebuild of operation in event of failue (base address, number of packets expected). If subsequent RDMA packet increment the number of transmitted packets in the operation.
HEAD2	Get the new transaction ID for the packet. Store to place in the footer.
BODY	start draining the data queue corresponding to the packet (RDMA queue, retransmission queue, shared memory queue).
FOOT1	Simply build flit from stored request information.
FOOT2	If retransmissing RDMA packet, dequeue the information required to rebuild the operation.

Table D.2: Controller for processing AXI responses to send to the network.

State	Action
HEAD1	Match the response ID against the response table entries. Locate the original ID and source node ID. Build flit from relevent information and push to network.
HEAD2	Simply build flit from relevant information.
BODY	If read data, drain from the input queue, until count equals expected number of data transfers.
FOOT1	Simply build flit from relevant information.
FOOT2	Simply build flit from relevant information.

Table D.3: Controller for processing incoming network request packets.

State	Action
HEAD1	Check the packet type incoming, store expected payload length.
HEAD2	If packet type is DMA_REGISTRATION, send relevant information to registration module (base address, total packets in registered operation). If packet type is RDMA_WRITE, send to the registration module to check against other entries.
BODY	Every flit increments the count of the data. Wait until the expected number of flits arrives from the payload length. Write the data into an output buffer.
FOOT1	If request is a shared memory operation, add relevant information to table to build the response packet (source node, original transaction ID).
FOOT2	If shared memory operation retrieve the new ID issued for the AXI transaction.If DMA operation, add relevant information to table to build response packet (source node, original transaction ID, dma operation issue number).

Table D.4: Controller for processing incoming network response packets.

State	Action
HEAD1	Check packet type. Store relevant information depending on type; Shared memory response, RDMA response, Completion notification.
HEAD2	If the packet is a notification of completion, store relevant information for building notification.
BODY	Drain the input into a buffer for storage. Must check the data is correct before writing to memory.
FOOT1	If the packet is a notification of completion, store relevant information for building notification. If the response is from a DMA operation or Shared memory operation push relevant information into the request table to get the old ID back.
FOOT2	If the response is from a DMA operation or Shared memory operation get the old ID back from the request table. If the response is a negative acknowledgement start the retransmission process. Add the entry back to the request table and reissue a new ID, and then send to retransmission request queue.